Une application de l'intelligence artificielle à la synthèse architecturale des circuits intégrés VLSI

Alain Blaise Fonkoua

THESE

présentée par

FONKOUA Alain Blaise

Ingénieur ENSIMAG

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 5 juillet 1984)

(Spécialité : **INFORMATIQUE**)

==========

Titre : Une application de l'intelligence artificielle à la synthèse architecturale des circuits intégrés VLSI

==========

Date de soutenance : 4 octobre 1989.

Composition du Jury :

Mr. J. MOSSIERE	Président
Mr. R. GERBER	Examinateurs
Mr. J. P. LAURENT	
Mr. G. MAZARE	
Mr. SAGNES	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

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

E.F.P.
CHARUEL Robert

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. NEMOZ Alain

Année Universitaire 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique,
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures

JOSELEAU Jean Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J.Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodymamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. I.U.T.1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA: IUT 1

PROFESSEURS de 2ème classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
LEVIEL Jean Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci

HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.

MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

Remerciements

Je tiens à remercier en particulier :

- M. Jacques MOSSIERE directeur de l'ENSIMAG pour l'honneur qu'il me fait de présider mon jury de thèse

- M. Roland GERBER chef de la division CCI du CNET Grenoble pour l'intérêt et les encouragements qu'il m'a donnés au sujet de la thèse et du travail effectué. Je suis flatté qu'il ait accepté d'être rapporteur et membre du jury de cette thèse.

- M. George SAGNES professeur à l'université de MONTPELLIER qui a accepté de rapporter cette thèse et de faire partie du jury.

- M. Jean Pierre LAURENT professeur à l'université de CHAMBERY, membre du jury de thèse, pour les suggestions interessantes qu'il m'a faites concernant la suite du travail.

Durant cette thèse, j'ai bénéficié de l'encadrement de M. Guy MAZARE mon directeur de thèse qui m'a aidé tant sur le plan académique que financier. Les réunions que nous avons eu ont été décisives pour l'évolution du travail. Je suis particulièrement touché de la confiance qu'il m'a faite pendant toute la durée de cette thèse.

Le travail présenté dans ce mémoire a été effectué au Centre National d'Etudes des Télécommunications où j'ai pu bénéficier d'un cadre de travail agréable et des conseils de plusieurs chercheurs du centre. Qu'ils trouvent tous ici, l'expression de ma sincère reconnaissance. En particulier, j'aimerais remercier :

- M. Denis ROUQUIER chef du département AMS qui a montré beaucoup d'intérêt pendant cette thèse. Les discussions que nous avons eu m'ont été profitables. D'autre part je suis particulièrement reconnaissant pour le sérieux avec lequel il a relu le manuscrit de cete thèse.

- M. Vincent OLIVE ingénieur au CNET Grenoble qui a suivi mon travail pendant toute la durée de la thèse. Il m'a assuré des conditions travail exceptionnelles, sans compter les conseils et les réunions qui ont été déterminants pour l'avancement du travail.

- plus généralement, la proximité de l'équipe ADA (Jacques ROUILLARD, Jean Michel BERGE, Louis Olivier DONZELLE et Vincent OLIVE) m'a permis bien des fois de surmonter certaines difficultés de programmation.

- Je suis redevable à Louis Olivier DONZELLE, Alain PUISSOCHET, Pierre François DUBOIS dont les remarques m'ont permis d'améliorer la rédaction et la présentation de ce mémoire.

- Je ne saurais finir sans souligner la formidable ambiance qui régnait au sein du département...

à maman
à papa

Abstract : This thesis is concerned with the use of artificial intelligence (AI) methods for the automatic synthesis of integrated circuits architectures.

Once the architectural synthesis problem has been introduced, an analysis of the encountered difficulties justifies the use of artificial intelligence methods to automate this task.

Next, some AI concepts are displayed through a critical description of a few expert system generators (PROLOG, OPS5, TROPIC...). These concepts are cast in an expert system building tool called ODSE, conceived and programmed in ADA. The knowledge representation formalism, the inference mechanisms, and the implementation are described along with a justification of the choices made.

Lastly, the dissertation shows the use of ODSE to realize a tool for Architectural Synthesis Aid (ASA). ASA takes as its input a functionnal description of an IC and yields a structural description of an architecture after speed/space trade-offs to conform to user defined constraints and available hardware resources in the given chip library. The methods used by ASA (dataflow analysis, scheduling, resources allocation) are showed and their expression using ODSE formalism given. The ASA aproach is next compared to others used in similar tools. Results are also given for a test case using ASA to synthesize a numeric 9th order LEAPFROG filter.

Key-words : silicon compilation, VLSI integrated-circuit, architectural synthesis, datapath, artificial intelligence, expert systems, ADA language, object oriented conception.

1. Évolution des outils de CAO VLSI vers la compilation de silicium

1.1. Introduction

Le but ultime de la conception des circuits intégrés est de fournir la description complète des masques qui seront utilisés pendant la phase de fabrication pour les produire [anc84].

Les progrès rapides et continus de la technologie de fabrication des circuits intégrés ont permis de multiplier la densité d'intégration (nombre de transistors gravés sur une surface de 1 mm2 de silicium) par un facteur variant de 1,6 à 2 chaque année et ce depuis les années 60 (cf figure 1). Comme conséquence de cet essor, la complexité croissante des circuits a nécessité l'élaboration de méthodes de conception spécialisées et difficiles à maîtriser, à l'instar de la programmation structurée mise au point par les informaticiens pour la conception de logiciels complexes (préface française [MEA79]). La figure 2 schématise les différentes étapes de l'activité de conception de circuits intégrés depuis la spécification initiale du circuit jusqu'à la génération des masques. On peut les classer en 4 phases [wec86] :

—synthèse au niveau système : à partir d'un cahier des charges décrivant la fonction à réaliser(description fonctionnelle), le circuit est analysé et décomposé en sous ensembles fonctionnels de complexité moindre ; les connexions entre les sous systèmes sont complètement spécifiées ; les sous systèmes complexes sont à leur tour décomposés...A la fin de l'étape, le circuit est spécifié sous une forme dite "structurelle", c'est-à-dire comme un ensemble de modules interconnectés ;

—synthèse logique : chacun des modules issus de l'étape précédente est entièrement spécifié au niveau logique à l'aide de portes logiques (and, or, nor, etc) ;

—synthèse électrique : à partir de la description logique précédente, le circuit est entièrement spécifié sous la forme d'un réseau de transistors interconnectés ;

—conception du plan de masse et dessin des masques : le schéma électrique issu de l'étape précédente est spécifié au micron sous la forme d'un jeu de masques ; les cellules sont dessinées, l'ensemble est placé et les interconnexions tracées de manière à occuper une surface minimale.

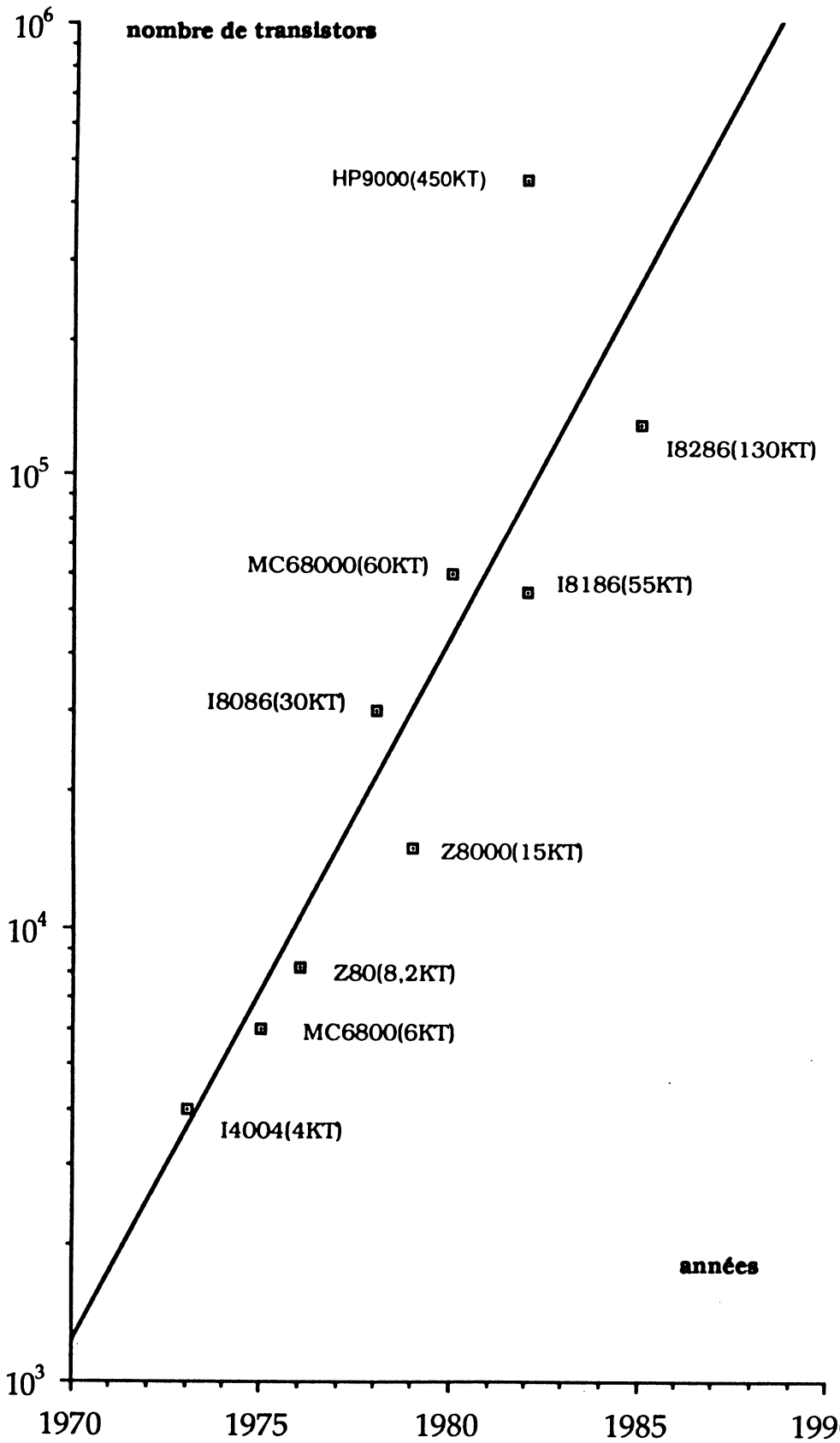


Figure 1 : Croissance en complexité des C.I. (tiré de [Anc 84])

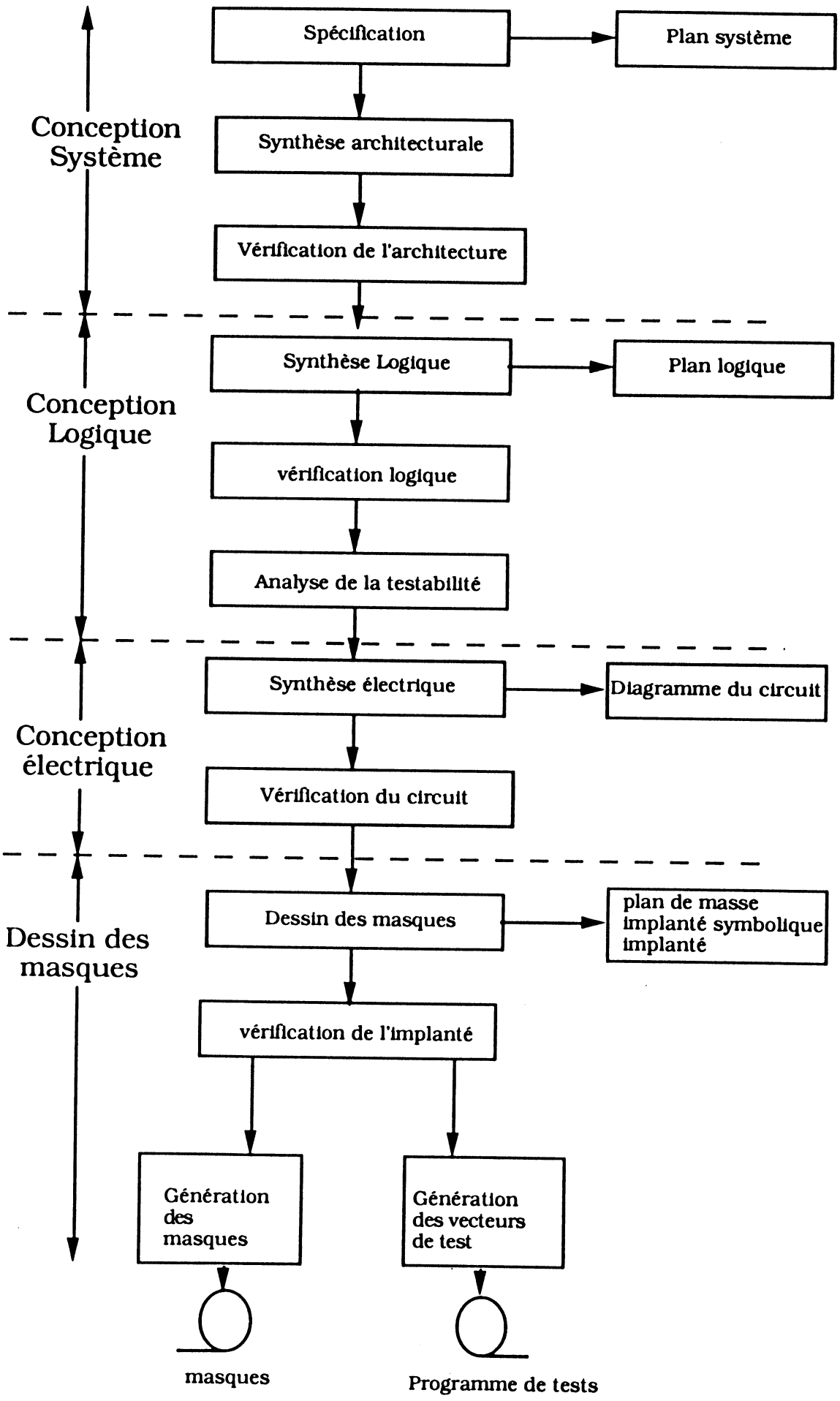


Figure 2 : étapes générales de conception d'un Circuit Intégré [Hor 86]

Chacune de ces phases est assortie d'une étape de vérification pour s'assurer que :

—la représentation du circuit généré est conforme à la spécification et aux contraintes issues de l'étape précédente ; on parle de compatibilité verticale [WEC86] ;

—la représentation du circuit en cours de synthèse ne viole pas les règles de conception imposées par la technologie de fabrication (compatibilité horizontale) ; cette vérification est cruciale pour les phases 3 et 4 qui sont les plus dépendantes de la technologie.

La non conformité à l'un de ces tests engendre une nouvelle synthèse et peut entraîner une remise en question de l'étape précédente qui doit alors être reprise.

Les phases 1 et 2 ci-dessus décrites sont souvent désignées par le terme de "synthèse logique" et sont traditionnellement accomplies par les ingénieurs système alors que les phases 3 et 4 qui requièrent des compétences en micro-électronique sont recouvertes par le vocable de synthèse physique [wec86].

Il ressort de cette présentation succincte que la conception d'un CI est une activité longue et coûteuse nécessitant un personnel qualifié ; c'est pourquoi, très tôt, une intense activité de recherche s'est développée dans le domaine de la conception assistée par ordinateur de circuits intégrés à haute intégration (VLSI : "very large scale integration") visant à :

—libérer les concepteurs de tâches fastidieuses et répétitives : dessin des masques, vérification des règles de dessins, tests de compatibilité verticale...

—automatiser tout ou partie de l'activité de conception.

Cet effort de recherche a permis le développement de systèmes intégrés de C.A.O de circuits intégrés V.L.S.I par exemple CASSIOPEE (développé au CNET Grenoble) avec :

—des éditeurs graphiques des niveaux logique, électrique, implanté et symbolique (STICK) ;

—des simulateurs fonctionnel, logique (par exemple FIDEL [HAZ88]), électrique (par exemple ELDO [Hen85]), logico-électrique (exemple FIDELDO) ;

—des vérificateurs automatiques de règles de dessins(D.R.C), des extracteurs

Malgré l'aide importante procurée par ces outils que l'on pourrait nommer "de la première génération", la conception des CI à la main (" Full Custom "en anglais) est restée onéreuse, ce qui la cantonne :

—à des applications stratégiques : défense, télécommunications...

—à la production de masse qui permet d'amortir les coûts de conception sur les quantités importantes de circuits vendus (économie d'échelle). Dans ce cas on même intérêt à optimiser la surface occupée par les circuits pour augmenter le nombre de circuits par unité de surface.

Les circuits ainsi produits sont caractérisés par :

—la généralité des fonctions qu'ils réalisent (micro-processeurs ROM, RAM par exemple) ; ce qui les rend applicables à un grand nombre de domaines moyennant un travail d'adaptation (circuits imprimés, microprogrammation...) plus ou moins important,

—par leur qualité : surface,vitesse et consommation optimales puisqu'ils sont conçus par des spécialistes.

Les gains de productivité obtenus par les fabricants de circuits intégrés ont très tôt permis d'envisager la production de circuits répondant aux besoins spécifiques des utilisateurs (ASIC pour Application Specific Integrated Circuit). On prévoit [DeM85][HOR86] que d'ici les années 90, la moitié des circuits intégrés produits seront spécialement conçus pour les applications des utilisateurs ou orientés vers ces applications. Face à cette tendance et aux coûts de conception engendrés par les méthodes classiques, de nouvelles méthodologies se développent et s'affinent continuellement. Elles visent à fournir aux ingénieurs système des moyens leur permettant d'obtenir directement le jeu de masques à partir de spécifications structurelles ou fonctionnelles. Pour le développement de ces méthodes, on fonde beaucoup d'espoirs sur l'usage des techniques d'intelligence artificielle. Pour l'heure, l'automatisation de la synthèse notamment physique a connu des progrès importants par le développement des circuits prédiffusés, précaractérisés et des générateurs de blocs fonctionnels ; de même, le passage automatique de la description comportementale à la description structurelle est en phase de maturation. Dans les chapitres suivants, nous exposerons notre contribution à la synthèse automatique de l'architecture des circuits intégrés à partir de spécifications fonctionnelles. Avant de présenter les grandes lignes de notre exposé, la suite du chapitre donne un aperçu des méthodes semi-automatiques et automatiques de conception des ASIC actuellement utilisées.

1.2. Outils de c.a.o pour les circuits prédiffuses

Les circuits prédiffusés sont des tableaux de transistors ("gate array" en anglais) non connectés qui ont subi toutes les étapes de la fabrication à l'exception des phases de métallisation et de contacts. Ces phases sont spécifiées par l'ingénieur système et correspondent à l'ensemble des connexions qu'il faut insérer entre les transistors pour obtenir le circuit voulu (fig 3).

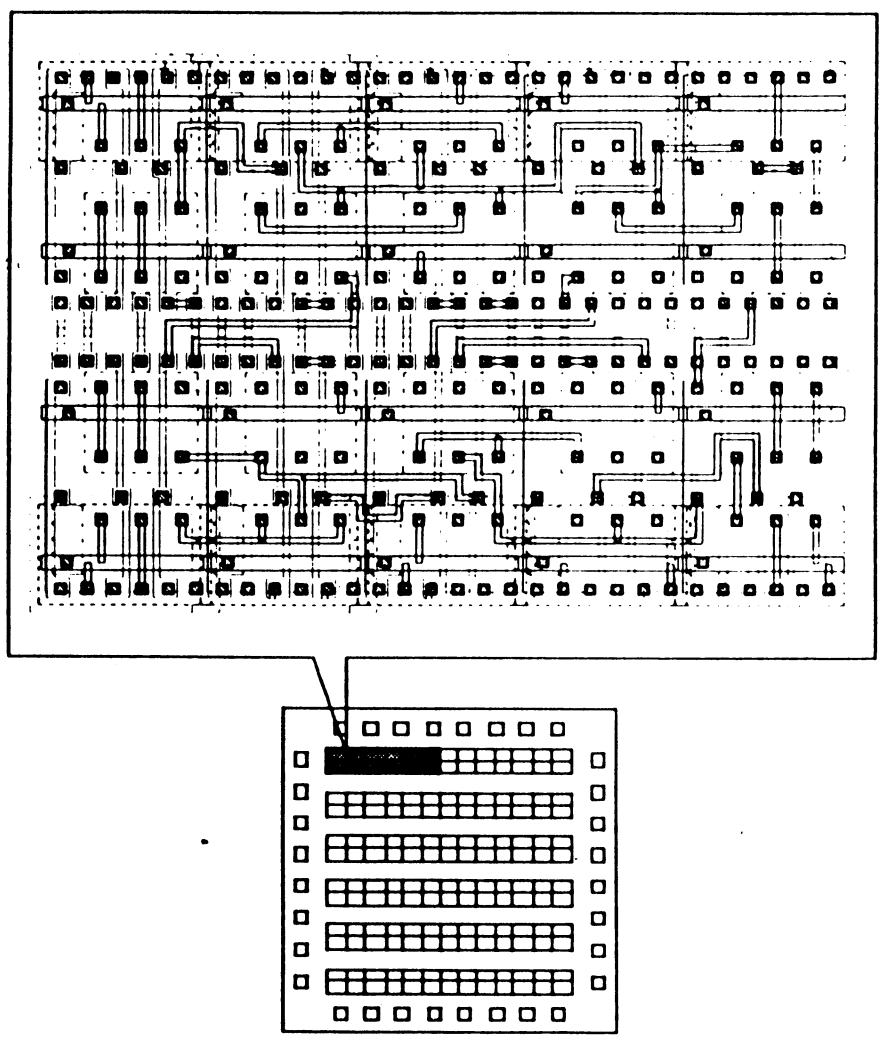


Figure 3 : routage d'une bascule D en circuit prédiffusé (tiré de [Wec 86])

La synthèse physique est alors accomplie par des moyens automatiques et se résume à un routage des connexions du circuit prédiffusé.

Les avantages des circuits prédiffusés sont multiples [Wec86] :

—les délais de conception et de fabrication d'un circuit sont considérablement réduits ainsi que les coûts de conception ; ce qui explique en partie le fait que ces méthodes soient très largement utilisées pour un développement rapide de prototypes ou pour la production des ASIC à faible échelle [Wec87] (circuits de complexité pouvant atteindre 10000 portes logiques pour des fréquences allant jusqu'à 15 Mhz [ROU 88] et une production annuelle inférieure à 100 [Kes85]) ;

—la méthode se prête fort bien à une automatisation permettant des conceptions plus sûres (correctes par construction) ; de fait, plusieurs logiciels sont disponibles sur le marché (vendus par les sociétés VENUS,MENTOR,SILVAR-LISCO,DAISY...)[DeM85] ;

—une compétence système suffit pour mener à bien la tâche de conception des circuits.

Ces avantages sont néanmoins sanctionnés par des circuits de moins bonne qualité comparés aux circuits faits à la main [DeM85][Wec86] :

—faible utilisation de la surface de silicium : des expériences indiquent des différences de densité d'intégration variant d'un facteur 2 à 3 par rapport aux circuits faits à la main [DeM85] ;

—circuits lents et consommant beaucoup d'énergie ;

—niveau de conception trop bas (transistors), ce qui allonge le temps de conception pour des circuits complexes ; cette limitation est partiellement résolue par l'introduction de "macrofonctions" dans les logiciels de routage de circuits prédiffusés ; ces macrofonctions permettent d'étendre le langage d'entrée à des portes logiques simples et même à des unités fonctionnelles évoluées (bascules par exemple) ; ces portes et unités fonctionnelles sont construites automatiquement suivant un plan de routage défini et programmé dans les macrofonctions correspondantes ;

1.3. Outils de cao pour les circuits précaracterisés

Les circuits précaracterisés sont conçus par assemblage de cellules standard ("standard cells"). Ces cellules, dessinées et testées par des concepteurs experts sont caractérisées par :

—leur fonction : logique aléatoire (inverseurs, NAND, NOR, AND, OR...), mémorisation (bascules, registres, "latch"...), ports d'entrée–sortie...

—leurs caractéristiques électriques (modèles logique et électrique de simulation) ;
—leur forme géométrique et leurs points d'entrée-sortie.

Ils sont conçus en respectant des contraintes de forme et de structure sévères afin de faciliter la génération automatique des masques et le tracé des interconnexions : les cellules ont toutes la même hauteur et les alimentations sont réalisées par aboutement (figure 4). La conception d'un circuit utilisant les cellules standard revient à spécifier les cellules à utiliser ainsi que les interconnexions. Des outils automatiques se chargent de placer et de router le circuit global.

L'avantage des circuits précaractérisés par rapport aux circuits prédiffusés peut être résumé par deux aspects fondamentaux :

—les cellules standard sont tirées d'une bibliothèque de cellules qui peut être étendue à volonté alors que les circuits prédiffusés n'utilisent qu'un nombre limité de cellules de base ; les cellules standard permettent ainsi d'élever le niveau de conception du circuit (on n'est pas limité à l'utilisation de portes logiques élémentaires) ; de fait, avec les cellules standard, on travaille pratiquement au niveau logique.
—les circuits obtenus à l'aide des cellules standard sont plus compacts puisque, contrairement aux circuits prédiffusés, les cellules n'ont pas une position prédéfinie ; ce qui permet aux programmes de génération d'optimiser leur placement.

En contre partie, la production d'un circuit à l'aide de cellules standard est :

—plus longue en temps : pour les circuits prédiffusés, seuls quelques niveaux de masque doivent être ajoutés alors que les cellules standard requièrent un cycle complet de fabrication ;
—plus coûteuse : l'investissement dans une bibliothèque de cellules standard est assez élevé [Wec86]. Mais ce coût est amorti au fil des conceptions successives puisque les cellules de la bibliothèque sont réutilisées au cours de ces conceptions.

Malgré les améliorations obtenues par rapport aux circuits prédiffusés, les circuits obtenus par placement et routage automatique de cellules standard présentent une utilisation non optimale de la surface : on estime à 50-80% environ, la surface occupée par les interconnexions par rapport à la surface totale du circuit [Kess85].

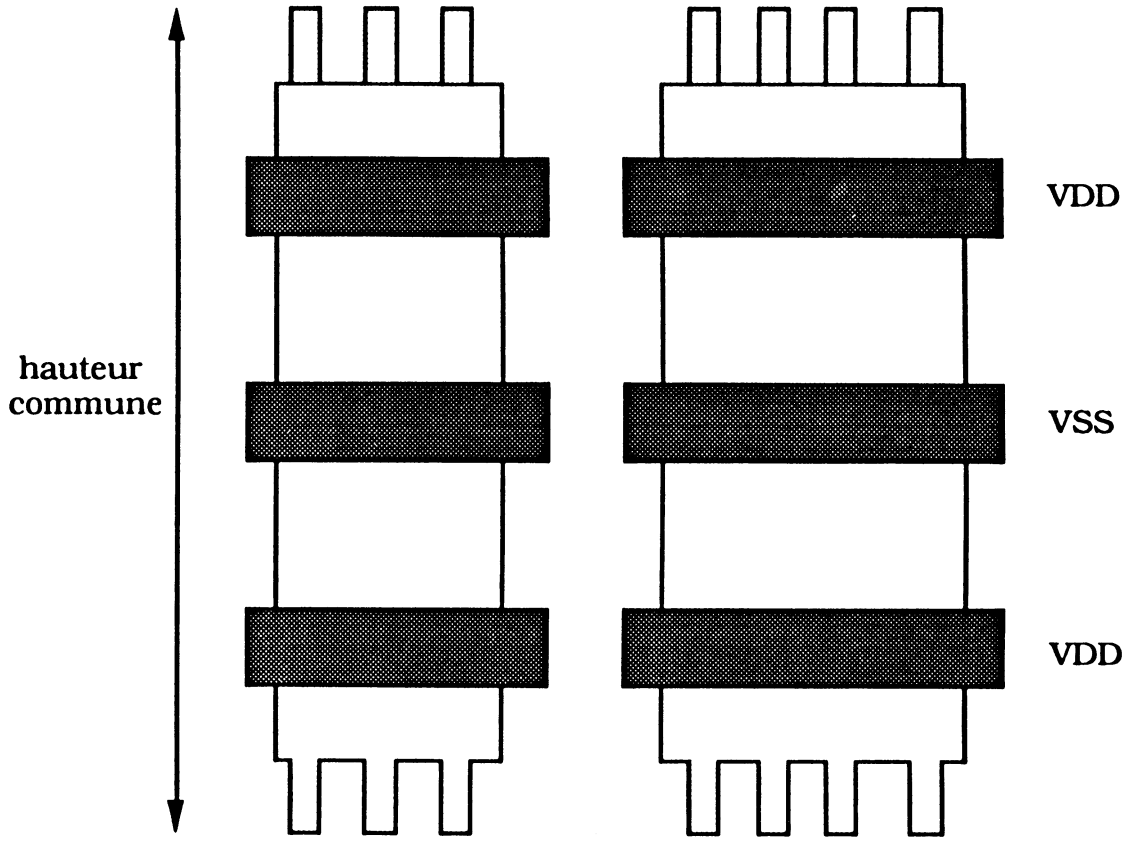


Figure 4 Structure des cellules standards

Les cellules standard se prêtent bien à la conception des ASIC de complexité comprise entre 1000 et 6000 portes [Kess85].

1.4. Générateurs automatiques de blocs fonctionnels

L'étape suivante dans l'automatisation de la synthèse de circuits intégrés est franchie par les générateurs automatiques de blocs fonctionnels. Au lieu de spécifier son circuit à l'aide de portes logiques (circuits prédiffusés et précaractérisés), l'ingénieur système le décompose en modules fonctionnels interconnectés. Les modules fonctionnels sont des macrocellules de complexité comparable aux circuits intégrés à moyenne ou grande intégration (MSI, LSI) tels les chemins de données, les compteurs, les registres, PLA, RAM, ROM, et les microcontrôleurs. [DeM85]. Une fois le circuit spécifié en termes de blocs fonctionnels interconnectés, le reste de la conception est assurée automatiquement par programme et procède en deux étapes :

—génération des macrocellules à l'aide de générateurs automatiques de blocs fonctionnels ;

—placement et routage automatique des macrocellules générées.

1.4.1. Les macrocellules

Cette méthode de conception repose sur le concept de macro-cellule. Une macro-cellule est obtenue par assemblage de cellules primitives ou d'autres macro-cellules. Ainsi donc, on peut voir les macro-cellules comme une hiérarchie de cellules. Au niveau le plus élevé, la macro-cellule est définie comme un ensemble de cellules interconnectées. Dans cette description, les cellules composantes sont référencées par leurs caractéristiques fonctionnelles. Chacune de ces cellules est à son tour définie à partir de cellules moins complexes. On itère le processus jusqu'aux cellules primitives constituées de transistors.

1.4.2. Les générateurs de modules fonctionnels

Les générateurs de modules fonctionnels sont des procédures dont l'exécution construit des macro-cellules. Les paramètres définissent les caractéristiques fonctionnelles, électriques et structurelles des macro-cellules à générer. Par exemple un générateur de registres, peut recevoir en entrée le type de registre (registre simple latch, maître-esclave, bascule D...) et la taille du registre en nombre de bits. Un générateur de PLA recevra en entrée un ensemble de fonctions booléennes qu'il optimisera pour produire en sortie les masques du PLA résultant.

Les générateurs sont caractérisés par :

—une liste de composants : cellules primitives ou macro-cellules utilisées pour produire le module fonctionnel ;

—un algorithme qui assemble les composants suivant les contraintes exprimées par les paramètres ; cet algorithme est souvent exprimé dans un langage de programmation évolué (compilé ou interprété) bien adapté à une description hiérarchique de l'assemblage de cellules. Le système LOF [Oli 86] utilise le langage Ada pour exprimer l'algorithme d'assemblage des générateurs.

1.4.3. Obtention des cellules primitives

Les cellules primitives intervenant dans la définition d'une macrocellule peuvent être obtenues de deux façons :

—par sélection dans une bibliothèque de cellules pré-dessinées et testées, auquel cas on parle de générateurs à base de cellules ("cell based generators" [DeM85]) ; en général ces cellules sont étudiées pour permettre un assemblage par aboutement plutôt que par routage ; on obtient ainsi des densités d'intégration bien meilleures qu'avec les circuits précaractérisés [DeM85] ; l'inconvénient majeur dans ce cas est la dépendance vis-à-vis de la technologie : puisque les cellules sont dessinées dans une technologie donnée, il faut les redessiner si l'on change de technologie. Une solution serait de définir les cellules sous une forme symbolique comme un ensemble de transistors interconnectés et d'utiliser un programme compacteur pour obtenir des masques vérifiant les contraintes de la technologie ;

—par des programmes ; auquel cas on parle de générateurs procéduraux. Cette méthode n'est efficace que pour des cellules de complexité faible en nombre de transistors ou présentant une structure régulière (PLA, WEINBERGER, SLA, "gate matrix"...). Des recherches appliquant les techniques d'intelligence artificielle à la conception de cellules aux niveaux implanté et symbolique commencent à porter des fruits (cf [Kim 83]).

1.4.4. Conclusion

La construction des générateurs est une tâche complexe et coûteuse réalisée par une équipe pluridisciplinaire comprenant des spécialistes en conception de circuits intégrés et des informaticiens. Mais les avantages sont multiples :

—la conception d'un circuit intégré à l'aide de générateurs fonctionnels s'arrête au niveau structurel (on dit aussi RTL pour "Register Transfert Level") : la spécification comportementale du circuit à réaliser est transformée en un ensemble de modules fonctionels interconnectés. Les générateurs automatiques construisent progressivement les modules fonctionnels à partir des cellules primitives suivant une stratégie montante ("bottom-up"), et l'ensemble est routé. Cette méthode de conception est appelée "meet-in-the-middle"[Wec86] pour désigner à la fois la conception descendante("top-down") faite par le concepteur, et la construction "montante" ("bottom up") effectuée par les générateurs automatiques de modules fonctionnels.

—les circuits construits à l'aide de générateurs sont meilleurs en vitesse, consommation et surface que les circuits précaractérisés puisque les macrocellules sont conçues pour être construites par aboutement de cellules plutôt que par routage.

1.5. Vers la Compilation de silicium

Les approches les plus ambitieuses dans l'automatisation de la conception des circuits intégrés sont fournies par les compilateurs de silicium. Ceux-ci visent à produire directement le dessin implanté d'un circuit à partir d'une spécification donnée de même que les compilateurs de langages de programmation qui produisent le code machine correspondant à un programme donné. L'utilisateur n'a plus à s'occuper des aspects physiques du circuit en cours de construction et peut donc se consacrer davantage au problème de l'adéquation de sa description fonctionnelle aux spécifications du cahier des charges. On peut classer les compilateurs de silicium selon deux critères :

—le langage d'entrée ;

—les architectures des circuits générés.

1.5.1. Le langage d'entrée

Il s'étend du niveau structurel au niveau fonctionnel avec des variantes.

Au niveau structurel, le circuit est décrit par un formalisme reflètant la structure architecturale du circuit à réaliser. Le circuit est donc décrit en entrée au niveau RTL. le compilateur de silicium correspondant n'est bien souvent qu'un assembleur de silicium qui se contente du placement et du routage des composants du circuit selon la structure décrite en entrée avec peu ou pas d'optimisation sur le circuit généré. Les composants sont en général produits par des générateurs de modules fonctionnels selon des paramètres spécifiés dans la description fournie en entrée du compilateur de silicium. Comme exemples de tels assembleurs, on peut citer CONCORDE et GENESIL commercialisés respectivement par les sociétés SS (Seatle Silicon) et SCS (Silicon Compiler Systems).

Au niveau fonctionnel, le circuit est décrit en entrée par un algorithme, un ensemble d'équations ou tout autre formalisme (graphique notamment) décrivant le comportement du circuit à dessiner. Souvent, la description fonctionnelle est accompagnée d'un ensemble de contraintes que doit satisfaire le circuit généré :

par exemple une vitesse minimale de fonctionnement, une surface maximale du circuit limitée, un jeu restraint de ressources fonctionnelles à utiliser... Ici, la tâche du compilateur est plus difficile puisqu'il doit décomposer la description fonctionnelle initiale en un ensemble de modules interconnectés tout en respectant les contraintes exprimées. Le langage d'entrée est soit :

—dérivé d'un langage de programmation classique étendu à quelques primitives permettant d'exprimer les contraintes structurelles ; par exemple MAC-PITS utilise LISP en entrée ;

—un formalisme permettant de décrire un automate d'état fini ; GASP est un générateur de microcontroleurs [Fla84] qui reçoit en entrée un automate décrit par un langage dérivé des réseaux de Pétri ;

—un symbolisme grahique ; IMHOTEP reçoit en entrée un filtre décrit à l'aide d'un ensemble de symboles graphiques représentant les opérateurs élémentaires utilisés dans le filtrage(voir chapitre 2) ;

—un formalisme dérivé des langages RTL permettant de spécifier l'algorithme d'interprétation d'un micro-processeur à réaliser. Dans cette catégorie, on peut citer SYCO qui part d'une description inspirée de deux langages HDL ("Hardware Description Langage") IRENE et LDS [Jer-86].

1.5.2. Les architectures générées

On peut distinguer deux cas selon que les architectures produites ont une même structure prédéfinie ou pas.

Dans le premier cas, le circuit est généré selon une architecture cible prédéfinie. Certains choix architecturaux (nombre de bus, UAL, ...) sont décidés une fois pour toutes dans l'architecture cible, le reste étant déterminé par le compilateur suivant les besoins du circuit à générer. L'architecture du circuit est souvent produite par un générateur qui reçoit en entrée des paramètres déterminés par le compilateur (taille des RAMs, nombre de registres...). Les problèmes de placement et de routage des modules du circuit sont réglés une fois pour toutes lors de la définition de l'architecture cible et programmés dans le générateur. L'ampleur de la tâche de compilation est déterminée à la fois par le niveau de description du langage d'entrée et les flexibilités de l'architecture cible. Lorsque le langage d'entrée se situe au niveau structurel, le compilateur est plutôt un assembleur de silicium qui produit en sortie une architecture conforme à la description initiale. Il y a peu ou pas du tout d'optimisation de l'architecture générée. Lorsque le langage d'entrée se rapproche du niveau fonctionnel, la tâche de compilation est plus complexe ainsi :

—BRISTLE BLOCKS travaille au niveau structurel [Joh 83]. Les problèmes de placement et de routage sont résolus par l'architecture cible du circuit à produire. Cette architecture cible est un cœur de micro-processeur ;

—SYCO [Jer86] utilise une architecture cible inspirée du microprocesseur MC68000 composée d'une partie opérative à deux bus (fig 5). Les bus peuvent être segmentés pour définir plusieurs tranches de partie opérative opérant en parallèle. Chacune de ces tranches est constituée d'un ensemble de registres, ROM, RAM en simple ou double accès et des ALUs connectés à deux bus. La partie contrôle est définie par un PLA, une ROM ou suivant une structure de séquencement reposant sur une base de temps [OBR82]. SYCO reçoit en entrée l'algorithme d'interprétation des instructions d'un microprocesseur qu'il affine suivant une stratégie descendante ("top down") en une hiérarchie de niveaux d'interprétations successifs plus proches de la structure définie par l'architecture cible (figure 5). Dans la hiérarchie, chaque niveau d'interprétation fournit des primitives plus simples qui serviront à réaliser l'interpréteur du niveau directement supérieur.

L'inconvénient majeur de cette catégorie de compilateurs est d'être limité à un domaine bien précis : par exemple SYCO ne s'intéresse qu'aux micro processeurs, CATHEDRAL ne s'intéresse qu'aux circuits intégrés de traitement du signal. Néanmoins ces compilateurs permettent une bonne utilisation de la surface de silicium puisque les problèmes de placement et de routage sont réglés lors de la définition de l'architecture cible.

La deuxième catégorie de compilateurs élargit le champ d'investigation à un ensemble d'architectures obtenues par interconnexion de ressources fonctionnelles disponibles en bibliothèque ou produits par des générateurs. Ici, les problèmes de placement et de routage ne peuvent pas toujours être résolus à l'avance. D'autre part l'espace de recherche est beacoup plus grand en raison des nombreux choix architecturaux possibles (pipe-line, parallèle, série...). Comme précédemment, le niveau du langage d'entrée a une incidence sur la complexité de la compilation :

—lorsque le langage d'entrée se situe au niveau structurel, le compilateur est assimilable à un programme de placement et de routage de modules fonctionnels. La complexité du problème de placement impose soit des contraintes de forme sur les modules fonctionnels (structures "bit-slice",dimensions, position des points d'interconnexion...), soit l'assistance de l'utilisateur pour guider le compilateur dans la recherche d'un

placement optimal. Comme exemples de tels compilateurs on peut citer GENESIL, MENTOR et SPOT ([Rob 89] voir chapitre 2).

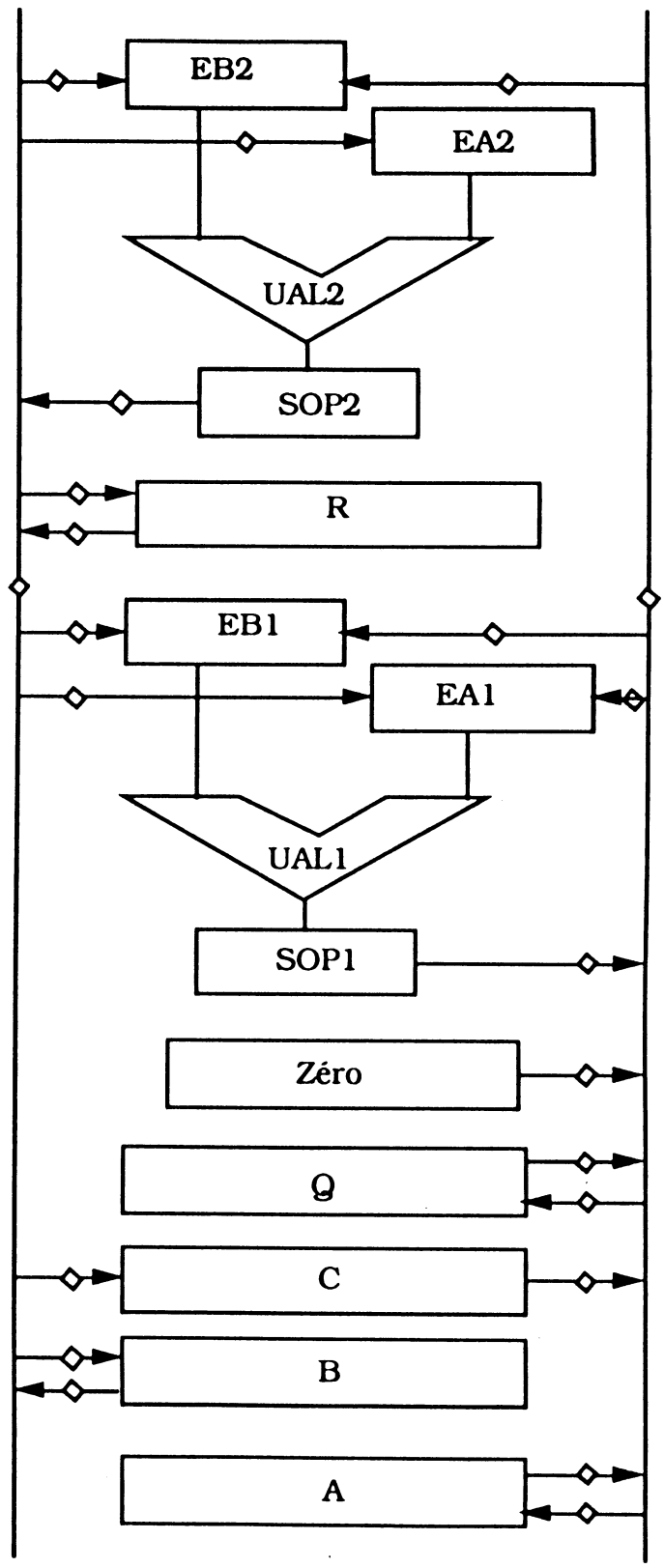


Figure 5 : Architecture cible de SYCO [Jer 86]

—lorsque le langage d'entrée se situe au niveau fonctionnel, le circuit à réaliser est décrit par un algorithme ou par un formalisme graphique définissant la relation entre les entrées et les sorties du circuit. Cette relation est en général exprimée sous la forme d'un enchaînement d'opérations à effectuer sur les entrées du circuit pour obtenir les sorties. La compilation consiste à élaborer une architecture optimale selon des critères (vitesse, surface, consommation...) définis par l'utilisateur. La difficulté de la synthèse structurelle est accrue par les nombreux choix que le compilateur doit effectuer pour définir l'architecture du circuit : ordonnancement des opérations, choix des opérateurs fonctionnels, allocation des ressources, réalisation des interconnexions. Comme on le voit, le point critique est la génération automatique d'une architecture optimale à partir d'une description fonctionnelle en respectant les contraintes de l'utilisateurs. Bien qu'il n'existe à l'heure actuelle aucun système commercialisé et fiable capable de produire une architecture pour des applications et des contraintes réelles, un certain nombre de travaux prometteurs ont été publiés sur le sujet ([Gae-88]). Avant de discuter des performances de certains de ces systèmes parmi les plus prometteurs, nous décrivons dans les chapitres suivants, celui que nous avons conçu et réalisé en partie dans le cadre de cette thèse.

1.6. Plan de la thèse

Au chapitre 2, nous présentons avec plus de détail le problème de la synthèse architecturale, le cahier des charges de l'outil que nous avons développé ainsi que les raisons qui nous ont poussé à adopter une approche par l'intelligence artificielle.

Au chapitre 3 nous décrivons et analysons le moteur d'inférence ODSE que nous avons conçu pour résoudre le problème précédemment évoqué. Nous indiquons les choix que nous avons faits ainsi que les raisons qui nous ont amené à écrire un moteur d'inférence (un de plus !) en ADA.

Le système expert d'Aide à la Synthèse Architecturale (ASA) de circuits intégrés est décrit en détail au chapitre 4

Le chapitre 5 compare les performances de ASA à celles de quelques systèmes existants.

En conclusion, le chapitre 6 montre les extentions possibles et les perspectives à court et long terme du compilateur de silicium et du moteur d'inférence que nous avons développés.

2. Synthèse automatique des architectures de circuits intégrés

Après avoir posé le problème de la synthèse automatique, ce chapitre définit les spécifications externes de ASA, outil d'Aide à la Synthèse Architecturale de circuits intégrés que nous avons développé. Ensuite, une analyse de la complexité des problèmes inhérents à la synthèse automatique permet de justifier le recours à des techniques de l'intelligence artificielle pour la réalisation de ASA.

2.1. Position du problème

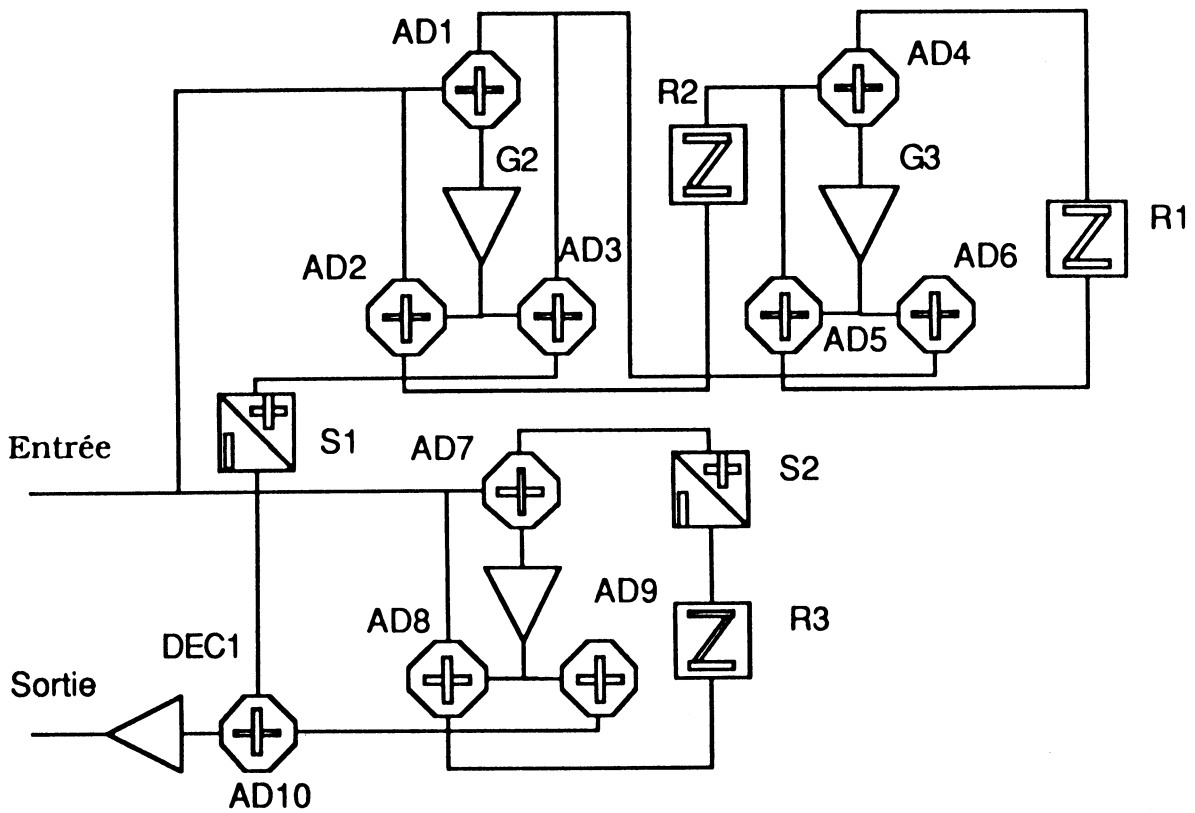
2.1.1. Approche informelle

La synthèse automatique des architectures de circuits intégrés VLSI consiste, à partir d'une description comportementale du circuit et d'un jeu de contraintes, à proposer par voie automatique (programme), la description structurelle d'un circuit qui fonctionne suivant le comportement donné et respecte les contraintes indiquées.

La description comportementale définit les interactions du circuit avec son environnement sous la forme d'une relation entre les entrées et les sorties du circuit. Cette relation exprime les transformations que le circuit effectue sur les séquences de signaux reçus en entrée pour produire les signaux de sortie. Comme nous l'avons vu au chapitre précédent, cette description peut se situer à divers niveaux d'abstraction mais nous ne nous interessons ici qu'au niveau fonctionnel : le circuit est décrit sous une forme algorithmique sans préjuger de sa structure. Par exemple IMHOTEP utilise un formalisme graphique pour décrire le comportement des filtres qu'il doit réaliser (fig 1). D'autres systèmes utilisent des langages algorithmiques (comme Pascal) [Tri 87] ou des langages de description fonctionnelle de circuits du type ISPS [Kow85] [Jer 86].

L'expression des contraintes est aussi très variable selon les systèmes : limitations imposées à la consommation d'énergie ou à la surface occupée par le circuit, vitesse minimale de fonctionnement du circuit... Elles prennent des formes diverses. Ainsi, dans IMHOTEP, l'utilisateur peut limiter le type et le nombre d'opérateurs (additionneurs, multiplieurs...) à utiliser pour la synthèse et imposer une fréquence minimale de fonctionnement du filtre. Les contraintes ont une incidence directe sur l'architecture du circuit final. En modifiant le jeu

de contraintes initial, le concepteur peut explorer plusieurs alternatives pour un même circuit.



Liste des paramètres :

Taille des données : 16 bits
G1 : Cœfficient valant 0,1
G2 : cœfficient valant 0,001
G3 : cœfficient valant 0,0001

Contrainte de temps du circuit : 360 nano secondes

Figure 1 : exemple de description de filtre pour IMHOTEP

La description structurelle est donnée sous la forme d'une liste de modules fonctionnels interconnectés. Cette description est élaborée à partir de la description comportementale et du jeu de contraintes initial. Pour une spécification comportementale donnée, il existe en général plusieurs architectures de circuit possible. C'est pourquoi les contraintes sont utilisées pour réduire l'espace de recherche. La plupart des systèmes limitent l'espace de recherche en imposant un jeu de modules fonctionnels à utiliser pour la synthèse architecturale. Cette limitation peut aller jusqu'à imposer les grandes

lignes de la structure du circuit à réaliser ; c'est le cas de SYCO, MACPITTS et bien d'autres. Les contraintes servent aussi à guider le choix d'une architecture satisfaisante. Par exemple dans IMHOTEP, l'activité de synthèse se ramène à un problème d'ordonnancement de tâches sous contraintes de temps et de ressources [Rey 85]. Il s'agit pour IMHOTEP de construire un circuit de surface minimale et vérifiant les contraintes. La figure 2 présente un exemple d'architecture généré par IMHOTEP pour le filtre de la figure 1.

A l'heure actuelle, les compilateurs de silicium disponibles sur le marché travaillent au niveau structurel ou sur des architectures spécialisées. Un certain nombre de prototypes existent qui essaient d'automatiser la production des circuits spécifiques ASIC en offrant aux utilisateurs une interface de haut niveau.

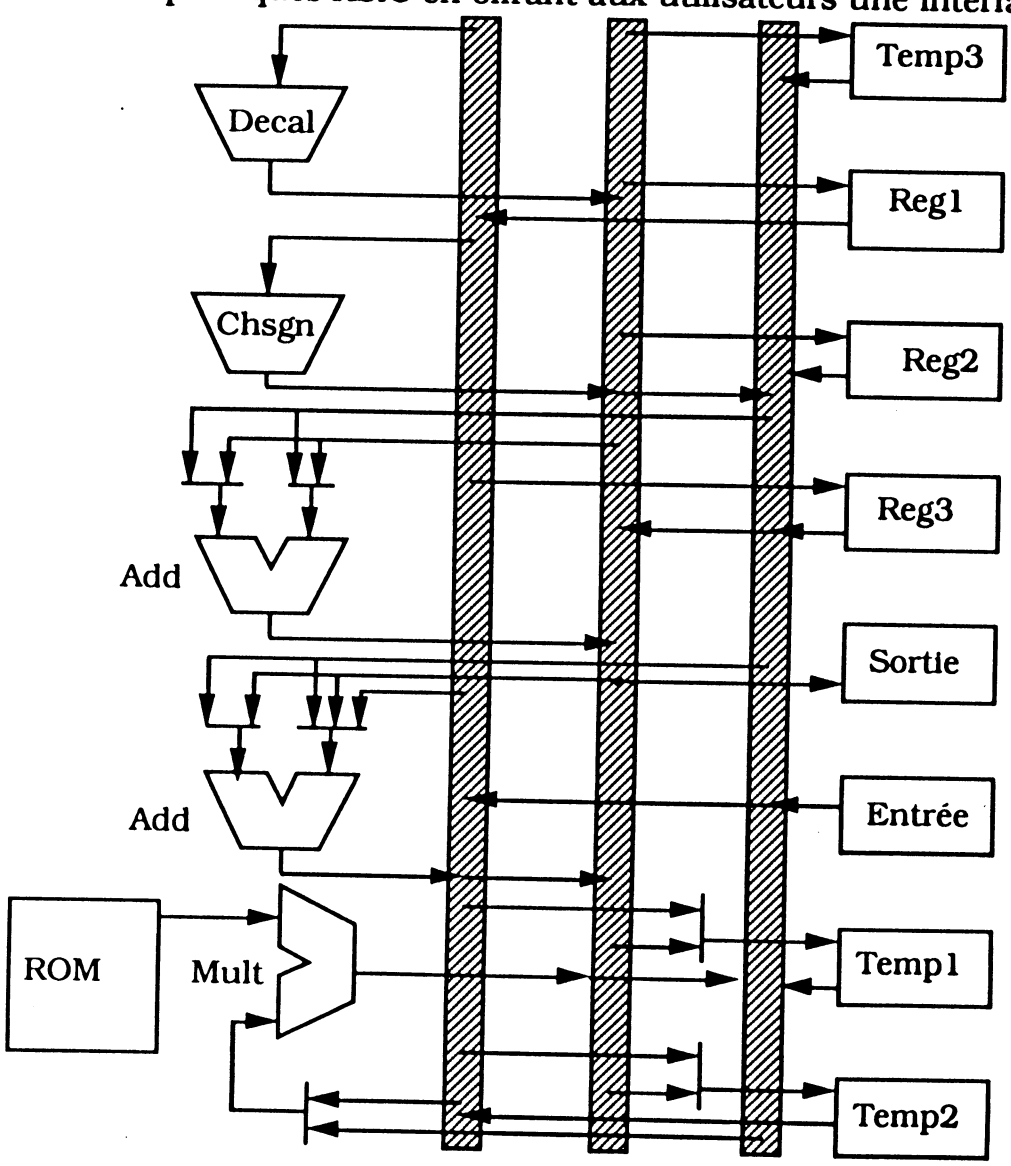


Figure 2 : Architecture produite par IMHOTEP

2.1.2. Spécification de ASA

Comme la pluspart des prototypes existant dans le domaine, ASA a été conçu pour :

—réduire les délais de conception des ASIC ;

—réduire les erreurs de conception en utilisant des méthodes automatiques sûres produisant des circuits corrects par construction ;

—aider les concepteurs à faire des compromis coût/performances en leur permettant d'explorer et d'évaluer rapidement plusieurs choix architecturaux possibles ;

—rendre la technique de conception des circuits intégrés accessible à un plus grand nombre de personnes et notamment aux ingénieurs systèmes.

Le langage d'entrée

ASA accepte en entrée la description fonctionnelle du circuit exprimée dans un langage dont la syntaxe est dérivée de ADA [Ber 89].

Les seuls types de donnée pris en compte sont : les entiers, les booléens et les tableaux. Pour les entiers, on peut préciser le domaine des valeurs visé ce qui permet au système de déduire une taille optimale en nombre de bits pour les chemins de données du circuit.

La description d'un circuit suit à peu près la syntaxe de définition des procédures paramétrées en ADA. il comporte un en-tête, une partie déclaration des variables et un corps.

L'en-tête définit le nom du circuit ainsi qu'une liste de paramètres servant à spécifier l'interface du circuit avec son environnement extérieur. Les paramètres passés en mode valeur (*in* en Ada) décrivent la communication du monde extérieur vers le circuit alors que le mode résultat (*out* en Ada) décrit la communication dans le sens inverse. Les paramètres sont interprétés par le compilateur comme des contraintes structurelles correspondant à des ports d'entrée sortie sur le circuit final.

La partie déclaration permet de définir le type des variables manipulées par le corps de la description. Ces variables servent exclusivement à définir le comportement du circuit et ne correspondent pas nécessairement à un élément de mémorisation dans le circuit final. En effet plusieurs optimisations sont effectuées par le compilateur pour réduire le nombre d'éléments de mémorisation (voir chapitre 4).

Le corps de la description exprime l'algorithme du comportement du circuit sous la forme d'un enchaînement d'opérations et d'actions sur les

variables et les paramètres précédemment définis. Les structures de contrôle utilisées sont : la structure séquentielle, la structure conditionnelle et la structure itérative. Ces structures de contrôle ont pour seul but d'exprimer les contraintes de dépendance entre les données manipulées et le flot de contrôle de l'algorithme. Elles ne servent donc pas à limiter à priori le parallélisme ou à imposer a priori un ordre d'exécution des opérations dans le circuit final. En fait, l'algorithme est transformé sous une forme interne qui ne prend en compte que les contraintes de dépendance entre les données et le flot de contrôle caractérisant la sémantique de l'algorithme. Comme nous le verrons au chapitre 4, cette forme se prête bien à toute une série d'optimisations. Cependant une exception est à noter pour les opérations d'entrée-sortie sur les paramètres qui sont exécutées dans l'ordre indiqué dans la description.

```
Circuit equadifff is
-- cet exemple est tiré de [Pau 85]
-- il essaie de résoudre l'équation
-- différentielle y"+5xy'+3y=0
u,x,dx,y,a,y1,u1,u2,
u3,u4,u5,u6: integer range -127..128;
begin
-- a contient l'abcisse pour laquelle
--on veut connaitre la valeur de la
--fonction, les conditions aux limites
--sont définies par
--les valeurs initiales de x,u,y
-- dx contient le pas de discrétisation
while x<a loop
u1:=u*dx;
u2:=5*x;
u3:=3*y;
y1:=u*dx;
x:=x+dx;
u4:=u1*u2;
u5:=dx*u3;
y:=y+y1;
u6:=u-u4;
u:=u6-u5;
end loop;
end equadiff;
```

Figure 3 description fonctionnelle d'un circuit pour ASA

La figure 3 présente un exemple de circuit à réaliser. On trouvera en annexe, la syntaxe complète du langage utilisé. Comme en ADA, les commentaires sont introduits par deux tirets et ne sont pas pris en compte par le compilateur.

En fait, le langage recouvre la classe des algorithmes dits à flots réductibles [Aho 86], bien adaptés à certaines formes d'optimisation (cf chapitre 4). Les restrictions apportées au langage de description du circuit ne nuisent pas tellement à la généralité de ASA puisque selon Aho [Aho 86], la grande majorité des logiciels développés suivant les méthodes de programmation structurée ont un flot réductible.

Les contraintes

Le but étant de développer un outil intelligent d'aide à la synthèse de circuits, la flexibilité du système doit permettre de prendre en compte une gamme variée de contraintes. Dans ASA, ces contraintes sont spécifiées de manière interactive et peuvent aller des contraintes structurelles où l'on définit pratiquement l'architecture cible, aux contraintes de performances en termes de vitesse, surface et consommation du circuit. Les contraintes pris en compte par ASA sont :

—les contraintes relatives aux types et nombre de ressources fonctionnelles utilisées : on peut limiter par exemple le nombre d'additionneurs à utiliser, le nombre de bus... Ce type de contraintes permet éventuellement d'imposer une architecture cible ;

—les contraintes de vitesse :

  —on peut imposer la durée du cycle de base du circuit. Dans ce cas, les unités fonctionnelles qui ont un temps de réponse plus grand que la période d'horloge sont ordonnancés sur plusieurs cycles ;

  —le système permet aussi d'exprimer des contraintes de temps d'exécution. Ces contraintes peuvent s'appliquer à l'ensemble du circuit ou à une partie de la description. Si l'on veut realiser un filtre fonctionnant à une fréquence donnée, le filtre peut être décrit par un algorithme iteratif dont le corps de boucle doit s'exécuter en un temps imposé par la fréquence de fonctionnement du filtre ;

—les contraintes d'optimisation fournissent un critère de qualité qui guide le choix d'une architecture appropriée à une description donnée. L'espace du choix est limité par la bibliothèque d'opérateurs qui sert de support à ASA (cf § ci-dessous). Le critère de qualité se présente sous deux formes : dans le premier cas, il s'agit de réaliser le circuit le plus rapide occupant une surface aussi réduite que possible ; dans le deuxième cas, il s'agit de réaliser un circuit de surface minimale dont la vitesse est la plus élevée possible. Dans l'un et l'autre cas la contrainte revient à ordonner les critères de vitesse et d'encombrement du circuit qui doivent présider aux choix architecturaux, notamment en cas de conflit. Dans le premier cas, le

critère de vitesse prend le pas sur le critère de surface alors que dans le second cas, c'est l'inverse.

Le contexte

La tâche de ASA est de proposer une architecture de circuit en fonction de la description et des contraintes qu'il reçoit en entrée. Les architectures proposées par ASA sont obtenues par assemblages de modules fonctionnels tirés de la bibliothèque d'opérateurs appelée FBL (Flexible Bloc Library). Cette bibliothèque délimite indirectement l'espace des architectures possibles que ASA peut synthétiser.

FBL est une bibliothèque d'opérateurs en cours de développement au CNET GRENOBLE dans le cadre du projet européen CVS. Les opérateurs disponibles couvrent une grande variété de fonctions : portes logiques, registres, additionneurs, UAL, RAM, ROM, chemins de données...Les opérateurs sont regroupés par familles suivant les fonctions qu'ils réalisent. Les opérateurs d'une même famille se distinguent par leurs formes et leurs performances. Chaque famille est associée à un générateur qui produit une instance en fonction des caractéristiques qui lui sont fournies en paramètres. Les générateurs sont programmés à l'aide du système LOF (langage des opérateurs flexibles) réalisé au CNET Grenoble [Oli 87].

SPOT (Synthèse de Partie Opérative en Tranches) [Rob89] est un macrogénérateur de cette bibliothèque qui produit des chemins de données en tranches ("bit-slice" en anglais) à partir d'une description structurelle.

En sortie, ASA produit un fichier au format d'entrée de SPOT décrivant la partie opérative du circuit.

Caractéristiques des architectures produites par ASA

Les architectures produites par ASA sont constituées d'une partie opérative composée d'un ensemble d'opérateurs fonctionnels interconnectés, et d'une partie contrôle. Ces deux parties sont contrôlées par une horloge commune. La période de l'horloge est demandée à l'utilisateur avant la synthèse.

Les architectures systoliques ou pipeline ne sont pas traitées par ASA.

2.2. Les grandes étapes de synthèse architecturale

Du point de vue de ASA, la synthèse doit être effectuée à l'aide de modules fonctionnels tirés d'un éventail de possibilités contenues dans une bibliothèque. Cette synthèse procède en plusieurs étapes (fig 4) :

—traduction de la description initiale sous une forme permettant d'effectuer une série d'optimisations pour éliminer les maladresses éventuelles de la description. Ces optimisations ressemblent à celles qui sont effectuées par les compilateurs de langage de programmation : élimination des branches mortes ("dead code" [Aho 87]), propagation des constantes, élimination des sous-expressions communes, déroulement des boucles...(cf chap 4) ;

—ordonnancement des opérations de l'algorithme suivant une chronologie respectant les contraintes exprimées. Le but de l'ordonnancement est de minimiser la durée ou le nombre de cycles requis pour exécuter l'algorithme dans la limite des ressources disponibles et des contraintes imposées ;

—allocation des ressources fonctionnelles constituant le chemin de donnée du circuit. Le problème est ici de minimiser la quantité de ressources fonctionnelles utilisée par le circuit. Ces ressources se divisent en unités fonctionnelles (additionneurs, UAL, multiplieurs...), éléments de mémorisation (registres, RAM, ROM...) et organes d'interconnexion (bus, multiplexeurs...). Pour minimiser les unités fonctionnelles, on cherchera à regrouper les opérations mutuellement exclusives (celles qui sont ordonnancées sur des tranches de temps disjoints) en une même unité fonctionnelle. Ces regroupements sont faits tant que la bibliothèque dispose d'unités fonctionnelles capables d'effectuer toutes les opérations d'un groupe donné. L'allocation des éléments de mémorisation et des organes de connexion peut être formulée de manière analogue. On se rend bien compte des interactions entre les problèmes d'ordonnancement et d'allocation. Un ordonnancement recherchant un parallélisme maximal exigera plus de ressources fonctionnelles qu'un ordonnancement qui sérialise les opérations ;

—synthèse de la partie contrôle du circuit. Une fois que les problèmes d'ordonnancement et d'allocation ont été résolus, la partie contrôle peut être décrite sous la forme d'une machine d'états finis. Cette machine d'états finis peut être réalisée de plusieurs façons :

—par codage des états et optimisation de la logique combinatoire, ce qui mène à une réalisation à l'aide de compteurs, registres, et PLA ;
—par utilisation d'une mémoire morte et de registres ;
—par décomposition de l'automate de contrôle en automates de taille réduite dans un souci d'accroître les performances de la partie contrôle et de réduire la surface totale occupée [Obr 82].

2.3. Particularités des problèmes de synthèse architecturale

La synthèse automatique d'une architecture à partir d'une description algorithmique et d'un jeu de contraintes procède par toute une série de choix et de décisions pour obtenir l'architecture désirée. La principale difficulté vient de la multitude des possibités à envisager pour trouver le choix idéal. D'autre part l'ordre dans lequel les sous-problèmes précédemment évoqués sont résolus a une incidence sur la qualité de l'architecture obtenue. Ainsi un ordonnancement déterminé impose un jeu minimal de ressources fonctionnelles de même qu'un jeu de ressources imposé limite les performances du circuit. Les choix à faire sont bien souvent conflictuels : une surface minimale se traduit par des circuits peu performants et inversement des circuits rapides exigent beaucoup de ressources fonctionnelles. Toutes ces difficultés font de la synthèse automatique des architectures de circuits intégrés un problème NP-complet. C'est-à-dire qu'il n'existe pas d'algorithme déterministe s'exécutant en un temps polynomial capable de résoudre ce problème [Sah80]. C'est pourquoi les recherches qui s'attaquent à ce problème doivent suivre une approche heuristique ou d'intelligence artificielle. Un certain nombre de considérations nous ont fait pencher pour la deuxième approche.

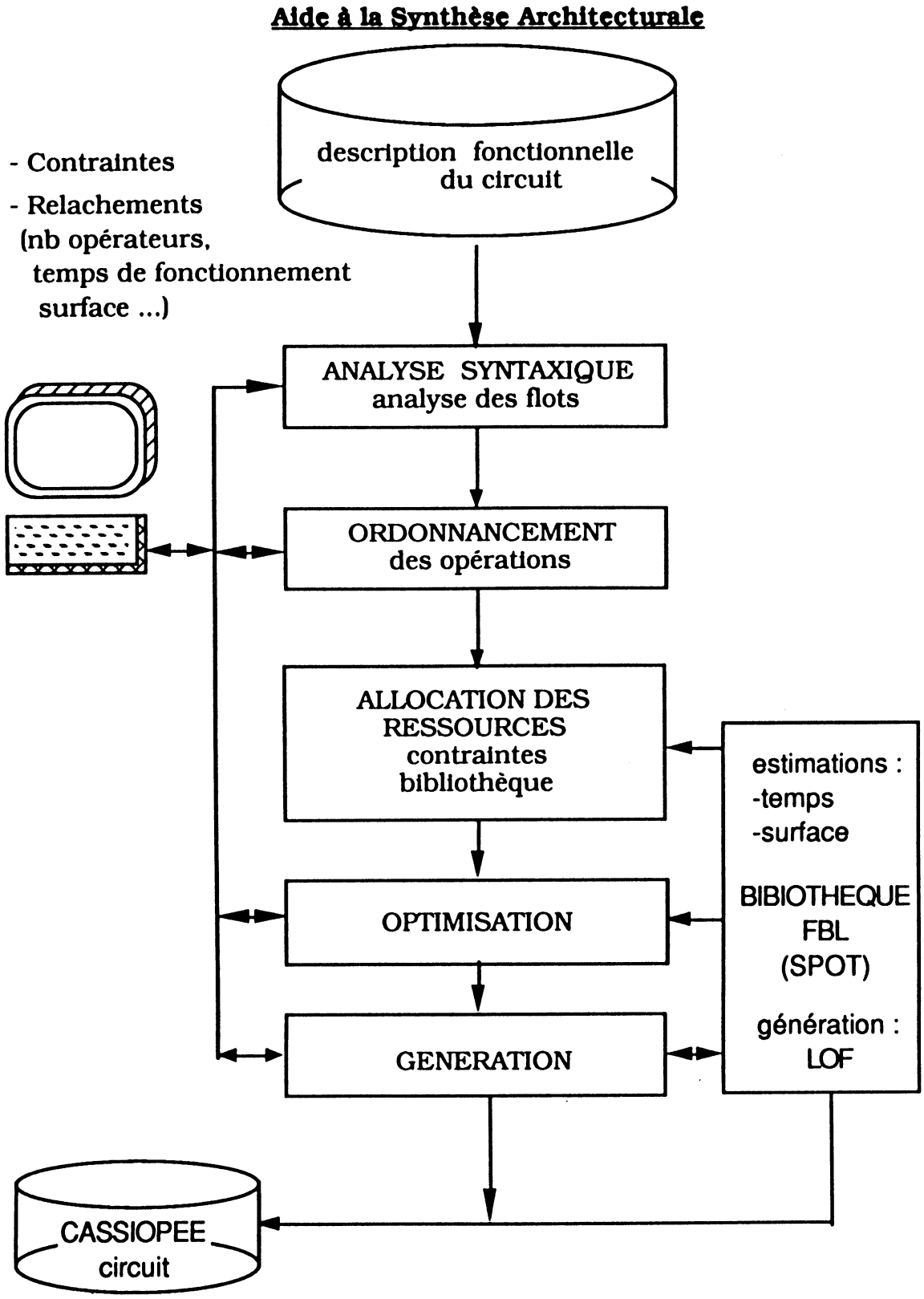


Figure 4 : étapes de la synthèse architecturale

Limites des approches heuristiques

Une heuristique est un critère, une stratégie ou une méthode qui associe à chaque alternative, d'un choix donné, une estimation de son efficacité à résoudre le problème initial ou de la qualité probable de la solution obtenue. La résolution de problèmes à l'aide de méthodes heuristiques se sert de ces estimations pour choisir les chemins à suivre. Les chemins dont les estimations heuristiques sont défavorables sont écartées de la recherche. Cela permet de développer des algorithmes de complexité raisonnable car ils s'appuient sur une forme dégénérée du problème initial à résoudre. En effét, les heuristiques sont en général calculées en considérant des modèles mathématiques simplifiés du problème à résoudre. De ce fait, les valeurs obtenues sont approximatives et même parfois erronnées [Pea84]. En conséquence on n'est plus sûr d'obtenir une solution optimale ou même une solution tout court. Considérons par exemple le cas de IMHOTEP. IMHOTEP part de la description d'un filtre et d'un ensemble de contraintes sur les ressources à utiliser pour produire une architecture de circuit fonctionnant à une fréquence minimale imposée. Il s'appuie sur un ensemble de fonctions heuristiques pour résoudre les problèmes de choix. L'affectation des ressources fonctionnelles aux opérations du filtre est effectuée comme suit :

—à chaque unité fonctionnelle capable d'exécuter l'opération on associe un critère statique caractérisant son adéquation à effectuer l'opération. Un additionneur simple aura par exemple un critère statique plus élevé pour une opération d'addition qu'une UAL ;

—un critère dynamique est attribué à chacune des unités fonctionnelles précédentes rendant compte de leur disponibilité et des connexions éventuelles existantes avec les éléments de mémorisation contenant les opérandes de l'opération concernée ;

—les unités fonctionnelles sont classées suivant un critère global obtenu en effectuant une somme pondérée des deux critères précédents ;

—les unités fonctionnnelles en tête du classement sont retenues pour être essayées ;

—la qualité de l'architecture est estimée par une approximation de son temps de réponse et de sa surface. Le temps de réponse est évalué par la somme des temps d'exécution de toutes les opérations du graphe décrivant le filtre. Pour les opérations non encore traitées, on retient comme temps d'exécution le barycentre des temps d'exécution des opérateurs fonctionnels capables de l'effectuer pondérés par les critères

globaux respectifs. La surface est estimée par la somme des surfaces des ressources fonctionnelles de l'architecture corrigée par un facteur rendant compte de la surface des interconnexions ;

—le choix retenu est celui qui débouche sur la meilleure architecture suivant les estimations de temps et de surface. L'estimation du temps de réponse permet d'éliminer les choix susceptibles de violer les contraintes de temps.

Bien qu'il ait produit des architectures convenables pour des filtres simples, IMHOTEP a été incapable de proposer une architecture même non optimale pour un filtre de complexité réelle [Fon 85].

Les circuits produits par la plupart des outils automatiques courants présentent des performances bien moindres de celles élaborées par des concepteurs humains. D'autre part, rares sont les outils capables de s'attaquer à des problèmes de taille réeelle. La faculté de traiter des problèmes de taille réelle nécessite des modèles de résolution complexes et flexibles. La faiblesse des modèles heuristiques vient du fait qu'ils sont obligés de confondre des situations différentes pour permettre leur traitement systématique à l'aide d'algorithmes convergents. Dans le cas de la synthèse automatique par exemple, bien qu'un corps de connaissances existe sur le sujet, aucune théorie unificatrice n'a été élaborée. Ces connaissances sont souvent issues de l'expérience des concepteurs et s'expriment sous la forme d'un ensemble de règles de conduite de la forme : dans telles situations, il est préférable de faire tel choix ou de suivre telle stratégie. Plusieurs de ces choix sont qualitatifs (styles d'architectures : pipe-line, parallèle...) et se prêtent mal à une formalisation heuristique.

Avantages des systèmes à base de règles

Pour résoudre des problèmes de complexité exponentielle, l'intelligence artificielle a permis le développement de systèmes dits à base de règles. Ces systèmes exploitent une base de connaissances exprimées sous forme symbolique pour guider leur comportement dans la recherche de la solution à un problème donné. La représentation des connaissances sous forme symbolique permet de mieux différencier les situations et de mieux prendre en compte les connaissances qualitatives précédemment évoquées. Cela permet de tirer parti de l'expérience des experts humains pour imiter leur comportement dans la résolution d'un problème, d'où le nom de système expert souvent donné aux systèmes à base de règles.

D'autre part les systèmes à base de règles présentent une flexibilité facilitant leur développement de façon progressive dite incrémentale. Le système, progressivement enrichi de connaissances exprimées sous forme de règles est capable d'exécuter des tâches de plus en plus complexes rendant ainsi automatique une part croissante de l'activité de conception des CI.

2.3. Solution retenue

Compte tenu de toutes les remarques précédentes, nous avons choisi une approche basée sur les techniques d'intelligence artificielle pour implémenter ASA. ASA est donc un système à base de règles. Comme tous les systèmes à base de règles, il est composé de :

—une base de faits qui contient la description du problème à résoudre ainsi que la solution lors de son élaboration ;

—une base de connaissances qui contient un ensemble de règles propres à la synthèse architecturale de circuits intégrés ;

—un moteur d'inférence qui exploite la base de connaissances pour résoudre le problème contenu dans la base des faits.

En général, pour les systèmes à base de règles, le moteur d'inférence est indépendant d'une application particulière. De tels moteurs existent dans le commerce et nous aurions pu utiliser l'un d'entre eux pour réaliser ASA. Pour des raisons qui seront expliquées au chapitre suivant, nous avons été amenés à écrire un moteur d'inférence pour implémenter ASA. Avant de présenter les détails de la solution retenue au chapitre 4, le chapitre suivant décrit le moteur d'inférence en justifiant les choix que nous avons effectués pour sa construction.

3. Le générateur de systemes experts ODSE

Dans ce chapitre, nous décrivons le générateur de systèmes experts (SE) ODSE qui nous a permis de construire ASA, le SE d'Aide à la Synthèse Architecturale de circuits intégrés VLSI. La première partie introduit la notion de générateur de SE. A travers une étude critique de quelques générateurs existants, il met en évidence l'importance et le rôle des générateurs dans la conception d'un SE. Cette étude, est aussi l'occasion de préciser les concepts d'intelligence artificielle qui nous ont semblés essentiels pour le développement de ASA. Ensuite, nous présentons les raisons qui nous ont amené à écrire un générateur de SE en langage ADA. Un aperçu du langage de programmation ADA est donné ainsi que la technique de conception orientée objets qui a présidé au développement de ODSE. Cette première partie se termine par un résumé des principales caractéristiques de ODSE.

La deuxième partie présente plus en détail le générateur ODSE. Le point de vue adopté est celui de l'ingénieur de la connaissance à qui incombe la tâche de construire les systèmes expert à partir d'un générateur. Cette partie est essentielle à la compréhension du chapitre suivant puisqu'on y expose les principaux composants de ODSE, le formalisme de représentation des faits et des règles ainsi que les mécanismes d'inférence et de contrôle mis en œuvre.

La troisième partie dévoile les mécanismes internes de ODSE. L'organisation interne de la mémoire (structures de données) est abordée ainsi que les techniques utilisées pour obtenir un filtrage efficace de la base de faits et une interprétation rapide de la base de règles.

La dernière section décrit l'implémentation en langage ADA du prototype ODSE.

3.1. Présentation generale

3.1.1. Les Systèmes à Base de Règles

Selon Hayes-Roth, les Systèmes à Base de Règles (SBR) constituent le meilleur moyen informatique actuellement disponible pour coder le savoir-faire des experts humains [Hay 85]. Ce savoir-faire est souvent exprimé par un ensemble de règles de la forme "Si situation alors action", ce qui privilégie les

SBR comme outils de développement. Un Système à Base de Règle[Hay 85] est constitué d'une base de connaissances et d'un moteur d'inférence. La base de connaissances contient des faits et des règles. Une règle est une granule de connaissance exprimée sous une forme symbolique et composée d'un antécédent (ou partie gauche) et d'un conséquent (ou partie droite). La partie gauche exprime un ensembles de conditions qui doivent être satisfaites par les faits pour que la règle soit **activable** (ou **tirable**). Interpréter une règle consiste à exécuter sa partie droite si elle est activable. Si la partie droite est une suite d'actions, l'exécution revient à effectuer ces actions. Si par contre c'est une conclusion, l'effet est de la déduire.

Les faits sont des assertions concernant des propriétés, relations, ou propositions exprimées statiquement sous une forme également symbolique. A la différence des règles que le SBR interprète comme impératives, les faits sont silencieux par rapport à la valeur pragmatique ou l'utilisation dynamique de leur contenu [Hay 85]. Nous dirons que les faits sont des **connaissances déclaratives** et les règles des **connaissances opératoires**.

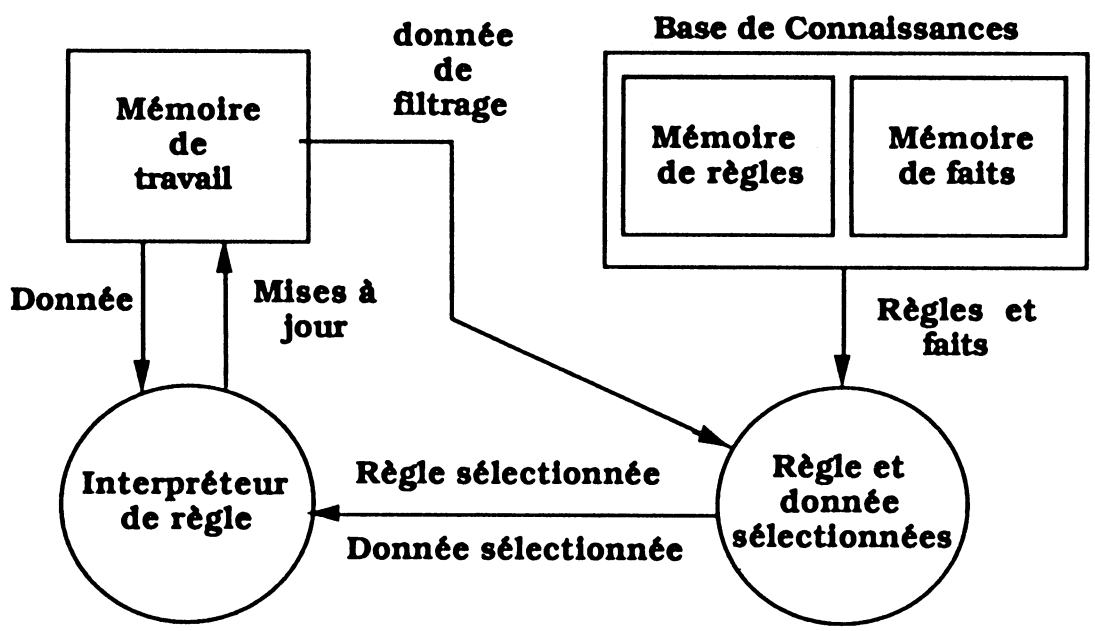


Figure 1 : Principaux composants d'un système à base de règles ([Hay 85])

En plus des connaissances statiques stockées dans la base de connaissances, un SBR utilise une mémoire de travail pour ranger les données initiales du problème à résoudre et les assertions intermédiaires produites par

l'interprétation successive des règles. La mémoire de travail décrit l'état d'avancement du problème en cours de solution. La figure 1 schématise les principales caractéristiques d'un système à base de règle.

Le moteur d'inférence ou interpréteur de règles est un programme général qui sélectionne les règles et les interprète dans un cycle de recherche de solution. Il détermine le moment et l'ordre dans lequel les règles doivent être interprétées.

3.1.2. Les générateurs de systèmes experts

Un **Générateur de Système Expert (GSE)** est constitué de la donnée d'un formalisme de représentation des problèmes et des connaissances ainsi que d'un moteur d'inférence. Construire un système expert revient à transcrire sous forme de règles et de faits, les connaissances propres au domaine d'application visé, dans le formalisme de représentation des connaissances proposé par le GSE.

Le choix d'un générateur de systèmes experts est une étape importante lors de la construction d'un SE. Les générateurs diffèrent entre eux par les formalismes de représentation des connaissances et les stratégies de filtrage, d'unification et de recherche mises en oeuvre par le moteur d'inférence.

Pour construire notre SE d'aide à la synthèse architecturale, nous nous sommes intéressés à un certain nombre de générateurs (§ 3.1.3) que nous n'avons pu utiliser soit parce que nous n'en disposions pas, soit à cause des contraintes (§3.1.4) issues de la politique suivie par le département Architecture de Micro-Systèmes (AMS) du CNET en matière de développement de logiciels. Cela nous a conduit à construire **ODSE** un générateur de système expert dont nous présentons les particularités à la fin de cette section.

3.1.3. Quelques générateurs existants

3.1.3.1. PROLOG

PROLOG est basé sur la théorie des prédicats du premier ordre et la programmation logique. Les problèmes et les connaissances opératoires doivent être décrits à l'aide d'un ensemble de prédicats. La résolution de problème est envisagée comme une démonstration automatique de théorème : le théorème à démontrer modélise le problème à résoudre ; les faits sont considérés comme un ensemble de prémisses et les connaissances comme un ensemble d'axiomes ou de théorèmes déjà démontrés ; le moteur d'inférence, suivant une stratégie

de cheminement arrière, déduit des faits nouveaux par application aux prémisses d'axiomes (clauses) sélectionnés jusqu'à l'obtention de la propriété à démontrer. En cas d'échec (impasse), le moteur retourne sur ses pas jusqu'au point de choix le plus récent pour emprunter une autre voie. Cette démarche conduit à une exploration systématique de toutes les solutions possibles si on n'utilise pas le prédicat "/" ("cut") pour arrêter l'exploration.

Comme générateur de systèmes experts, PROLOG présente quelques limitations :

—les prédicats, malgré leur puissance de description, ne sont pas toujours bien adaptés à la description de situations complexes. Dans une forme propositionnelle, il faut plusieurs formules différentes dispersées dans la base de faits pour décrire un objet. Ceci conduit à une granularité trop fine de la base de faits, ce qui ralentit les accès à cette base (cf figure 2). Un des reproches couramment faits à PROLOG réside dans sa lenteur. Ce problème peut être atténué utilisant des techniques d'indexation des clauses dans les bases [Cha87] ;

—la sélection des clauses est figée et déterminée par l'ordre dans lequel les clauses ont été écrites ;

—le contrôle de résolution est fait suivant une stratégie figée d'exploration systématique en profondeur avec retour en cas d'échec au dernier choix effectué. Le cut "/" de PROLOG apporte quelques améliorations mais il ne permet pas un retour guidé par le contexte. En outre, le chaînage avant n'existe pas en PROLOG ;

—PROLOG n'est pas bien adapté au traitement de problèmes qui se posent en termes d'obtention d'une solution optimale pour des critères donnés. Celà ne semble possible qu'à travers une exploration systématique de l'espace des solutions avec comparaison des solutions entre elles ;

—PROLOG n'offre pas un moyen d'étendre les stratégies de résolution par une couche de méta-connaissances sinon en les mêlant aux connaissances.

Ces limitations sont en partie surmontées par des systèmes qui comme METALOG [Meh 83] élargissent les stratégies figées de PROLOG en l'assistant d'un méta-interpréteur et en offrant un formalisme d'expression des méta-connaissances pour guider la résolution du problème.

TYPE(add1, operateur)
OPERATION(add1,plus)
SURFACE(add1,300)
DELAI(add1,10)

a) description d'un additionneur à l'aide de clauses

(OPERATEUR add1
(operation plus)
(surface 300)
(delai 10))

b) le même additionneur vu comme une structure à champs multiples (objet) : un terme au lieu de quatre précédemment

Figure 2 : granularité comparée de la représentation clausale et la représentation par structure

PROLOG [Bar 83] est disponible au CNET et nous avons même commencé un premier prototype de générateur automatique d'architecture de circuits-intégrés VLSI. Nous avons dû abandonner ce prototype à cause des limitations évoquées et surtout pour nous conformer à l'orientation suivie par le département(cf paragraphe 3.1.4.).

3.1.3.2. Les systèmes de production

OPS5 est un système de production. Il utilise deux mémoires pour représenter les problèmes et les connaissances : la mémoire de travail modèlise le problème par une collection d'objets structurés en classes ; la mémoire de production contient un ensemble de règles de la forme condition-action. Chaque règle peut modifier la mémoire de travail ou même dans certains cas la mémoire de production si ses conditions sont vérifiées.

Le moteur d'inférence effectue continuellement un cycle au cours duquel il évalue l'ensemble des règles activables (ensemble de conflit), en choisit une et l'exécute. La solution du problème initial est ainsi élaborée par la séquence de règles exécutées par le moteur d'inférence. Cette stratégie est connue sous le nom de chaînage avant.

Ce qui nous a attiré vers OPS5 c'est l'efficacité du filtrage : les règles manipulées par OPS5 sont compilées et efficacement interprétées suivant un algorithme dit de RETE [For 82]. Ce procédé rend l'interprétation d'un ensemble de règles indépendant du nombre de règles ; d'où son intérêt pour des systèmes manipulant un grand nombre de règles. D'autre part le formalisme de représentation proposé par OPS5 permet une représentation plus compacte que PROLOG (cf fig 2). Bien que OPS5 repose sur le chaînage avant, il offre plusieurs critères de choix de la règle à tirer (MEA, LEX.) qui permettent d'émuler d'autres stratégies de contrôle (chaînage arrière, réduction de problèmes.) moyennant un ensemble de méta-règles plus ou moins important. Par exemple R1 [McJ 82] est un système expert permettant de configurer les ordinateurs VAX selon les besoins des clients. La version initiale, écrite avec OPS4 contenait 722 règles dont 292 indépendantes de la tâche de configuration (méta-règles) et utilisées entre autres choses pour contrôler la décomposition de tâches en sous-tâches. Signalons que OPS5 a aussi été utilisé pour la synthèse architecturale de circuits intégrés VLSI [Kow 85].

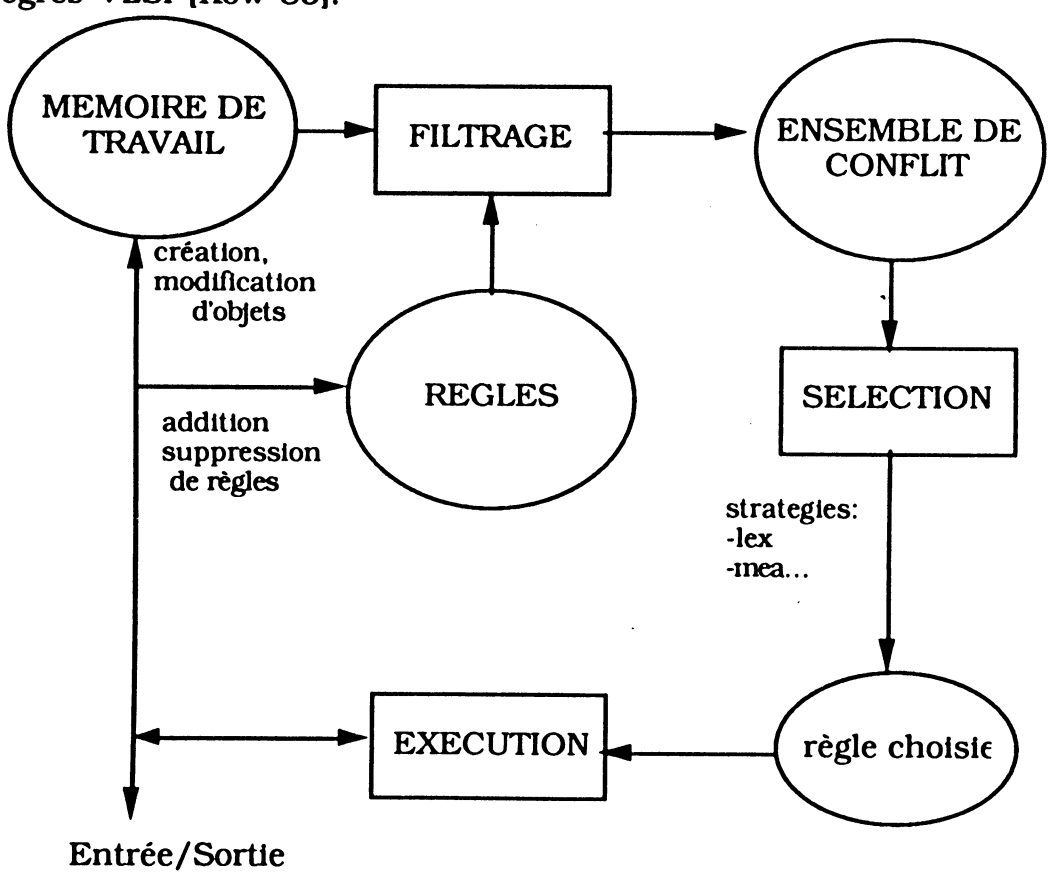


Figure 3 : fonctionnement général de OPS5

OPS5 ne possède aucun mécanisme de gestion des retours-arrière ("BACKTRACKING")

Le débogage et la maintenance d'une base de règles écrite en OPS5 sont rendus difficiles par :

—le non déterminisme qui est induit par les critères dynamiques de sélection des règles. On ne peut pas toujours prévoir statiquement l'ordre d'exécution des règles de production puisque les procédures de sélection tiennent compte de facteurs qui ne sont connus qu'au moment de l'exécution comme l'ordre d'introduction des faits dans la mémoire de travail ou les dates d'exécution des règles. L'opacité du contrôle de résolution est accentuée par le fait que les règles sont isolées les unes des autres (une règle ne peut commander l'interprétation d'une autre règle). Elles ne sont pas structurées en une hierarchie dans laquelle une règle pourrait être composée de plusieurs autres règles comme c'est le cas des procédures dans les programmes conventionnels [Ban1 81]

—la communication des informations entre les règles s'effectue par modification des "variables globales" (objets) contenus dans la mémoire de travail. Cette démarche est connue sous le nom d'"effet de bord".

3.1.3.3. Les Systèmes de Réduction de Problèmes

La réduction de problèmes ou planification repose sur la décomposition d'un problème en sous-problèmes dont la résolution fournit une solution au problème initial. Cette technique présente un intérêt pour des problèmes qui peuvent être décomposés en sous-problèmes de complexité moindre. Ces décompositions sont exprimées en termes de plans. Un plan pour un but donné est un ensemble de sous-buts qu'il faut atteindre pour accomplir le but initial.

Un Réducteur de Problème résout un problème en :

- cherchant un plan pour ce problème : on parle de génération de plan ;
- et en exécutant ce plan : on parle de coordination.

TROPIC [Lat 79], SIPE [Wil 84], MOLGEN... sont des générateurs de S.E. basés sur le concept de planification hiérarchique. Ils intègrent les principes d'abstraction et de propagation de contraintes.

L'abstraction [Sac 74] désigne l'aptitude qu'ils ont à s'attaquer aux problèmes les plus importants avant de se préocuper des détails. La génération de plan procède par raffinement successif. La solution est d'abord esquissée par un plan vague mais complet dont les parties sont progressivement et récursivement précisées jusqu'à l'obtention d'une séquence complète et

détaillée d'opérateurs primitifs dont l'exécution produit la solution cherchée. L'intérêt de l'abstraction est de réduire le temps consacré à la génération d'un plan : une recherche de complexité O(n) peut être réduite à O(Log n) [Korf 87]. Cette démarche présente beaucoup d'affinités avec les méthodes de conception descendante des circuits-intégrés que nous avons évoquées au chapitre 1.

La recherche de plan est une tâche pénible et sujette à beaucoup d'erreurs. Ces erreurs sont souvent issues des interactions facheuses entre les sous-problèmes d'un plan en cours d'élaboration. Deux problèmes interagissent lorsque la solution de l'un peut remettre en cause la solution de l'autre.

Pour traiter le problème d'interaction entre les sous-problèmes, EL et TROPIC font recours à un retour arrière guidé par le contexte. Ils évitent ainsi d'explorer systématiquement l'espace de recherche en analysant les causes d'erreurs avant de choisir une autre alternative. Alors que EL et TROPIC mettent en oeuvre des méthodes indépendantes du domaine d'application concerné pour effectuer l'analyse des échecs, GARI [Des 85] et plus récement VT [Mar 88] utilisent des connaissances spécifiques du domaine.

MOLGEN ([Ste 81a] et [Ste 81b]) suit une stratégie de moindre engagement basée sur la propagation de contraintes [Ste 81] et la métaplanification [Ste 81]. Il évite de prendre une décision jusqu'à ce que toutes les contraintes affectant la décision soient connues ou la déterminent. La métaplanification est réalisée par une couche de connaissances (méta-connaissance) qui propose de manière structurée, les décisions à prendre lorsque les contraintes laissent plusieurs alternatives possibles [Ste 81b]. Comme MOLGEN, SIPE essaie une stratégie de moindre engagement basée sur la propagation des contraintes. Mais l'interaction entre les sous-problèmes est traitée lors de la coordination par un raisonnement sur les ressources utilisées [Wil 84].

Ce qui nous rebute en TROPIC et EL c'est le formalisme de représentation qui est dérivé de la logique des prédicats du premier ordre. SIPE résout ce problème en proposant en outre le concept d'objet qui permet de regrouper les caractéristiques de l'opérateur fonctionnel dans une structure compacte semblable aux articles utilisés dans les langages de programmation évolués. En plus SIPE offre un formalisme d'expression des contraintes adapté à la description d'objets partiellement définis.

L'inconvénient majeur que nous avons trouvé aux planificateurs hiérarchiques que nous avons étudiés réside dans la complexité de la génération des plans. Dans ces générateurs, la génération de plan est traitée comme un problème de recherche heuristique. Il s'agit de trouver un ordonnancement

adéquat des opérateurs permettant de résoudre un problème. Cette recherche peut être longue et sujette à beaucoup d'erreurs. Le retour arrière nous semble une solution trop coûteuse malgré les améliorations apportées par TROPIC, EL et plus récemment GARI et VT. Les stratégies suivies par MOLGEN et SIPE, basées sur le moindre engagement ("least comitment strategie") et la propagation de contraintes nous semblent plus adaptées à la synthèse architecturale de circuits-intégrés malgré leur incomplétude. En effét les choix des concepteurs sont très souvent le fruit de leur expérience qui leur suggèrent les bonnes décisions à prendre sans avoir besoin de les évaluer toutes. Dans la conception de ASA (cf chapitre 4), nous entendons miser d'avantage sur cette intuition issue de l'expérience. Néanmoins nous envisagerons les stratégies de retour guidé par le contexte (dependency directed backtracking) pour faciliter l'exploration de plusieurs solutions alternatives.

Les systèmes de résolution de problèmes font encore l'objet d'une activité de recherche. Nous n'avons donc pas pu en disposer pour construire ASA. D'autre part la plus part de ces générateurs sont programmés en LISP ou en PROLOG, langages incompatibles avec les options choisies par le département AMS pour le développement des logiciels. Ces considérations nous ont amenés à développer ODSE, Outil de Développement de Systèmes-Expert.

3.1.4. Contexte de développement de ODSE

Dans une première sous section, nous présentons le cadre dans lequel ODSE a été développé.

Le langage Ada utilisé pour la programmation du générateur est ensuite succintement décrit en insistant sur les aspects qui nous ont été particulièrement utile pour la conception de ODSE.

La section se termine par un exposé de la méthode dite de conception orientée objet qui nous a guidé dans le développement de ODSE.

3.1.4.1. Contexte du travail

Le département AMS du CNET Grenoble au sein duquel le travail s'est déroulé avait choisi Ada comme langage de développement unique afin de limiter les problèmes de portabilité, d'interfaçage avec des programmes écrits dans des langages différents et d'appliquer une politique cohérente en matière de génie logiciel. Certains logiciels initialement écrits en Pascal comme LOF ont été entièrement repensés et réécrits en Ada. Cette migration des logiciels vers Ada s'est accompagnée du développement d'une bibliothèque de composants logiciels qui s'est avérée très utile comme nous le montrerons plus loin.

D'autre part, divers logiciels en cours de développement au sein département (routage, synthèse électrique) ont mis en évidence le besoin d'un recours à des techniques d'intelligence artificielle pour résoudre certains problèmes locaux. Par exemple un compilateur de cellules en cours d'étude nécessite de pouvoir identifier des configurations opportunes de transistors pour optimiser la surface et les performances de la cellule. Cette identification opportune est candidate à une réalisation sous forme de système expert.

La disponibilité d'un générateur de systèmes experts écrit en Ada et pouvant s'intégrer aisément aux logiciels Ada s'est ainsi avérée utile pour rester conforme à la politique suivie par le département. Le reste de cette section donne un aperçu du langage Ada en insistant sur les aspects qui nous ont été d'une grande utilité dans le développament de ODSE. La méthode dite de conception orientée objets que nous avons suivie dans le développement de ODSE est exposée

3.1.4.2. Le langage ADA

Ada est un langage de programmation conçu par Jean Ichbiah de la Honneywell-BULL à l'époque, sur appel d'offre public du département de la défense des Etat-Unis (DoD) en vue de maîtriser les coûts de conception et de maintenance des logiciels et aussi pour surmonter ce qu'il est devenu commun d'appeler la crise du logiciel. Cette crise se manifeste par les symptomes suivants [Boo 83] :

- fiabilité douteuse des logiciels
- coûts excessifs et imprévisibles
- maintenance lourde et tatonnante
- retard dans les livraisons de logiciels
- portabilité pénible sinon impossible
- logiciels inefficients en temps et mémoire consommés

Pour surmonter cette crise, Ada se présente comme un langage de programmation couvrant un grand nombre de domaines d'applications et intégrant des concepts modernes de génie logiciel adaptés à la conception de logiciels complexes. Pour une description complète du langage, on peut consulter le manuel de référence Ada ou encore [Ber 89]. Nous nous limitons ici aux caractéristiques importantes du langage qui nous ont aidé au développement de ODSE.

Ada peut être vu comme une extention des langages de programmation procéduraux comme Pascal. En plus des concepts issus de la programmation structurée (découpage en procédures, typage des variables.), Ada offre la notion de paquetage pour la réalisation de types abstraits, la généricité qui facilite la

conception de composants logiciels ré-utilisables, les exceptions qui permettent un traitement propre des événements qui surviennent en cours d'exécution d'un programme.

Les paquetages

Le paquetage est une unité de programmation ADA qui permet d'encapsuler un groupe d'entités ayant des caractéristiques communes en contrôlant leurs interactions avec le monde extérieur. Il est composé d'une interface et d'un corps. L'interface spécifie les interactions possibles avec le monde extérieur sous la forme d'un ensemble de types de données manipulées, d'en-tête de procédures, de fonctions ou d'autres paquetages. Ada offre un ensemble de formalisme permettant de limiter la vision et les actions possibles du monde extérieur sur les entités définies par le paquetage.

Les exceptions

Les exceptions permettent de réagir aux évènements qui surviennent lors de l'exécution d'un programme. Ces évènement sont appelés exceptions. Ils peuvent être programmés ou provoqués par la détection d'une anomalie dans un programme en cours d'exécution. Ada offre un formalisme pour spécifier statiquement le traitement à effectuer lorsqu'une exception est levée. Cet aspect de ADA nous a été particulièrement utile pour le développement de ODSE comme nous le verrons à la section 3.2.1.6

La généricité

La généricité permet de définir un ensemble de traitements applicables à toute une classe de types de données. Un programme générique est comme un générateur de programmes. Il définit un traitement en utilisant des paramètres dits génériques parcequ'ils sont incomplètement définis. Ces paramètres peuvent être des valeurs, des types, des fonctions et des procédures. Une unité de programme générique doit être instantiée avant d'être utilisée. L'instantiation consiste à produire un programme complètement défini à partir d'un programme générique en précisant la définition des paramètres génériques.

La généricité factorise l'effort de programmation en permettant d'instantier un même jeu de traitements sur des types de données différents. Cet aspect a favorisé le développement de bibliothèques de composants logiciels réutilisables [Boo 83]. Le département AMS du CNET GRENOBLE dispose d'une telle bibliothèque qui nous a été utile lors du développement de ODSE. Le paquetage d'addressage dispersé ("hashed code"), le générateur automatique d'analyseur de syntaxe, le paquetage d'expressions symboliques que nous avons utilisés ont étés tirés de cette bibliothèque.

Ada permet aussi de traiter le parallélisme (tâches, synchronisation). Nous nous sommes limités ici aux aspects du langage qui nous ont été les plus utiles au développement de ODSE.

3.1.4.3. Conception orientée objets

Pour tirer parti du langage ADA, nous avons suivi la démarche proposée par Grady BOOCH [Boo 83] et connue sous le nom de conception orienté objets. Bien qu'on puisse y trouver des ressemblances, il faut bien distinguer la conception orientée objet de la programmation par objets. La conception orientée objets est une discipline de développement de logiciel tirant parti des enseignements du génie logiciel et des possibilités offertes par le langage ADA pour supporter ces enseignements. Pour un traitement complet du sujet, nous renvoyons le lecteur à l'ouvrage de Grady BOOCH [Boo 83]. Avant de présenter les grandes lignes de la conception orientée objet, et pour éviter toute confusion, nous rappellerons les caractéristiques essentielles de la programation par objets.

Les langages à objets

Le langage à objets le plus connu est SMALLTALK [Gol 83]. Il offre un puissant formalisme de description des objets ainsi que les relations entre les objets. Les notions de modularité et d'abstraction (types abstraits) encouragées par le génie logiciel y sont réalisées par les concepts de classes et d'objets. Un objet est une entité appartenant à une classe donnée et possédant une mémoire locale décrivant son état. Une classe regroupe un ensemble d'objets partageant un ensemble de fonctionnalités spécifiques appelées méthodes. Le concept **d'héritage** qui permet de définir de nouvelles classes en particularisant des classes existantes est très utile pour décrire des situations caractérisées par une classification hiérarchique d'objets (une bibliothèque d'opérateurs par exemple). Le contrôle s'effectue en SMALLTALK par un échange de messages entre les objets. Lorsqu'un objet reçoit un message, le système cherche dans l'ensemble des méthodes caractérisant la classe de l'objet, celle qui correspond au message reçu et l'exécute en lui passant les paramètres d'appel. Une exécution peut engendrer des envois de messages. L'intérêt de SMALLTALK réside dans la puissance et l'efficacité du formalisme de représentation des objets qu'il offre. D'autre part, il permet une conception rigoureuse des programmes par les principes de génie-logiciel qu'il intègre (modularité, abstraction). Cependant, en 1986 (début de la thèse), il constituait encore un système fermé dont la communication était difficile avec les langages de programmation évolués [Tag 87]. Les choses ont changé depuis lors, puisque SMALLTALK est maintenant disponible sur la plupart des stations de travail (Apollo, SUN, MacII...).

SMALLTALK a été utilisé pour le développement de GSE comme OSMOSE, conçu et réalisé au CNET Lannion sur un modèle dérivé de OPS5 [Ali 87].

Les langages à objets favorisent un style de programmation connu sous le nom de programmation par objets : la tâche du programmeur consiste à modéliser l'environnement de son application en termes d'objets communiquant par envoi de messages. La programmation par objet n'est pas possible avec ADA puisque le mécanisme d'héritage et de communication par messages (au sens SMALLTALK) y sont absents. Malgré ces limitations, ADA possède la généricité et les paquetages qui permettent moyennant une discipline de programmation, de se rapprocher de la programmation par Objets dont les avantages ont été évoqués plus haut. Cette discipline que nous avons observé pour le développement de ODSE est connue sous le nom de conception orientée objets et décrite paragraphe suivant.

La conception orientée objets

La conception orientée objets (ou par objets) [Boo 83] procède par affinement successif pour décomposer le logiciel à construire suivant des abstractions (objets) plutôt que suivant des actions comme c'est le cas avec la programmation structurée. Une abstraction est vue ici comme la représentation d'une réalité physique avec ses caractéristiques (valeurs) et ses propriétés (opérations). Dans une décomposition, chaque abstraction sera réalisée par un paquetage. Le programme sera constitué d'un ensemble de paquetage qui interagissent pour élaborer la solution voulue. Les grandes ligne de la démarche préconisée par Grady BOOCH sont les suivantes :

—identifier les abstractions ;

—identifier les opérations ;

—établir la visibilité : il s'agit de définir les relations entre les objets. Pour chaque "objet", on définit les "objets" auxquels il peut accéder ainsi que les "objets" qui pouront l'atteindre. De cette étape dépend le réseau de dépendance des divers paquetages ;

—définir l'interface de l'objet avec le monde extérieur. Cette interface constitue la spécification du paquetage ;

—implémenter les objets en écrivant le corps du paquetage ;

A la section 3.4, nous décrirons la structure des paquetages obtenue par application de cette méthode. Pour le moment, nous présentons les caractéristiques principales de ODSE au paragraphe suivant, suivies par une description détaillée aux sections 3.2. et 3.3.

3.1.5. Présentation générale de ODSE

ODSE est un Outil de Développement de Systèmes Experts basé sur les concepts de planification et de réduction de problèmes. Nous donnons dans cette section les caractéristiques principales de ODSE en montrant leur intérêt pour le problème de synthèse architecturale que nous avons à résoudre.

Représentation des faits

Les faits sont représentés par des objets. Ces objets peuvent être organisés en graphes. En outre, ODSE offre un langage d'expression des contraintes permettant d'associer à tout objet de la base, un ensemble de contraintes. Ce formalisme est adapté à la représentation des informations incomplètes ou partielles qui interviennent au cours de la résolution d'un problème. Celà permet aussi une meilleure représentation des propriétés dynamiques des objets.

La base de faits (bdf) et la base de connaissances (bdc) sont indexées pour permettre un filtrage efficace de la bdf et un accès rapide à la bdc.

Représentation des connaissances

ODSE représente les connaissances opératoires sous la forme d'un ensemble de plans. Chaque plan contient les grandes lignes de résolution d'un problème spécifique. Il exprime des circonstances dans lesquelles un problème complexe peut être décomposé en sous-problèmes. Ce mode de représentation se prête très bien à la modélisation de l'expérience des concepteurs en matière de conception des circuits intégrés. La sémantique associée à un plan peut être résumée par la phrase suivante :

"Pour effectuer telle tâche, lorsque telles conditions sont remplies par la base de faits, on peut effectuer telles étapes."

Un plan décrit une manière d'accomplir une tâche donnée. Plusieurs plans peuvent proposer des solutions diverses pour résoudre un même problème, à charge au moteur d'inférence d'en sélectionner une pour le résoudre. Un plan pour concevoir une architecture de circuit intégré peut s'exprimer ainsi :

"Pour construire l'architecture d'un circuit intégré, si on recherche le circuit le plus rapide sans limitation des ressources utilisées, il faut :
- ordonnancer les opérations du circuit
- allouer les ressources fonctionnelles
- définir les interconnexions
- effectuer les optimisations possibles"

Stratégies de résolution de problème

ODSE remplace le problème de la génération de plan (recherche de plan) par la la consultation d'un catalogue contenant un ensemble de plans. De cette manière, la recherche est rendue plus déterministe. Une telle demarche exige une grande quantité de connaissances spécifiques du domaine d'application. L'efficacité du SE repose sur la qualité des plans de la base de connaissance.

ODSE combine plusieurs stratégies de résolution de problème : chaînage-avant (démons), chaînage arrière (théorème) et chaînage-mixte.

La gestion des exceptions et des interruptions bien connue des programmeurs ADA [Ber 89] est possible dans ODSE à travers une catégorie particulière de démons.

ODSE intègre la notion de MACRO (bien connue en LISP) permettant d'élargir les stratégies de résolution de problèmes mises en oeuvre à des méthodes plus spécifiques des problèmes que l'on veut traiter. Les méthodes de recherche dites faibles, par exemple, peuvent être programées par des macros.

ODSE dispose de fonctions spéciales qui assurent l'interaction avec l'utilisateur au cours de la résolution d'un problème.

Programmation : langage ADA

ODSE offre un formalisme permettant d'activer des procédures écrites en ADA. Réciproquement, plusieurs primitives ODSE permettent à des programmes écrits en ADA de consulter et modifier la base de faits sous le contrôle du moteur d'inférence.

ODSE présente d'autres caractéristiques que nous évoquerons dans le reste du chapitre. Le prochain paragraphe présente avec plus de détail les composants de ODSE. ODSE y est présenté du point de vue de l'ingénieur de la connaissance à qui incombe la tâche de construire le système expert. Le paragraphe 3.3 examine en profondeur les mécanismes d'inférence mis en oeuvre. Le dernier

paragraphe décrit l'implantation du prototype de ODSE ainsi que les techniques de programmation utilisées.

3.2. Description de ODSE

La section 3.2.1 décrit la base de faits ainsi que les commandes qui permettent sa construction et sa modification. Le paragraphe 3.2.2 décrit la syntaxe et l'interprétation des diverses catégories de règles.

3.2.1. La base de faits

La base de faits est un ensemble d'objets structurés en classes et en graphes et modélisant l'ensemble des problèmes dont on souhaite la résolution. ODSE offre plusieurs primitives permettant de construire et de modifier la base de faits.

3.2.1.1. Les objets

La représentation des faits en ODSE est centrée sur le concept d'objet. Un objet est caractérisé par une classe, un nom et un ensemble de couples attribut-valeur(s). Le nombre d'attributs d'un objet n'est pas limité à priori et peut varier dynamiquement en cours d'exécution.

Les attributs peuvent prendre des valeurs numériques ou symboliques. Elles peuvent aussi avoir une ou plusieurs valeurs.

La classe d'un objet joue un rôle un peu semblable à la notion de type dans la programmation conventionnelle. Elle définit un ensemble minimal d'attributs pour les objets de la classe et détermine l'ensemble des opérations (opérateurs, démons cf section suivante) qui leur seront applicables.

3.2.1.2. Construction de la base des faits

Deux directives principales permettent de définir les classes en ODSE : **def_attr** et **def_obj**

def_attr définit un attribut en précisant :

—son nom ;

—son domaine de définition : ensemble discret fini ou infini de symboles ou d'entiers qu'il pourra prendre comme valeur ;

—son mode : mono ou multivalué.

def_attr *category* ***singlevalued*** *(storage bus multiplexer f_unit)*

définit l'identificateur category comme un attribut monovalué (***singlevalued***) pouvant prendre ses valeurs dans l'ensemble fini de symboles *(storage, bus, multiplexer, f_unit)*. Les directives :

```
def_attr input1 multivalued atom
def_attr input2 multivalued atom
def_attr output multivalued atom
```

définissent input1, input2, output comme des attributs multivalués pouvant prendre n'importe quelles valeurs litérales ou numériques.

```
def_attr surface singlevalued 0 100000
```

définit *surface* comme un attribut numérique monovalué pouvant prendre une valeur comprise entre 0 et 100000.

def_obj définit une classe d'objet en lui associant un nom et un ensemble d'attributs.

```
def_obj resource (category surface time input1 input2)
```

crée une classe d'objets appelée *resource* dont les attributs *category, surface, time, input1, input2* et *output* sont contraints selon les directives ***def_attr*** ci-dessus présentées Lorsque un attribut intervenant dans la définition d'une classe n'a pas été défini préalablement à la définition de la classe, l'attribut en question n'est soumis à aucune contrainte particulière.

Les deux directives que nous venons de voir permettent de définir les attributs et les classes de la base des faits.

3.2.1.3. Les opérateurs primitifs

L'ensemble des opérations que l'on peut effectuer sur les objets constituent ce que nous appellerons les opérateurs primitifs ou actions primitives. Elles se traduisent bien souvent par la modification de la base des faits. On peut les regrouper en trois catégories :

—la création et la suppression des objets et des graphes (voir plus loin) ;

—les actions modifiant la valeur des attributs ;

—les actions sur les contraintes.

3.2.1.4. Création des objets

Les objets sont créés par l'action **MAKE**.

```
(make (resource additionneur1)(category f_unit))
```

crée un objet de la classe resource et lui associe le nom resource1. Avant l'exécution de cette action, il faut que la classe resource ait été définie. Seul l'attribut category est initialisé à la valeur `f_unit`. Les autres attributs n'ont pas de valeur. En l'occurrence, si on commande l'affichage de `additionneur1`, on obtient :

```
*************************
RESOURCE : ADDITIONNEUR1
-------------------------
category : f_unit ; constraints : ((in_set (storage, bus, multiplexer,
     f_unit))
-----------------------------
surface : undefined ;constraints : ((ge 0)(le 100000))
-----------------------------
input1 : undefined
-----------------------------
input2 : undefined
-----------------------------
output : undefined
-----------------------------
```

On remarque qu'à chaque attribut est associé un ensemble de contraintes éventuellement vide *(input1, input2, output)* sur son domaine de valeurs. Ces contraintes ne sont pas statiques mais peuvent changer par l'exécution d'actions appropriées (cf § contraintes). De ce point de vue, la directive ***def_attr*** n'est qu'un moyen de préciser le jeu de contraintes initial caractérisant un attribut donné.

Pour supprimer un objet de la base, on utilise l'opérateur ***remove***.

```
(remove (resource resource1))
```

supprime l'objet *(resource resource1)* de la base des faits.

A tout objet de la base est associé deux attributs qui permettent d'organiser les objets de la base en arbres ou en graphes :

—le premier pointe vers un ensemble d'objets de la base appelés prédecesseurs de l'objet courant ;

—et le deuxième vers un ensemble d'objets appelés successeurs de l'objet courant.

La construction et la modification des graphes se fait à travers les opérateurs spécifiques :

—***add_successor***, ***add_predecessor*** permettent respectivement d'ajouter un successeur ou un prédécesseur à un noeud ;

—***retract_predecessor***, ***retract_successor*** permettent de supprimer respectivement un prédécesseur ou un successeur à un noeud donné.

Considérons comme exemple la séquence suivante tirée de la description fonctionnelle du circuit Equadiff donnée au chapitre 2 figure 3 :

```
u6 :=u-u4 ;
u :=u6-u5 ;
```

figure 3

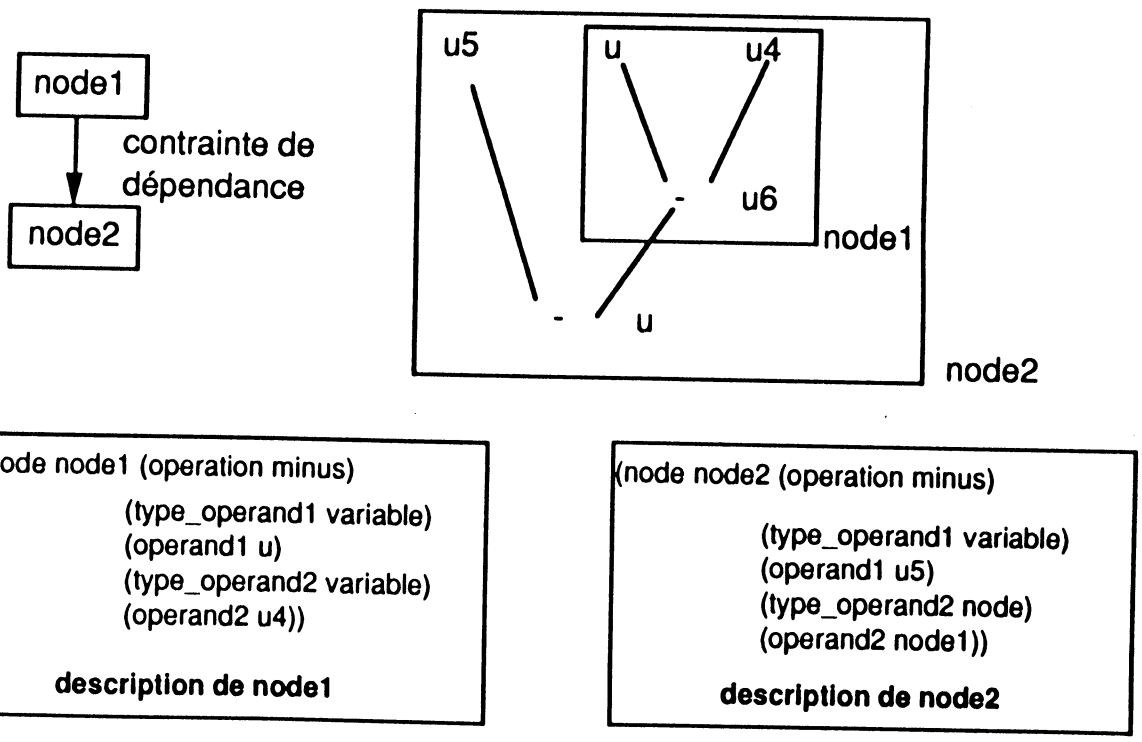


Figure 4

représentant deux instructions d'affectation. Dans ASA, ces deux instructions sont modélisées à l'aide d'objets appelés nodes.

Ces "*node*" sont organisés en graphes acycliques représentant les contraintes de dépendance entre les données intervenant dans les instructions d'affectation (cf fig 4). Ainsi, soient *node1* et *node2* les noms des objets représentants dans l'ordre les deux instructions d'affectation.

*(**add_successor** node2 (node node1))*

ajoute node2 à l'ensemble des successeurs de node1.

*(**add_predecessor** node1 (node node2))*

définit *node1* comme étant un prédecesseur de *node2*.

3.2.1.5. Modification des objets

ODSE propose une série d'opérateurs primitifs pour modifier les attributs d'un objet. La syntaxe générale de ces opérateurs est :

(<opérateur> (<classe><nom>){(<attribut><valeur>)})

<opérateur> désigne l'action effectuée :

—***augment***,ajoute une valeur à un attribut multivalué

—***modify*** modifie la valeur d'un attribut

—***retract*** supprime la valeur d'un attribut dans le cas d'un attribut monovalué et supprime une valeur à un attribut dans le cas d'un attribut multivalué

—***remove*** supprime l'objet de la base des faits

(<classe><nom>) désigne l'objet de la base qui est affecté par l'opérateur. *<classe>* représente la classe de l'objet et *<nom>* son nom.

{(<attribut><valeur>)} représente une liste éventuellement vide de couples *(<attribut><valeur>)*. Dans chacun de ces couples, <attribut> désigne l'attribut affecté par l'opérateur.

<valeur> est une expression combinant des informations de la base. L'évaluation d'une telle expression fournit une valeur qui est utilisée par <opérateur> pour modifier <attribut>.

L'expression :

[(resource additionneur1)category]

s'évalue à la valeur de l'attribut *category* de l'objet *additionneur1*. Conformément à la création précédente de cet objet, il est égal à *f_unit*. La syntaxe complète des expressions utilisées dans ODSE est donnée dans l' annexe A. Elle inclut les expressions arithmétiques sur les informations de la base, des opérations sur des ensembles, et même des fonctions (au sens du langage de programmation Ada). Par exemple

l'action

```
(modify(resource additionneur)(surface 4000))
```

affecte à l'attribut surface de l'objet resource1, la valeur 4000.

```
(modify(node node2)(alap_cycle
                (max alap_cycle (node in successors(node node2))-
                        [(famille [(node node1)operation])nbcycles]))
```

est une action qui calcule le cycle d'ordonnancement au plus tard (*alap_cycle*) d'un noeud donné. Ce cycle est égal au maximum des cycles d'ordonnancement au plus tard de ses successeurs auquel on soustrait le nombre de cycles (*nbcycles*) dont il a besoin pour s'exécuter (cf chapitre 4). Ce nombre de cycles est obtenu en prenant la valeur de l'attribut nbcycles de la famille des opérateurs désignée par l'attribut operation de *node1*.

3.2.1.6. Les exceptions

Les opérateurs primitifs peuvent lever des exceptions lorsque des anomalies sont détectées (objet inexistant dans la base, violation de contraintes). Lorsqu'une exception est levée, le cheminement normal du programme en cours d'exécution est interrompu et le contrôle est passé à un moniteur qui traite l'exception. Dans ODSE, le traitement des exceptions est effectué par une catégorie particulière de démons que nous verrons ultérieurement.

L'action

```
(modify (resource additionneur1)(surface 200000))
```

lève l'exception ***le_constraint_violation*** qui indique une violation de la contrainte portant sur la borne inférieure de l'attribut surface de l'objet *additionneur*. Si *additionneur1* est inexistant, dans la base, une autre exception est levée.

En annexe, nous donnons la syntaxe complète des divers opérateurs ainsi que la descriptions des exceptions qu'ils peuvent lever.

3.2.1.7. Actions sur les contraintes

Comme nous l'avons fait remarquer, chaque attribut d'un objet est caractérisé par un jeu de contraintes dynamique. L'ensemble des contraintes pesant sur un attribut à un instant donné définit, pour cet attribut, un domaine de valeurs possibles. Dans cette perspective, la définition d'un attribut (directive ***def_attr***), équivaut à affecter à cet attribut, un jeu initial de contraintes. La définition de l'attribut surface donnée plus haut contraint initialement cet attribut à prendre ses valeurs dans le domaine défini par les nombres compris entre 0 et 100000. L'attribut *category* a un domaine de valeurs initialement contraint à l'ensemble des valeurs *(f_unit, storage, bus)* alors que les attributs *input1*, *input2*, et *output* n'ont aucune contrainte initiale sur leur domaine.

L'évolution du jeu de contraintes initial d'un attribut est assurée par deux actions primitives : ***add_constraint*** et ***relax_constraint***.

add_constraint renforce les contraintes pesant sur un attribut donné en réduisant d'avantage le domaine des valeurs de cet attribut.

```
(add_constraint    (node node2)
     (cycle >=
          [(node node1)cycle]+
               [(famille[(node node2)operation])nbcycles]))
```

par exemple contraint l'attribut *cycle* de *node2* à être supérieur ou égal à l'attribut correspondant de *node1* auquel on ajoute le nombre de cycles (*nbcycles*) dont *node2* a besoin pour s'exécuter. Cette contrainte garantit que node2 ne pourra s'exécuter qu'après l'exécution de node1 dont il utilise le résultat comme un de ses opérandes. Dans ASA, ***add_constraint*** est largement utilisé pour propager les contraintes d'ordonnancement issues des graphes acyclique comme nous le verrons au chapitre 4.

Lorsque ***add_constraint*** réduit le domaine d'un attribut à une seule valeur, cette valeur est affectée à cet attribut. Par exemple les actions :

```
(add_constraint (resource additionneur1)
     (operation in_set '(add mult minus))

(add_constraint resource additionneur1)
```

(operation1 ***not_in_set*** *'(mult minus))*

contraignent l'attribut operation de l'objet *(resource additionneur1)* à l'unique valeur *add*. En effet la première action limite le domaine de operation à l'ensemble

{add, mult, minus}

tandis que la deuxième exclut de ce domaine, les valeurs mult et minus. Après l'exécution de ces deux actions, *[(resource resource1)operation]*=add.

relax_constraint permet d'alléger les contraintes portant sur l'attribut d'un objet. On peut ainsi étendre le domaine des valeurs possibles de l'attribut d'un objet.

L'exécution de ***add_constraint*** et ***relax_constraint*** peut lever des exceptions ; par exemple si une contrainte supplémentaire rend impossible l'affectation d'une valeur à un attribut, une exception est levée.

La description complète des actions ***add_constraint*** et ***relax_constraint*** est donnée en annexe A ainsi que la syntaxe des contraintes possibles et des exceptions levées.

3.2.1.8. Les entrées-sorties

Pour le moment, ODSE dispose de deux primitives permettant d'effectuer les entrées-sorties : ***ask*** et ***write*** dont les syntaxes suivent.

(ask<type> *[**in** <ensemble>])*
<type> : := ***integer/ atom***

*(**write**{<expr>})*
*(**writeln**{<expression>})*

ask est une fonction qui demande une valeur à l'utilisateur. La valeur demandée doit être soit un entier soit un atome suivant la valeur de <type>. La valeur demandée peut être contrainte à un ensemble de valeurs définies par <ensemble>. La fonction ***ask*** demande une valeur à l'utilisateur de manière itérative jusqu'à ce que la réponse soit conforme aux contraintes requises. Un message d'erreur est envoyé à l'utilisateur pour toute tentative infructueuse et une nouvelle valeur est demandée.

Par exemple

*(**ask integer in** 7..90)*

demande à l'utilisateur, un entier compris entre 7 et 90 ;

*(**ask atom in** '(f_unit storage bus))*

demande un symbole appartenant à l'ensemble indiqué.

write et ***writeln*** sont des actions primitives qui reçoivent en paramètre une liste d'expressions. Les expressions sont évaluées et le résultat affiché à l'écran dans l'ordre de leur occurrence. A la fin de l'affichage, ***writeln*** ajoute un saut de ligne.

Par exemple

```
(writeln "la surface de additionneur1 est"
         [(resource additionneur1)surface])
```

affiche à l'écran le message entre guillemets suivi de la valeur de l'attribut *surface* de l'objet *(resource additionneur1)*.

D'autres primitives existent qui permettent d'effectuer des entrées-sorties sur fichier (cf annexe).

3.2.2. La base de connaissances

La base de connaissances est constituée de règles. Les règles sont divisées en trois catégories : les opérateurs renferment des information sur la manière de décomposer les problèmes complexes, les théorèmes permettent de définir des concepts et des propriétés favorisant la concision des règles, et les démons permettent de propager les contraintes et de maintenir la cohérence de la base de faits après toute modification. Le moteur d'inférence utilise les règles de la base de connaissances pour décomposer et résoudre le problème décrit dans la base de faits.

Les parties gauches des règles expriment des conditions portant sur les objets de la base de faits (filtres) ou des prédicats. Avant de décrire les théorèmes, les opérateurs et les démons, nous précisons dans la suite, la syntaxe des filtres utilisées dans ODSE.

3.2.2.1. Les filtres

Les parties gauches des règles sont généralement décrites à l'aide de filtres. Un filtre est une expression définie par la syntaxe suivante donnée sous la forme Backus-Nauer :

```
<filtre> : :=
<variable> is <specif ensemble>
<variable> in <specif ensemble>
(<class> <variable> <lconditions>)
[ <expr1> <condition> <expr2> ]

<specif ensemble> : :=
{ <class> [in <expression symbolique>]
 [/ <lconditions>]}
<lconditions> : :=
<condition attribut> {,<condition attribut>}*
<condition attribut> : :=
<attr> <condition> <expression>
```

Une variable est un identificateur commençant par ?. Il est utilisé pour faciliter la description des règles. Dans une règle, toutes les occurrences d'une même variable représentent la même entité. Cette entité est définie par instantiation au cours de l'évaluation des règles.

<spécif ensemble> est une expression dont l'évaluation fournit une liste éventuellement vide contenant les noms des objets qui satisfont aux conditions indiquées.

Le mot clé ***in*** signifie que la variable associée au filtre est une **variable-élément**. L'évaluation d'un filtre contenant une variable-élément se traduit par une duplication de la règle en cours d'interprétation en autant d'instances qu'il y a d'éléments dans l'ensemble défini par le filtre. Si l'ensemble est vide, la règle n'est pas tirable et l'interprétation est interrompue. Si l'ensemble n'est pas vide, l'interprétation continue avec chacune des instances nouvelles crées. Dans chacune de ces instances, toutes les occurrences de la variable-élément sont remplacées par un élément de l'ensemble associé au filtre.

Le mot clé ***is*** signifie que la variable associée est une **variable-ensemble**. L'instantiation d'une variable-ensemble ne se traduit pas par une duplication de règle comme précédemment ; plûtot, toutes les occurrences de cette variable dans la règle sont remplacées par le résultat de l'évaluation de *<spécif*

ensemble>. Comme précédemment, si cet ensemble est vide, l'interprétation de la règle échoue.

la barre **/** introduit une liste de conditions *<lconditions>* servant à caractériser l'ensemble des objets visés.

Exemples :

```
?node in (node/cycle>3,operation=add)
?lnodes is (node in successors(node node1)/cycle=null)
```

Le premier exemple décrit l'ensemble des objets de la classe node dont l'attribut cycle est supérieur à 3 et l'attribut opération égal à add. Le deuxième sélectionne l'ensemble des objets de la classe node qui sont successeurs de l'objet *(node node1)* dont l'attribut cycle n'a pas de valeur.

En annexe A, on trouvera une description détaillée des filtres.

Au sein de la base de connaissances, les règles se répartissent en trois catégories distinctes :

—les théorèmes,

—les opérateurs,

—les démons

3.2.2.2. Les théoremes

Les théorèmes sont des artifices permettant de définir des concepts et des propriétés facilitant la lisibilité et l'expression des parties gauche de règles. Un théorème est une règle dont la partie droite (encore appelée propriété ou formule) est un prédicat caractérisé par :

—un identificateur : le nom du prédicat ;

—la liste de ses arguments (paramètres).

La règle est vue comme une condition suffisante (mais non nécessaire) à la validité de la propriété. Plusieurs théorèmes peuvent porter sur une même propriété.

Le système considère qu'une propriété est vérifiée si il existe un théorème actif dont le prédicat est égal à la propriété à vérifier.

Pendant l'interprétation d'une règle, l'occurrence d'une propriété dans la partie gauche de la règle permet d'initier un cycle de chaînage arrière guidé par la propriété. C'est-à-dire que le système cherche une règle active dans l'ensemble des théorèmes portant sur la propriété à établir. Dans le cas où aucune règle active n'est trouvée permettant de déterminer la valeur (vraie ou

fausse) de la propriété à établir, cette valeur est considérée comme inconnue. Ainsi donc, le système distingue trois valeurs pour une formule :

—vraie si il existe un théorème valide dont la conclusion affirme la véracité de la formule ;

—fausse si il existe un théorème concluant à une proposition contredisant la formule (si *<formule>* est une formule fausse, ***not*** *<formule>* est vraie et réciproquement) ;

—inconnue (***unknown***) si aucun théorème ne permet de conclure à l'une ou l'autre des valeurs de la formule.

Loin de constituer une limitation du système, ceci est un avantage puisqu'on peut définir une propriété comme vraie ou fausse par défaut : si *<formule>* est une formule dont la valeur est inconnue, la formule ***unknown*** *<formule>* est vraie ; dans les deux cas précités cette formule est fausse ;

On peut, par exemple, définir la propriété de commutativité des opérations arithmétiques par :

—une opération est commutative si c'est soit une addition (*plus*), soit une multiplication (*mult*), soit une conjonction (*and*), soit une disjonction (*or*). Dans le cas contraire, l'opération n'est pas commutative :

```
if ?ope1 in ( operation in '(add mult and or) )
conclude
commutative(operation ?ope1)
end
if unknown commutative(operation ?ope1)
conclude
not commutative(operation ?ope1)
end
```

qu'on peut lire : si une operation appartient à l'ensemble *(add, mult, or, and)*, elle est commutative ; sinon elle ne l'est pas.

L'usage des théorème permet d'alléger les parties gauches des règles et aussi de réduire considérablement le nombre de règles à écrire. Par exemple les règles :

```
if h1 & h2 & h3 then A1
if h4 then A1
```

```
if h9 then A1
if h7&h8 then A1
if h1&h2&h3 &h5 then a2
if h4 & h5 then a2
if h7&h8&h5 then a2
if h9 &h5then a2
```

peuvent être remplacées par :

```
if h1&h2&h3 conclude c1
if h4 then conclude c1
if h9 conclude c1
if h7&h8 conclude c1
if c1 then a1
if c1 &h5 then a2
```

en fait, tout ensemble de règles de la forme

```
if h1&h2&h3 then a4
if h4 then a4
if h9 then a4
if h7&h8 then a4
```

peut s'exprimer plus simplement par :

```
if c1 then a4
```

3.2.2.3. les operateurs

Les opérateurs sont soit des actions primitives, soit des règles exprimant des stratégies à suivre dans l'accomplissement d'une tâche complexe.

3.2.2.4. Les actions primitives

Dans le cas où l'opérateur est une primitive, il est :

—soit une action prédéfinie du système (création, modification d'objets de la base cf paragraphe précédent) ;

—soit une procédure fournie par l'utilisateur et gérée par le système.

Par exemple, ASA utilise une procédure ADA pour effectuer l'analyse syntaxique du fichier source contenant la description fonctionnelle du circuit à réaliser. Cette procédure correspond à l'une des entrées d'une instruction à choix multiple ADA exécutée par le moniteur de routines. Cette procédure est mise à la disposition du moteur d'inférence par la définition suivante :

```
to_do parse(file ?file)
call_routine 5 end
```

qui associe l'entrée 5 du moniteur de routines à l'action désignée par *(parse(file ?file))*.

Signalons que la procédure correspondante (*parse*) doit préalablement à son utilisation être compilée et intégrée au moniteur de routines. Pour écrire cette procédure, ODSE définit toute une interface permettant de modifier par programme la base des faits sous le contrôle du moteur d'inférence.

3.2.2.5. Les opérateurs complexes

Un opérateur complexe est caractérisé par une tâche et une règle.

La tâche ou format de l'opérateur comprend :

—le nom de l'opérateur ;

—la liste de ses arguments (paramètres).

La règle, dont la partie gauche est encore appelée précondition, est définie par une formule du style :

```
todo tache(arg1... agrn)
if précondition
do
     act1(argi..argj)
     ...
     actm(argk..arg1)
end
```

La règle peut être comprise de la manière suivante : pour effectuer la tâche définie en en-tête, si la précondition est satisfaite, on peut suivre la démarche indiquée (partie droite de la règle).

La partie gauche exprime les conditions que doit vérifier la base de faits pour que l'application de la partie droite permette d'accomplir la tâche visée, d'où le terme précondition.

L'ensemble des opérateurs complexes ayant un format identique définit ce que nous appellerons une "action complexe". Un opérateur complexe constitue donc une manière parmi d'autres d'accomplir une action complexe. Dès lors on pourra parler de l'occurrence d'une action complexe en partie droite de règle chaque fois que l'on y trouve une action correspondant au format d'un opérateur complexe.

L'exécution d'une action complexe, est faite en deux étapes :

—détermination de l'ensemble de tous les opérateurs activables dont l'en-tête est identique à l'action complexe

—sélection et interprétation de la règle la plus spécifique : celle dont la précondition est la plus chargée.

Pour fixer les idées, prenons l'exemple de l'opérateur ODSE *allocate_fu* qui alloue les ressources fonctionnelles aux opérations du circuit dans ASA :

```
(allocate_fu(node ?node))
```

alloue au noeud *?node*, une unité fonctionnelle permettant d'exécuter l'opération définie par ce noeud. Cet opérateur est invoqué dans ASA après que les noeuds aient été ordonnancés.

En général, pour effectuer cette tâche, il faut trouver une ressource fonctionnelle disponible pendant le cycle d'ordonnancement du noeud *(node ?node)*, ce que l'on exprime par :

```
to_do allocate_fu(node ?node)
if ?lres is ( resource/ category=f_unit,
 operation=[(node ?node)operation],
 available<[(node ?node)cycle])
do
 (allocate(resource (first '?lres))(node ?node))
end
```

L'attribut *available* désigne le moment à partir duquel une ressource fonctionnelle est disponible. L'opérateur ci-dessus propose de sélectionner l'ensemble (*?lres*) des ressources fonctionnelles pouvant effectuer l'opération du nœud (*operation=[(node ?node)operation]*) et disponibles avant le cycle d'exécution du nœud *(available<[(node ?node)cycle])*. La première ressource fonctionnelle ainsi trouvée est allouée au nœud *(node ?node)*. Or une telle

solution, quoique générale, ne donne pas toujours des résultats satisfaisant puisqu'on ne fait aucune distinction entre les ressources fonctionnelles disponibles : on en prends une au hasard (la première par exemple) et on l'affecte au noeud. Il semble souhaitable de préférer les ressources qui sont déja reliées aux élements de mémorisation contenant les opérandes. On tient compte de ce fait en ajoutant à la base de connaissances des règles selon le modèle suivant :

```
to_do allocate_fu(node ?node)
if ?lres is { resource/ category=f_unit,
     operation=[(node ?node)operation],
     available<[(node ?node)cycle],
     input1=[(node ?node)input1],
     input2=[(node ?node)input2]}
do
     (allocate(resource (first '?lres))(node ?node))
end
```

ici on alloue un opérateur déjà relié aux opérandes du nœud ;

```
to_do allocate_fu(node ?node)
if commutative(operation [(node ?node)operation]
     ?lres is { resource/ category=f_unit,
     operation=[(node ?node)operation],
     available<[(node ?node)cycle],
     input1=[(node ?node)input12,
     input2=[(node ?node)input1]}
do
     (allocate(resource (first '?lres))(node ?node))
end
```

ici on exploite la commutativité de l'opération pour sélectionner un opérateur dont les entrées sont reliées aux opérandes dans un ordre inversé (entrée 1 opérande 2 et entrée 2 opérande 1) ;

```
to_do allocate_fu(node ?node)
if ?lres is { resource/ category=f_unit,
 operation=[(node ?node)operation],
 available<[(node ?node)cycle],
 input1=[(node ?node)input1]}
```

```
do
  (allocate(resource (first '?1res))(node ?node))
end
```

cet opérateur complexe alloue une unité fonctionnelle dont l'entrée 1 est relié à l'opérande 1 du nœud.

Pour une situations donnée, plusieurs opérateurs peuvent être actifs. Le choix de ODSE se porte par défaut sur l'opérateur dont la précondition est la plus contrainte. Le choix d'une ressource fonctionnelle se portera donc en priorité sur celle qui a le plus de connexions avec les opérandes du noeud concerné (cf chapitre 4).

3.2.4.3. Les macro-operateurs

Afin d'accroître la puissance de résolution de problèmes de ODSE, un certain nombre de macro-actions peuvent être définies, suivant une syntaxe inspirée de l'expression des macros en LISP (mapcar,mapc...).

Dans le cadre de cette thèse nous ne nous étendrons pas plus sur ce sujet puisque nous n'en n'avons pas eu besoin pour réaliser ASA.

3.2.2.6. Les démons

Les démons sont des règles qui sont interprétées à l'occurrence de certains événements du système (modification d'un objet, création ou destruction d'un objet, exceptions levées). Un démon est donc une règle dont la première condition exprime un évènement (ou déclencheur) du système.

Cet événement peut être :

- la création d'un objet ou sa modification
- une exception levée par une action

exemple :

```
when modified asap_cycle(node ?node1)
if ?node in (node in successors(node ?node1)
do
     (add_constraint(node  ?node)
     (asap_cycle >= [(node ?node)asap_cycle]+
     [(famille[(node ?node)operation])nbcycles]))
     (asap_schedule(node ?node))
end
```

est un démon qui assure la propagation des contraintes d'ordonnancement des noeuds (*node*) décrivant les opérations d'un graphe acyclique. Chaque fois

qu'un noeud est affecté à un cycle donné, ses successeurs doivent attendre la fin de son exécution avant de s'exécuter à leur tour. Cela se traduit par une contrainte d'ordonnancement sur l'attribut *asap_cycle* des noeuds successeurs. L'action *asap_schedule* ordonnance les noeuds suivant les dates au plus tôt (cf chapitre 2 ou 4).

Les démons permettent de maintenir la cohérence de la base de faits en propageant les modifications au fur et à mesure qu'elles interviennent dans la base. De ce fait, les autres règles (opérateurs) n'ont pas à se préocuper de la mise à jour des relations existant entre les objets de la base.

3.2.2.7. Le traitement des exceptions

Les démons fournissent un moyen de traiter les exceptions qui peuvent survenir au cours de la résolution d'un problème(cf annexe A pour la syntaxe des catégories d'interruption envisagées). Par exemple

```
when not_found(node ?node)
do
 (make(node ?node))
 (retry)
end
```

est un démon qui récupère les exceptions qui surviennent lorsqu'une exception signale l'absence d'un objet de la classe node dans la base. Le traitement proposé consiste à créer un objet et à relancer l'action primitive défaillante.

Lorsqu'un démon actif est trouvé pour traiter une exception, le contrôle est passé au démon qui définit la marche à suivre. ODSE offre un ensemble de directives pour exprimer cette marche à suivre :

- —***retry*** ordonne la reprise normale du traitement précédant l'exception à partir de la primitive qui l'a suscitée ;
- —***resume*** ordonne la reprise du traitement à partir de l'action suivant la primitive qui a suscité l'exception.

Lorsque aucun démon actif n'a été trouvé pour traiter une exception, l'action en cours se termine par un échec qui arrête le moteur d'inférence. Ces échecs sont symptomatiques d'une défaillance dans la base des connaissances. La résolution de problème est envisagée dans ODSE comme un processus monotonique, ne tolérant pas les erreurs. Ce point sera précisé à la section suivante.

3.2.3. conclusion

Un système expert sera obtenu en :

—définissant l'ensemble des objets modélisant le problème à résoudre ;

—garantissant le maintien des relations qui doivent exister entre ces objets par l'écriture de démons qui rétabliront ces relations lors de toute modification de la base des faits ;

—définissant les opérateurs qui permettront au moteur de résoudre le problème initial.

Les directives de définition de classes d'objets, les commandes de construction de la base de faits ainsi que les règles, peuvent être écrites dans des fichiers texte.

La résolution d'un problème est invoquée par la commande :

(<operateur>{(<class1><objet1>)})!

<operateur> définit le problème à résoudre. En général, il correspond à un opérateur complexe.

{(<class1><objet1>)} est la liste des arguments.

Par exemple, pour construire l'architecture d'un circuit intégré à l'aide de ASA, on lance la commande :

(build(chip <nomcirc>))!

dans laquelle *<nomcirc>.SPC* correspond au fichier contenant la description fonctionnelle du circuit à réaliser. *build* est l'opérateur complexe définissant la tâche de synthèse architecturale.

L'interface extérieur offre aussi quelques commandes de mise au point facilitant l'élaboration des systèmes expert.

3.3. Les mécanismes d'inférence dans ODSE

Un système expert construit à l'aide de ODSE contient une base de faits et une base de règles. La base de règles contient plusieurs catégories de règles. L'action du système expert est conduite par le moteur d'inférence qui exploite les diverses catégories de règles pour construire une solution au problème modélisé dans la base de faits. Comment ce moteur coordonne-t-il les diverses

catégories de règles pour résoudre un problème donné ? Les bases de connaissances pouvant contenir de grandes quantités de règles, quelles sont les solutions pourvues par ODSE pour exploiter efficacement la base de règle et la base de faits ?

La résolution de problème est traitée au §3.3.1 ; l'efficacité de l'exploitation des bases de faits et de connaissances est obtenue grâce à des techniques d'indexation (§3.3.2) et à des techniques de compilation (§ 3.3.3).

3.3.1. La résolution de problème

La résolution d'un problème donné est obtenue par application d'une séquence appropriée d'opérateurs primitifs qui construisent progressivement la solution dans la base de faits. Le rôle du moteur d'inférence est de chercher cette séquence appropriée d'actions élémentaires. Pour ce faire il dispose d'un ensemble de connaissances opératoires exprimées sous la forme de squelettes de plans. Comme nous l'avons vu, chacun de ces squelettes exprime des conditions dans lesquelles un problème donné peut-être décomposé en une séquence de sous-problèmes. Partant du problème initial, le moteur scrute son catalogue de plans à la recherche de ceux parmis ces plans qui permettent de décomposer le problème initial en sous-problèmes compte tenu des particularités décrites dans la base des faits. Si aucun opérateur n'est trouvé dont les préconditions sont vérifies par la base des faits, le problème initial est en dehors des compétences du SE. Un aveu d'impuissance est envoyé à l'utilisateur pour rapporter ce fait. Ces situations sont symptomatiques d'une limitation inhérente à la base des connaissances. Elles arrivent chaque fois que le problème soumis au SE. présente des aspects et des conditions qui n'avaient pas été envisagés au moment de l'élaboration de la base de connaissances. Ces difficultés peuvent être surmontées en ajoutant dans la base de connaissances des opérateurs supplémentaires prenant en compte les spécificités nouvelles rencontrées. Les nouveaux opérateurs ne remettent pas en question les anciens, ils les complètent en permettant au SE de résoudre une classe élargie de problèmes. Ces extensions de la base de connaissances qui surviennent à l'encontre de nouveaux problèmes produisent une croissance progressive des compétences d'un SE.

Si des opérateurs sont trouvés qui permettent de réduire le problème initial, ODSE sélectionne l'opérateur le plus spécifique, c'est-à-dire celui dont les préconditions sont les plus sévères. L'heuristique d'un tel choix est qu'une solution est d'autant plus raffinée qu'elle prend en compte un grand nombre de

facteurs propres au problème à résoudre. Une fois la sélection effectuée, ODSE remplace le problème initial par la séquence de sous-problèmes préconisée par l'opérateur choisi et le cycle de résolution est repris pour chacun des sous-problèmes de la séquence et suivant l'ordre de ces sous-problèmes dans la séquence.

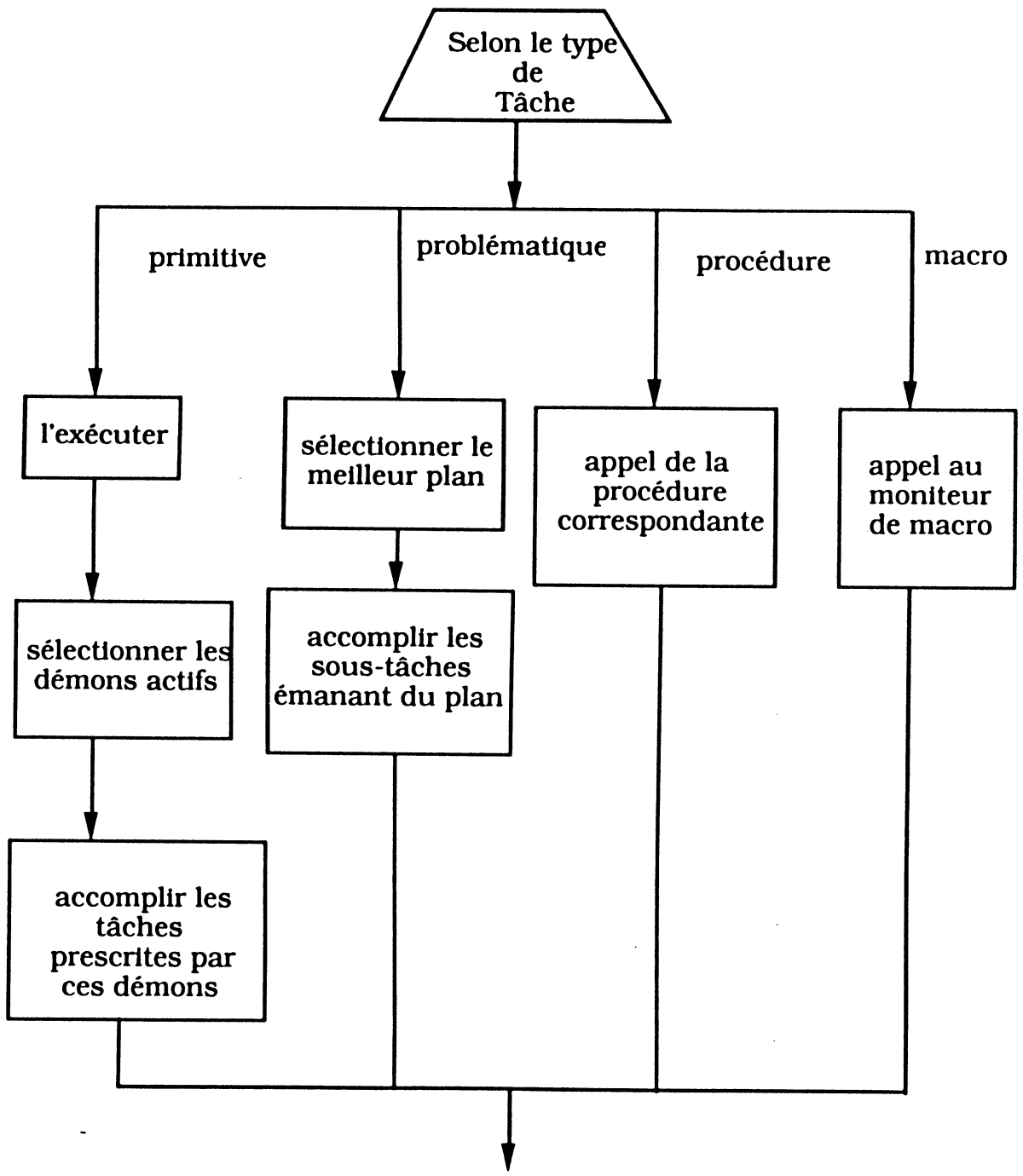


Figure 5 comment ODSE accomplit une tâche

Lorsqu'un opérateur primitif est rencontré, il est exécuté aussitôt. Cette exécution initie un cycle d'interprétation des démons en **chaînage avant** qui ne s'arrête que lorsque aucun démon actif n'est trouvé dans la base. Au début de ce cycle, ODSE filtre l'ensemble des démons pour déterminer ceux qui sont actifs. Ensuite, les démons actifs sont tirés. L'interprétation d'un démon peut provoquer l'exécution d'une séquence d'opérateurs entraînant à leur tour l'interprétation d'autres démons...

Lorsqu'une macro est rencontrée, la stratégie de résolution de problème est prise en charge par le moniteur de macro.

Pendant l'évaluation des règles (opérateurs ou démons), un prédicat en partie gauche de règle est considérée comme une propriété à démontrer qui engage le moteur dans une recherche en **chaînage arrière** de l'ensemble des théorèmes portant sur le prédicat. La recherche se termine lorsqu'un théorème actif permet de décider de la valeur de vérité de la propriété ou lorsqu'aucun théorème actif n'est rencontré. Dans ce dernier cas, la propriété a une valeur inconnue. Notons deux différences essentielles avec les démonstrateurs automatique de théorèmes comme PROLOG : il ne s'agit pas ici de trouver toutes les instantiations qui vérifient le prédicat initial puisqu'on s'arrête dès qu'un théorème actif est trouvé ; d'autre part on considère trois valeurs de vérité possibles (vrai, faux et inconnu) alors qu'en PROLOG, toute proposition est fausse toutes les fois qu'on ne peut prouver le contraire.

La figure 5 résume la stratégie de résolution de problème mise en oeuvre dans ODSE. Cette stratégie est très dépendante des connaissances de la base de règles. La recherche d'une solution passe par la consultation d'une grande base de données contenant les règles. On imagine bien la nécessité de réduire le temps d'accès à cette base. Cela est accompli d'une part en structurant les informations suivant leur contenu pour accélérer les accès associatifs et d'autre part en organisant les règles suivant un réseau de graphes accélérant le filtrage et le rendant pratiquement indépendant du nombre de règles. La première technique est connue sous le nom d'**indexation** et la deuxième sous celui de **compilation**.

3.3.2. Indexation de la base de faits

Les filtres utilisés dans les parties gauches des règles définissent des ensembles d'objets en compréhension : l'ensemble est défini par les propriétés caractéristique de ses éléments. Les propriétés sont en général des conditions que doivent vérifier les attributs des objets appartenant à l'ensemble.

L'évaluation d'un filtre consiste à sélectionner, dans la base, les objets définis par le filtre. Une implémentation naïve de la base de faits exigerait un parcours systématique des objets de la base pour déterminer l'extension d'un filtre donné.

ODSE propose une meilleure organisation en regroupant les objets suivant leur classe et en indexant les objets d'une même classe.

Les objets sont regroupés en collections suivant leur classe. De ce fait, on se limite au cours de l'évaluation d'un filtre à un sous-ensemble de la base de faits correspondant à la classe (commune) des objets définis par le filtre. Au sein d'une classe donnée, plusieurs listes sont constituées suivant les différentes valeurs des attributs des objets. Une liste repère l'ensemble des objets de la collection ayant la même valeur pour un attribut donné (figure 7). Les listes sont mises à jour au gré des modifications des valeurs des attributs. Cette indexation tend à rendre l'évaluation des filtres indépendante du nombre d'objets de la collection. En effet l'évaluation d'un filtre est obtenue par une séquence d'intersections de listes. Chaque condition du filtre correspond à une liste. les objets définis par le filtre doivent appartenir à chacune des listes correspondant aux conditions du filtre.

L'opération d'intersection de deux listes est facilitée par la représentation unique des symboles. Puisque chaque objet porte un nom, les listes sont construites à l'aide de pointeurs vers les symboles uniques représentant les noms des objets de la liste. L'intersection de deux listes est calculée par un algorithme "MARK-AND-TEST" de complexité linéaire par rapport à la somme des tailles des deux listes. Il consiste à marquer les éléments de la première liste et à parcourir la deuxième en ne retenant que les éléments marqués.

L'évaluation d'un filtre est dépendante du nombre de conditions dans le filtre et de la dispersion des objets suivant les valeurs des attributs. L'avantage obtenu par l'efficacité du filtrage peut être remis en cause par la taille de l'ensemble des règles à filtrer. Ici aussi, il convient de réduire la taille de cet ensemble de règles.

3.3.3. Indexation de la base de règles

Comme on l'a vu pour la base des faits, les règles de la base de connaissances sont regroupées par classes : les démons, les opérateurs et les théorèmes. Les techniques d'indexation sont également utilisées pour accélérer l'accès associatif à ces règles suivant les cas :

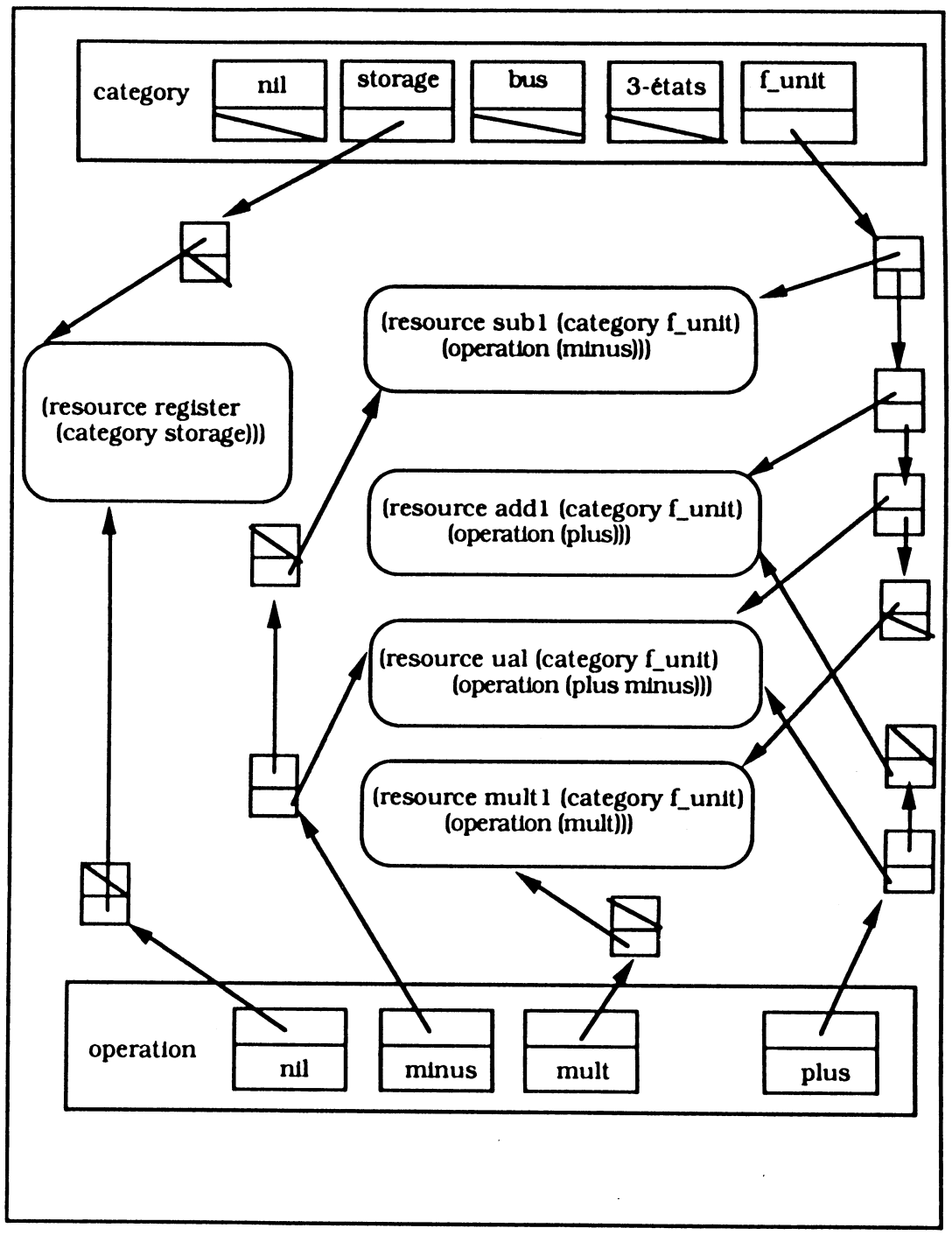


Figure 7 : indexation des objets de la classe resource

—les démons sont indexés suivant les évènements susceptibles de les activer ; tous les démons se rapportant à une même catégorie d'évènement sont chaînés dans une liste ;

—les théorèmes portant sur une même propriété sont regroupés dans une liste ;

—les règles définissant un même opérateur complexe sont regroupés en une liste ;

Cette indexation réduit considérablement l'ensemble des règles à interprêter à chaque étape de la résolution d'un problème :

—pendant la réduction d'un problème, on se limite aux opérateurs portant sur le problème à réduire ;

—pendant le chaînage avant, on se limite aux démons déclenchés par l'événement qui a commencer le cycle ;

—pendant le cycle de chaînage arrière, seuls les théorèmes ayant un rapport avec la propriété à démontrer sont pris en considération.

L'évaluation des règles est considérablement allégée par les techniques d'indexation qui assurent un filtrage efficace de la base de faits et réduisent notablement l'ensemble des règles à explorer à chaque étape de la résolution de problème. La compilation de la base fournit des résultats meilleurs en rendant l'interprétation d'un ensemble de règles indépendante du nombre de règles.

3.3.4. Compilation de la base de règles

Les ensembles de règles explorés par ODSE au cours de la résolution de problème sont de trois catégories possibles.

La première est un ensemble d'opérateurs portant sur un même problème. Cette catégorie intervient pendant la phase de réduction de problème. Il s'agit alors pour le moteur de sélectionner le "meilleur opérateur" de cet ensemble pour réduire le problème courant.

La deuxième catégorie est un ensemble de théorèmes se rapportant à une même propriété. Elle délimite l'espace de recherche pendant la détermination de la valeur de vérité d'un prédicat. Au court de cette détermination, le moteur cherche un théorème quelconque permettant d'obtenir le résultat désiré.

La dernière catégorie regroupe l'ensemble des démons déclenchés par un même événement. Elle définit l'ensemble des démons à interpréter à l'occurrence d'un évènement dans la base de faits. Pendant le cycle de chaînage avant qui suit cet événement, le moteur cherche l'ensemble de tous les démons actifs appartenant à la catégorie.

Pour chacune de ces catégories, la tâche d'interprétation effectuée par le moteur peut être définie comme un problème de recherche suivant trois critères différents selon les cas :

—recherche de la meilleure règle dans le cas des opérateurs ;

—recherche d'une règle quelconque dans le cas des théorèmes ;

—recherche de toutes les règles actives dans le cas des démons.

Dans la version courante de ODSE, la compilation n'a pas encore été implémentée. Dans la suite, nous nous contenterons de donner des indications générales sur la manière de la réaliser.

Les règles d'une même catégorie peuvent avoir des conditions en commun. Une exploration systématique d'un ensemble de règles conduit à plusieurs évaluations des conditions partagées par plusieurs règles. Pour éviter cette duplication de travail, les règles d'une même catégorie sont organisées en réseau.

Ce réseau est un graphe qui décrit les parties gauches des règles d'une même catégorie par des chemins. Les noeuds du graphe correspondent aux diverses conditions intervenant dans les règles. Les arcs traduisent l'enchaînement des conditions au sein des parties gauches des règles. Le graphe est construit de manière à factoriser au maximum les conditions. C'est-à-dire que deux règles ayant des conditions communes seront décrites par des chemins qui se recouvrent.

A chaque noeud du graphe, on associe la liste de toutes les règles dont les chemins passent par le noeuds. Toutes ces règles contiennent en partie gauche, la condition décrite par le noeud.

Chaque graphe est caractérisé par un noeud initial, point de départ de tous les chemins correspondant aux règles décrites par le graphe. A chaque règle correspond un noeud final dans le graphe. Le chemin à travers le graphe partant du noeud initial au noeud final décrit exactement la séquence de conditions correspondant à la règle.

Le graphe est progressivement construit au fur et à mesure que la base de règles est construite. Toute nouvelle règle est ajoutée au réseau décrivant sa catégorie. Ce processus est appelé compilation.

L'interprétation d'une règle à l'aide d'un graphe se ramène à un parcours de chemin dans le graphe. Les trois aspects du problème de filtrage d'un ensemble de règles que nous avons évoqués précédemment trouvent une solution acceptable lorsque les règles sont compilées en un graphe.

3.3.4.1. Recherche du meilleur opérateur

Partant d'un ensemble d'opérateurs traitant d'un problème particulier, il s'agit de sélectionner l'opérateur actif dont la précondition est la plus sévère. Transposé dans le graphe, ce problème revient à la recherche du plus long chemin possible.

Pour traiter ce problème, on associe à chaque chemin possible, une estimation de sa longueur. Cette estimation peut être faite statiquement au moment de la construction du chemin correspondant. Elle est fonction du

nombre de conditions élémentaires intervenant dans les préconditions des opérateurs. Les chemins ainsi pondérés peuvent être classés suivant les poids décroissants. Ce classement, également effectué pendant la phase de compilation, fournit un ordre d'évaluation de l'ensemble des règles.

La recherche du meilleur opérateur revient à explorer systématiquement les chemins du graphe suivant leur classement. L'exploration s'arrête lorsque toutes les conditions d'un chemin sont vérifiées. L'opérateur correspondant au chemin est retourné comme meilleur opérateur. Lorsqu'une condition fausse est rencontrée, on élague l'espace de recherche en supprimant de l'ensemble des opérateurs en considération, ceux qui contiennent cette condition en partie gauche. D'autre part, on évite de réévaluer les conditions en conservant les résultats des évaluations déjà obtenues.

3.3.4.2. Evaluation d'une propriété

L'évaluation d'une propriété se ramène à la recherche d'un théorème actif portant sur cette propriété. Il parait judicieux de commencer par évaluer les théorèmes dont les conditions sont les plus simples avant de s'attaquer le cas échéant aux théorèmes compliqués. Les chemins du graphe étant classés par longueur croissante, l'évaluation d'une propriété se ramène à la recherche du plus court chemin possible. L'heuristique sous-jacente à une telle stratégie est de dépenser le moins d'énergie possible pour établir la valeur de vérité d'une propriété.

3.3.4.3. Evaluation d'un ensemble de démons

L'évaluation d'un ensemble de démons diffère des deux cas que nous venons de traiter. Ici, ODSE doit déterminer l'ensemble de tous les démons actifs. A l'aide du graphe, il s'agit de rechercher tous les chemins possibles.

Plutôt que de classer les chemins suivant leur longueur, il paraît plus astucieux de pondérer les noeuds suivant la taille des listes de règles qui leur sont associées. L'évaluation de l'ensemble des règles se ramène à un parcours de l'ensemble du graphe suivant la stratégie suivante : les divers chemins sont explorés en parallèle. Le chemin dont le noeud courant a le poids le plus élevé est exploré en premier. On espère ainsi considérer les conditions dont dépendent un grand nombre de règles avant les autres. En cas d'échec, ces conditions réduisent considérablement l'ensemble des démons à évaluer.

3.3.4.4. Conclusion

Dans les trois cas que nous avons présentés, le problème de filtrage d'un ensemble de règles est formalisé en un problème de parcours de graphe suivant des critères définis statiquement au moment de la compilation de la base de règle. Cette compilation partitionne les trois classes de règles (démons, opérateurs, théorèmes) en graphes rendant plus efficace le problème de filtrage.

3.4. Implémentation du prototype

Comme nous l'avons signalé à l'introction de ce chapitre, ODSE a été développé à l'aide du langage ADA suivant une méthode dite de conception orientée objets.

L'application de cette démarche (cf §3.1.4.3) à la conception d'un générateur de système experts nous a conduit à identifier au niveau le plus élevé trois abstractions :

—la base des faits qui contient les objets et doit offrir un moyen d'agir sur ces objets suivant les opérations élémentaires vues au § 3.2. De plus il doit permettre un accès associatif aux objets ;

—la base des règles qui contient les règles et doit offrir des opérations permettant interpréter les différentes catégories de règles. Comme la base des faits, il doit permettre un accès associatif efficace des règles. Pour interpréter les règles, elle doit pouvoir accéder aux opérations de la base des faits ;

—l'interface générale du système. C'est le programme principal. Il reconnait les commandes de l'utilisateur et les exécute. Il "voit" donc les deux abstractions précédentes.

Une analyse plus fine de la base de règles et la base de faits permet d'identifier d'autres abstractions (figure 8) :

—un fait : entité élémentaire de la base de faits caractérisée par un ensemble d'attributs et un ensemble d'actions primitives ;

—une règle : entité élémentaire de la base de règles ayant des formes multiples (démon, opérateur, théorème).

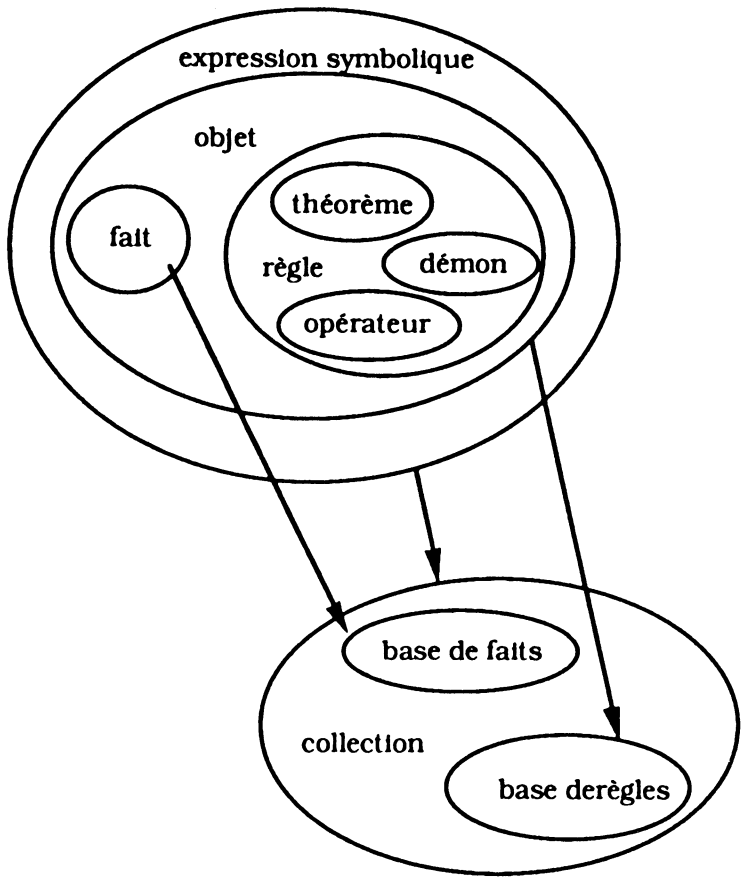


Figure 8 les abstractions de ODSE

Une analyse encore plus fine de la notion de règle permet de le voir comme un objet particulier. D'autre part l'accès associatif, suivant l'analyse que nous avons présentée au paragraphe précédent permet d'isoler une abstraction supplémentaire : la collection. La base des faits et la base des règles seront construits en utilisant des collections.

Dans le prototype, nous n'avons pas inclu les procédures permettant la compilation des règles. Elles sont écrites mais pas encore déverminées.

Le résultat global de ce découpage est schématisé à la figure 9. En annexe nous donnons la description complète des spécifications de quelques paquetages.

3.5. conclusion

ODSE est un réducteur de problème qui représente les connaissances opérationnelles par un ensemble de squelettes de plans. Cette représentation est adaptée à la représentation de l'experience dans un domaine d'expertise donné. ODSE a été écrit avec environ 10000 lignes de code ADA compte non tenu des

codes correspondant aux composants logiciels utilisés. L'exécutable occupe environ 500 K octets de mémoire.

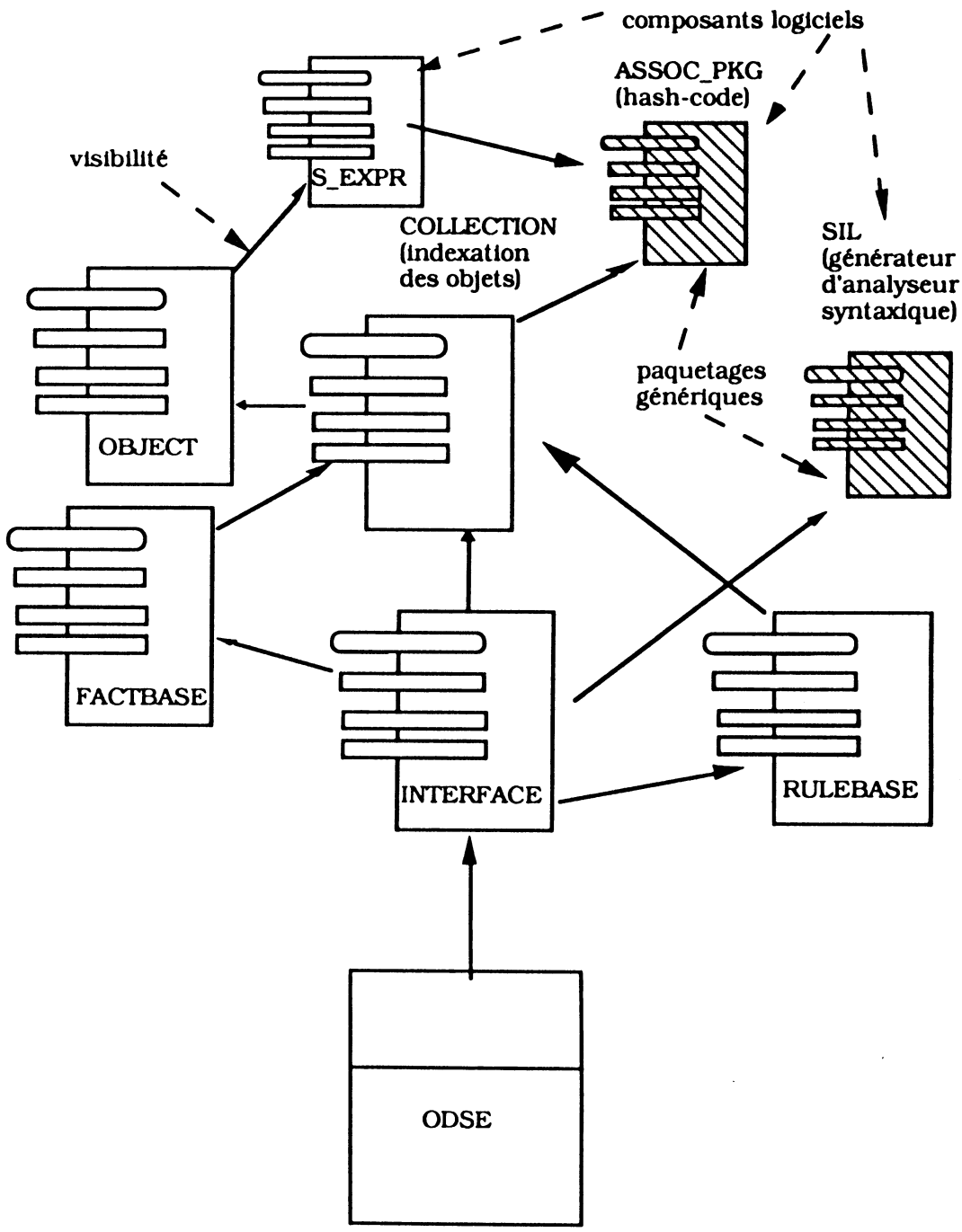


Figure 9 : les paquetages de ODSE

4. Le système expert d'aide à la synthèse architecturale de C.I. VLSI

ASA est un outil d'aide à la synthèse architecturale d'un circuit intégré à partir d'une description fonctionnelle. Il permet au concepteur d'explorer un ensemble d'architectures pour une même spécification fonctionnelle donnée, par le simple jeu de renforcement ou de relâchement de contraintes. Pour un jeu de contraintes déterminé, il s'efforce de proposer une architecture optimale au sens défini par ces contraintes, dans les limites imposées par la bibliothèque d'opérateurs. Conçu sous la forme d'un système expert (figure 1), il est composé d'une base de faits et une base de règles selon le formalisme de ODSE. La base de faits contient une représentation interne de l'algorithme du circuit, des contraintes et critères de l'utilisateur, la description de la bibliothèque d'opérateurs, ainsi que les détails de la solution architecturale construite. La base de règles contient les connaissances opératoires qui guideront ODSE dans la recherche et la construction d'une architecture de circuit intégré conforme aux spécifications de la base de faits. Ces connaissances décomposent la tâche de synthèse architecturale en cinq phases : acquisition des données du problème, ordonnancement des opérations, allocation des unités fonctionnelles, allocation des éléments d'interconnexion et optimisation générale. Avant de présenter plus en détail le contenu de la base de règles, nous décrivons les divers éléments de la base de faits qui modélisent le problème à traiter ainsi que la solution obtenue. Dans le chapitre suivant, nous comparons notre approche à quatre autres prises dans la littérature, à savoir FACET [Tse 83], HAL [Pau 87], CHIPPE [Pan 87], et MAHA [Par 86].

4.1. Modélisation du problème

Le problème de synthèse architecturale est défini par :

—la description algorithmique du circuit ;

- les contraintes imposées au fonctionnement ou à la structure du circuit final ;

—la bibliothèque des opérateurs qui définit l'ensemble des choix possibles de ressources fonctionnelles à utiliser dans l'architecture finale.

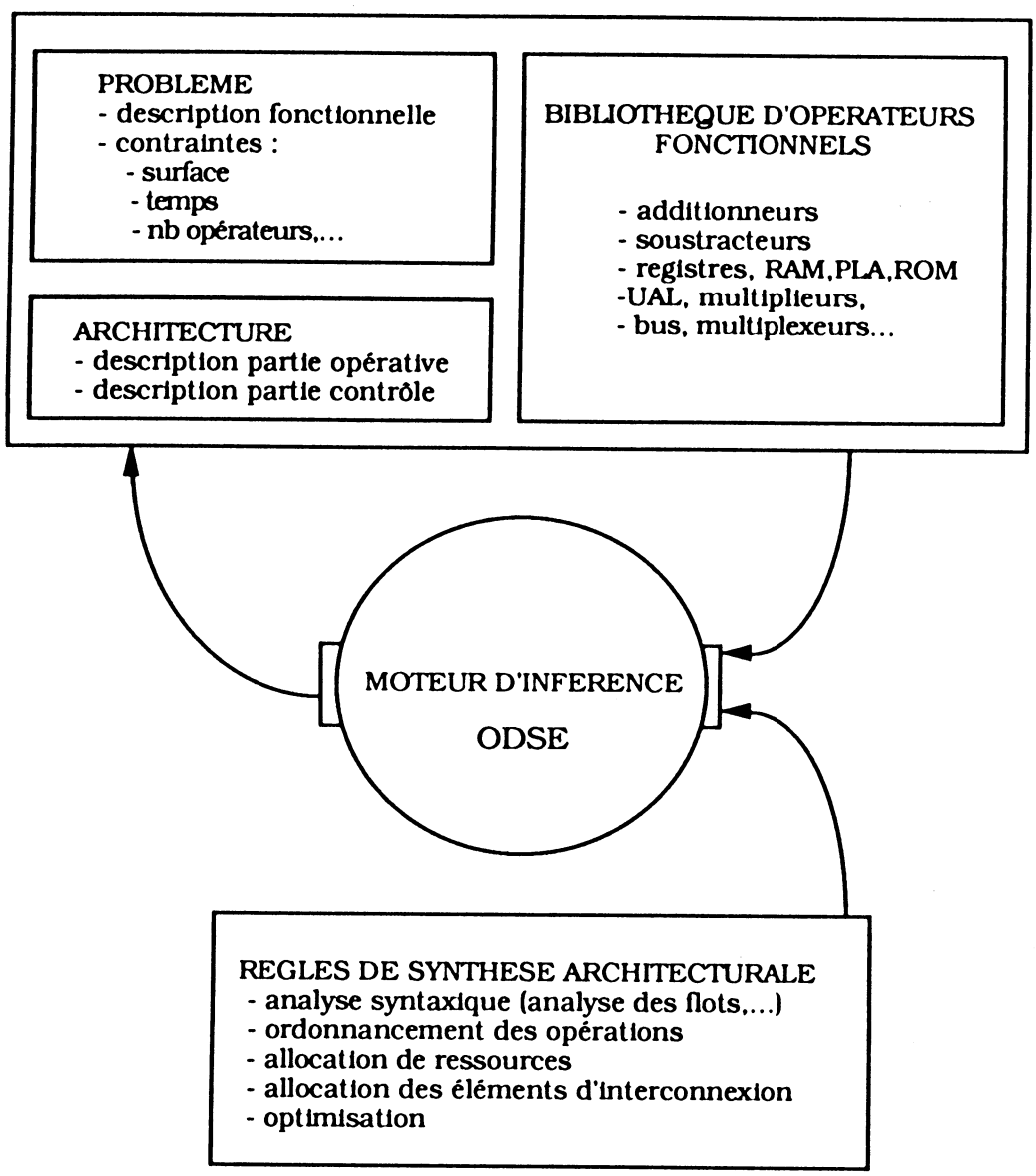


Figure 1 : le système expert ASA

La base de faits doit contenir, outre la description du problème, la représentation de l'architecture élaborée. Conformément au formalisme de ODSE, le problème ainsi que la solution retenue sont modélisés par un ensemble d'objets et de graphes. Ces objets ou graphes sont choisis en fonction de leur adéquation à faciliter l'expression des connaissances opératoires en synthèse architecturale.

4.1.1. L'algorithme du circuit

L'algorithme du circuit est fourni par l'utilisateur dans un langage de description fonctionnelle dont la syntaxe s'inspire du langage ADA (cf chapitre 2). Dans un tel algorithme, on peut isoler trois types d'informations :

CLASSE VARIABLE

ATTRIBUT	DOMAINE
name	nom de la variable
type	integer,boolean
stored	nom de la mémoire à laquelle la variable est affectée
size	taille en nombre de bits

CLASSE PARAMETRE

ATTRIBUT	DOMAINE
name	nom du paramètre
mode	in out (entrée ou sortie)
type	integer, real
affected_to	ressource allouée (port)
size	taille en nombre de bits

CLASSE CONSTANT

ATTRIBUT	DOMAINE
name	nom de la constante
type	integer , boolean
value	valeur de la constante

Figure 2 : principales caractéristiques des classes VARIABLE, PARAMETRE, CONSTANT

—les déclarations de variable et de paramètres : chaque identificateur déclaré est figuré dans la base de fait par un objet de classe variable ou paramètre. Chaque constante utilisée dans l'algorithme donne lieu à un objet de classe constante (figure 2) ;

—les blocs de base composés d'une séquence d'instructions ne comportant aucune instruction de branchement ;

—le flot de contrôle décrivant le transfert du contrôle entre les blocs séquentiels suivant la valeur de certaines conditions.

Ces deux derniers types d'informations sont modélisés dans la base de faits par deux collections d'objets : les blocs et les transitions.

4.1.1.1. les blocs de base

Un bloc de base correspond à une séquence d'instructions ne comportant aucune instruction de rupture de séquence. La figure 3 délimite les blocs correspondant à l'exemple EQUADIFF. L'effet d'un bloc est de définir un ensemble d'expressions numériques ou d'affectations. Ces expressions peuvent être utilisées par d'autres blocs pour définir de nouvelles expressions.

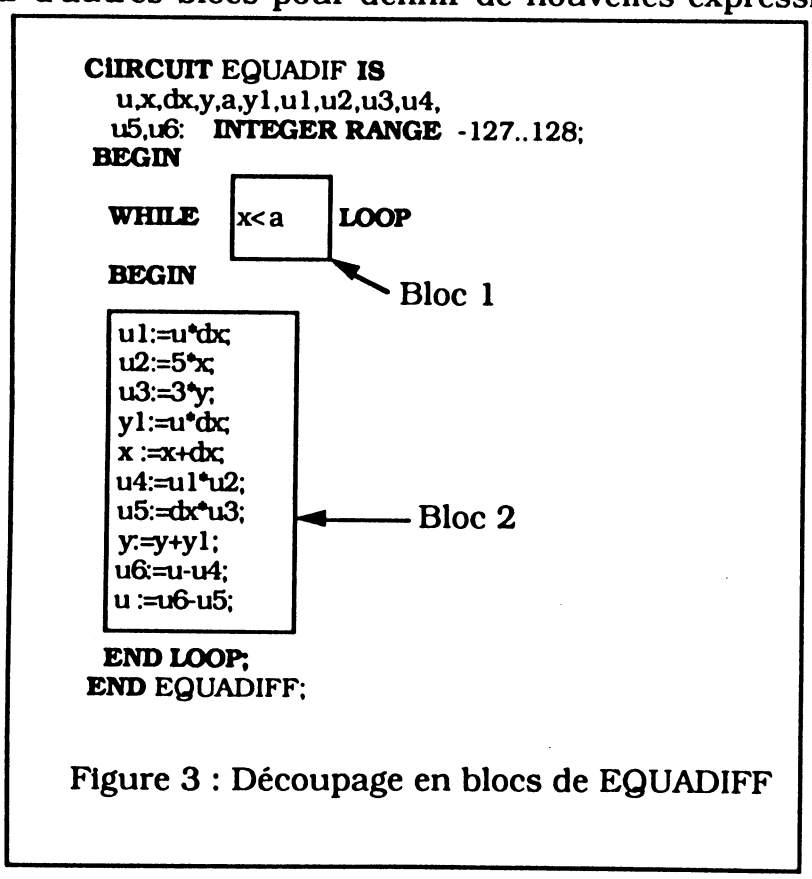


Figure 3 : Découpage en blocs de EQUADIFF

L'action d'un bloc est bien décrite par ce que AHO [Aho 84] appelle **un graphe acyclique orienté (GAO)**. Un tel graphe illustre la manière dont la valeur produite par une instruction affecte les valeurs calculées par les instructions

suivantes du même bloc. Autrement dit, il exprime les contraintes de dépendance topologique entre les diverses expressions calculées par le bloc telles qu'elles peuvent se déduire de la succession des instructions. Les graphes acycliques orientés se prêtent bien à toute une série d'optimisations sur les calculs décrits par les blocs. Nous aurons l'occasion d'y revenir lorsque nous décrirons le processus de construction de ces blocs à partir de la description fonctionnelle du circuit.

Un graphe acyclique orienté est un ensemble de nœuds liés par des relations de succession. Un nœud est formé d'une opération, de un ou deux opérandes et d'une liste d'identificateurs. Un opérande peut être une variable, une constante ou un autre nœud. La liste d'identificateurs représente les variables auxquelles sont affectées les valeurs calculées dans le bloc.

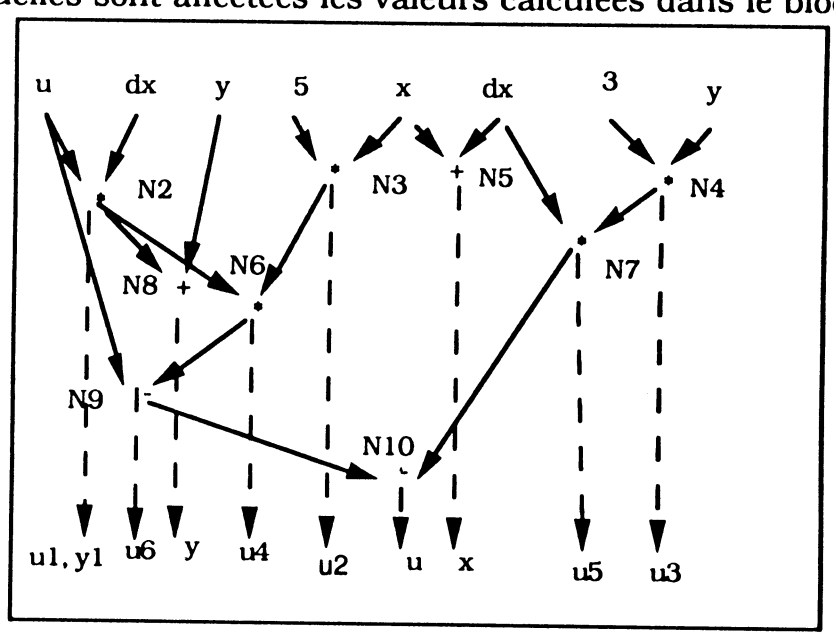


Figure 4 : graphe acyclique oriente du bloc 2

Les relations de succession entre les nœuds traduisent une dépendance entre ces nœuds : un nœud ne peut être évalué avant ses opérandes. La figure 4 décrit le graphe acyclique orienté correspondant au bloc 2. le nœud N10 ne pourra être évalué qu'après les nœuds N9 et N7.

En plus de l'ensemble des nœuds constituant le graphe orienté acyclique, les blocs de base sont également décrits par des objets de la classe BLOC dont nous indiquons les caractéristiques à la figure 5.

CLASSE NODE

VARIABLE	DOMAINE
operation	plus(+), minus(-),mult(*),div(/),shiftl,shiftr, or,and,not
type_operand1 type_operand2	node, constant, variable
operand1 operand2	nom de l'opérande
successors predecessors	liste des nœuds successeurs ou prédecesseurs
f_unit	unité fonctionnelle allouée
cycle	cycle auquel le nœud est alloué

CLASSE BLOC

VARIABLE	DOMAINE
nodes	liste des nœuds appartenant au bloc
operations	liste des opérations effectuées dans le bloc
maxcycle	cycle maximal autorisé pour l'ordonnancement
nbcycles	nombre de cycles
live_in	liste des variables vivaces en entrée du bloc
live_out	liste des variables vivaces en sortie de bloc
gen_v_e	liste des variables définies par le bloc
used_v_e	liste des variables dont le contenu a été utilisé dans le bloc

CLASSE TRANSITION

VARIABLE	DOMAINE
bloc_in	nom du bloc origine
bloc_out	nom du bloc destination
condition	nom de la condition

Figure 5 : caractéristiques principales des classes NODE, BLOC et TRANSITION

4.1.1.2. Le flot de contrôle

Le flot de contrôle décrit les enchainements des divers traitements que représentent les blocs. Ces enchaînements sont modelisés par des objets de classe *transition* et organisés en un graphe. Une transition comprend trois attributs :

—*bloc_in* représentant le bloc initial ;

-*bloc_out* représentant le bloc final ;

- une condition.

Une transition décrit un transfert de contrôle entre le bloc initial et le bloc final. Le transfert a lieu si, à la fin de l'exécution du bloc initial, la condition indiquée par la transition est vérifiée.

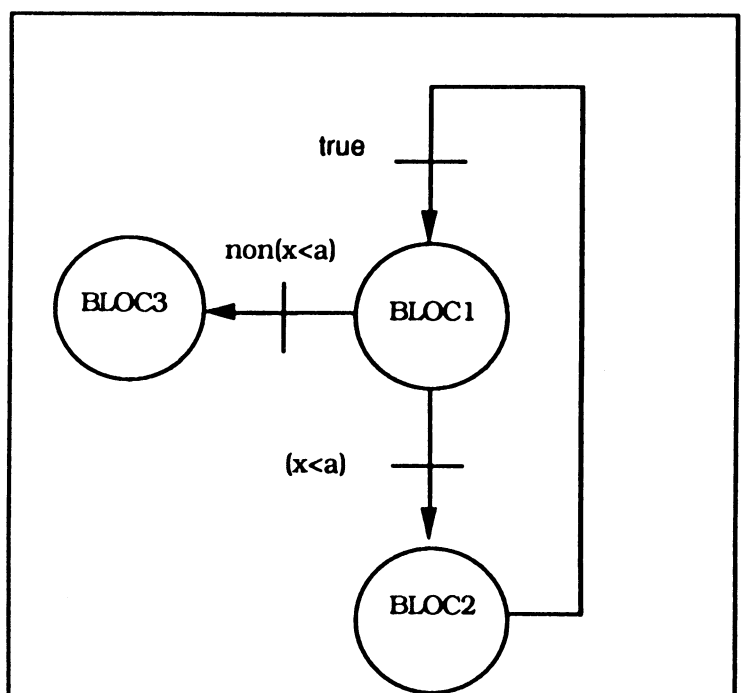


Figure 6 : flot de contrôle du circuit EQUADIFF

4.1.2. Représentation des contraintes

Une fois définies les fonctionnalités du circuit à construire, il faut recueillir les critères de l'utilisateur ainsi que ses contraintes. A cet effet, un objet de la classe *chip* représente le circuit à construire. On peut considérer cet objet comme partiellement défini au départ : il est défini par son comportement, les contraintes et les critères. La synthèse architecturale de ce point de vue essaie de compléter la définition du circuit en lui construisant une architecture. Un

objet de la classe *chip* est défini par un ensemble de blocs et de transitions décrivant son comportement, un ensemble de contraintes structurelles (nombre d'opérateurs, surface totale, vitesse...) et des critères (optimisation de surface, optimisation de temps...). Les contraintes et les critères sont décrits par des valeurs affectées à des attributs de l'objet(cf fig 7).

CLASSE CHIP

ATTRIBUT	DOMAINE
blocs	liste des blocs du circuit
transitions	liste des transitions du circuit
resources	liste des ressources constituant le circuit
operations	liste des opérations utilisées dans le circuit
cycle	durée en nanosecondes d'un cycle
constraint	surface, time
plus,minus, mult,div, shift, or, and,...	nombre d'unités fonctionnelles effectuant les opérations correspondantes dans le circuit. Ces attributs peuvent être contraint suivant les limites imposées au jeu de ressources fonctionnelles à utiliser.
time	durée maximale de fonctionnement du circuit

CLASSE FAMILY

ATTRIBUT	DOMAINE
category	storage, f_unit, bus, 3-states,multiplexer,...
operation	plus, minus, mult div, shiftl,or,and,...
surface	en microns carrés
time	en nanosecondes

Figure 7 caractéristiques principales des classes CHIP et FAMILY

4.1.3. Représentation de la bibliothèque d'opérateurs

La synthèse architecturale doit tenir compte de la disponibilité des opérateurs de la bibliothèque. On a donc besoin de se représenter les choix possibles d'opérateurs pouvant être utilisés pour la synthèse. D'autre part, cette bibliothèque est construite à l'aide d'un langage d'assemblage de cellules appelé LOF. Celui-ci offre des primitives permettant l'évaluation des caractéristiques électriques et topologiques des opérateurs. Deux choix s'offraient à nous pour la représentation de la bibliothèque d'opérateurs :

—soit on s'interfaçait directement avec LOF (SPOT) en utilisant les primitives de consultation de la bibliothèque ;

—soit on construisait dans la base de faits de ASA, une représentation de cette bibliothèque en termes d'objets.

La première solution offre l'avantage de disposer en permanence d'informations exactes sur le contenu de la bibliothèque et notament une prise en compte presque immédiate des modifications éventuelles de cette bibliothèque. Quant à la deuxième solution, elle est un peu lourde parce qu'elle implique une mise à jour de la base de faits à toute modification de la bibliothèque et présente un risque certain d'une incohérence entre les représentations de la base de faits et le contenu réel de la bibliothèque. Provisoirement, nous avons opté malgré tout pour la deuxième solution parce que cela permet de tester plus facilement ASA et de plus, la bibliothèque est encore en cours de construction.

4.1.3.1. Famille d'opérateurs

La bibliothèque d'opérateurs est donc représentée dans la base de faits par des collections d'objets. Chaque collection représente une famille d'opérateurs caractérisés par une fonction bien déterminée(addition, registre, multiplication...). Chaque modèle d'opérateur disponible en bibliothèque correspond à un objet de la classe *family*. Les caractéristiques électriques (temps de fonctionnement, consommation) et topologiques (surface, dimensions...) sont représentés par des attributs de l'objet. Un objet de la classe *family* résume les caractéristiques de l'ensemble des opérateurs correspondant à une fonction donnée. Les caractéristiques de la classe *family* sont présentées à la figure 7.

4.1.4. Représentation de l'architecture du circuit

L'architecture du circuit est représentée par une description du séquencement et un chemin de données.

4.1.4.1. Le graphe de contrôle

Pour le moment on ne s'est pas préocupé de l'architecture de la partie contrôle. On se limite à fournir la description du graphe de séquencement du circuit. Ce graphe de séquencement est modélisé par un ensemble d'objets de la classe *cycle*. Un *cycle* est caractérisé par une date et un ensemble de commandes. La date indique le top d'horloge au cours duquel les commandes seront générées.

4.1.4.2. La partie opérative

La partie opérative est un ensemble de ressources interconnectées. Chaque ressource est caractérisée par un type définissant la famille à laquelle elle appartient, des paramètres (nombre de bits, structure série, parallèle...), et l'ensemble de ses connexions aux autres ressources (cf figure 8). Les objets modélisant la partie opérative sont utilisés pour produire le fichier d'entrée de SPOT qui construit effectivement dans la base de donnée COSMIC la partie opérative en tranches du circuit.

CLASSE RESOURCE

ATTRIBUT	DOMAINE
category	storage, f_unit, multiplexer, 3-states, bus
operation	plus, minus, mult, div, shiftl, shiftr, or, and,...
surface	surface de la ressource en microns-carrés
time	temps de la ressource en nanosecondes
cycles	liste des cycles d'utilisation de la ressource
input1	ressources connectés à l'entrée 1
input2	ressources connectées à l'entrée 2
output	ressources connectées à la sortie
latches	input1, input2, output (attribut multivalué)

Figure 8 : caractéristiques principales de la classe RESOURCE

4.2. Les connaissances opératoires

Les connaissances opératoires sont exprimées par des squelettes de plans qui guident la synthèse architecturale du circuit. Cette synthèse procède en cinq étapes (cf figure 4 page 27). La première construit un modèle du problème à résoudre (création des blocs et des nœuds... décrivant le circuit à réaliser). La deuxième ordonnance les opérations (nœuds) des divers graphes acycliques issus de l'étape précédente en s'efforçant de minimiser le nombre d'opérations affectées à un même cycle tout en respectant les contraintes et les critères. Une fois effectué l'ordonnancement, l'allocation des ressources est faite avec pour souci d'en minimiser le nombre tout en respectant les contraintes de l'utilisateur et en tenant compte des disponibilités de la bibliothèque. L'optimisation est une phase qui permet, outre l'amélioration de l'architecture produite, de remettre en cause les décisions d'une étape antérieure et d'explorer ainsi plusieurs alternatives de solution. Cette démarche générale est exprimée par le plan :

```
to_do BUILD(CHIP ?CHIP) do
    (parse(CHIP ?chip))
    (schedule(chip ?chip))
    (allocate(chip ?chip))
    (optimize(chip ?chip))
    (gen_spot(chip ?chip))
end
```

En marge de ces connaissances purement opératoires, il y a des règles qui permettent de propager les contraintes d'intégrité entre différents objets de la base de faits. Par exemple les contraintes de dépendance d'un graphe acyclique induisent des contraintes de séquencement. Dans la suite, nous présentons les diverses étapes de la synthèse architecturale en indiquant à chaque fois les contraintes d'intégrités qui doivent être respectées entre les divers objets par rapport à l'étape traitée.

4.2.1. Construction des données du problème

La première étape de ASA consiste à créer dans la base de faits, un modèle définissant le fonctionnement et les caractéristiques de l'architecture à produire. Elle est importante puisque le modèle défini sert de support aux autres

étapes de la synthèse. A partir d'un fichier contenant l'algorithme du circuit à réaliser, il s'agit d'élaborer les blocs et le flot de contrôle correspondant dans la base de faits. Cette élaboration s'accompagne de quelques transformations inspirées des techniques utilisées dans les compilateurs optimisés de langage évolués [Aho 84] pour améliorer la description initiale. Ces transformations opèrent sur un plan local à chaque bloc ou global à l'ensemble des blocs. Elles ne visent pas à trouver un algorithme optimal pour le circuit (chose impossible en général), mais plutôt à améliorer la forme de l'algorithme initial suivant des critères de vitesse ou de coût.

Sur le plan local, elles consistent en général à remplacer une séquence d'opérations par une autre algébriquement équivalente mais plus efficace du point de vue du coût en matériel requis ou en temps d'exécution. On peut citer par exemple l'élimination de sous-expressions communes dans un bloc, la réduction en force des opérations coûteuses (par exemple une multiplication par une puissance de deux peut être remplacée par un décalage), la propagation des copies... Ces transformations sont appliquées pendant la construction des graphes acycliques modélisant les blocs. D'où l'utilité de ces graphes pour modéliser les blocs [Aho 84].

Sur un plan global, en plus des transformations citées précédemment, il s'agit de rendre plus efficaces les portions de l'algorithme qui sont susceptibles d'être le plus souvent exécutées, c'est-à-dire les boucles. Dans cette optique, on peut déplacer en tête de boucle, les portions de code qui calculent à chaque fois des expressions identiques, on peut dérouler les boucles, éliminer les variables d'induction (celles qui servent à compter le nombre d'itérations dans une boucle) etc. Les transformations globales sont souvent précédées d'une phase d'analyse de flot au cours de laquelle des informations sur les propriétés (vivacité, utilisation...) des variables et expressions de l'algorithme sont collectées. Toutes ces transformations sont exposées en détail dans [Aho 84].

Nous avons implémenté quelques unes de ces transformations : réduction en force des opérations (remplacement de multiplications ou de divisions par des décalages lorsque l'un des opérandes est une puissance de deux), éliminations des expressions communes au sein d'un bloc, analyse de la vivacité des variables. Le reste pourra être ajouté au cours d'un développement ultérieur du prototype ASA.

Construction des blocs de base

Dans ASA, l'analyseur syntaxique du fichier contenant l'algorithme d'entrée est faite à l'aide d'un générateur automatique d'analyseurs de langage appelé SIL et développé au sein du département AMS [Rou 88]. La syntaxe du langage à

reconnaitre est écrite suivant un modèle LL(1). Dans cette description, la définition des unités syntaxiques est parsemée d'étiquettes. Cette syntaxe est fournie en entrée à SIL ainsi qu'une procédure Ada appelée *make*contenant une instruction à choix multiple. Les entrées de l'instruction à choix multiple correspondent aux étiquettes figurant dans la syntaxe du langage d'entrée. Le programme produit par le générateur effectue l'analyse syntaxique des algorithmes qui lui sont soumis tout en invoquant la procédure *make* suivant l'ordre et les valeurs des étiquettes ponctuant la syntaxe du langage d'entrée. Au niveau du système expert ASA, l'analyse syntaxique du fichier d'entrée est traitée par une procédure (*parse*) considérée comme une action primitive. Pendant cette analyse, la procédure *parse* exécute des primitives ODSE pour construire dans la base de faits, les objets correspondant aux blocs et aux transitions.

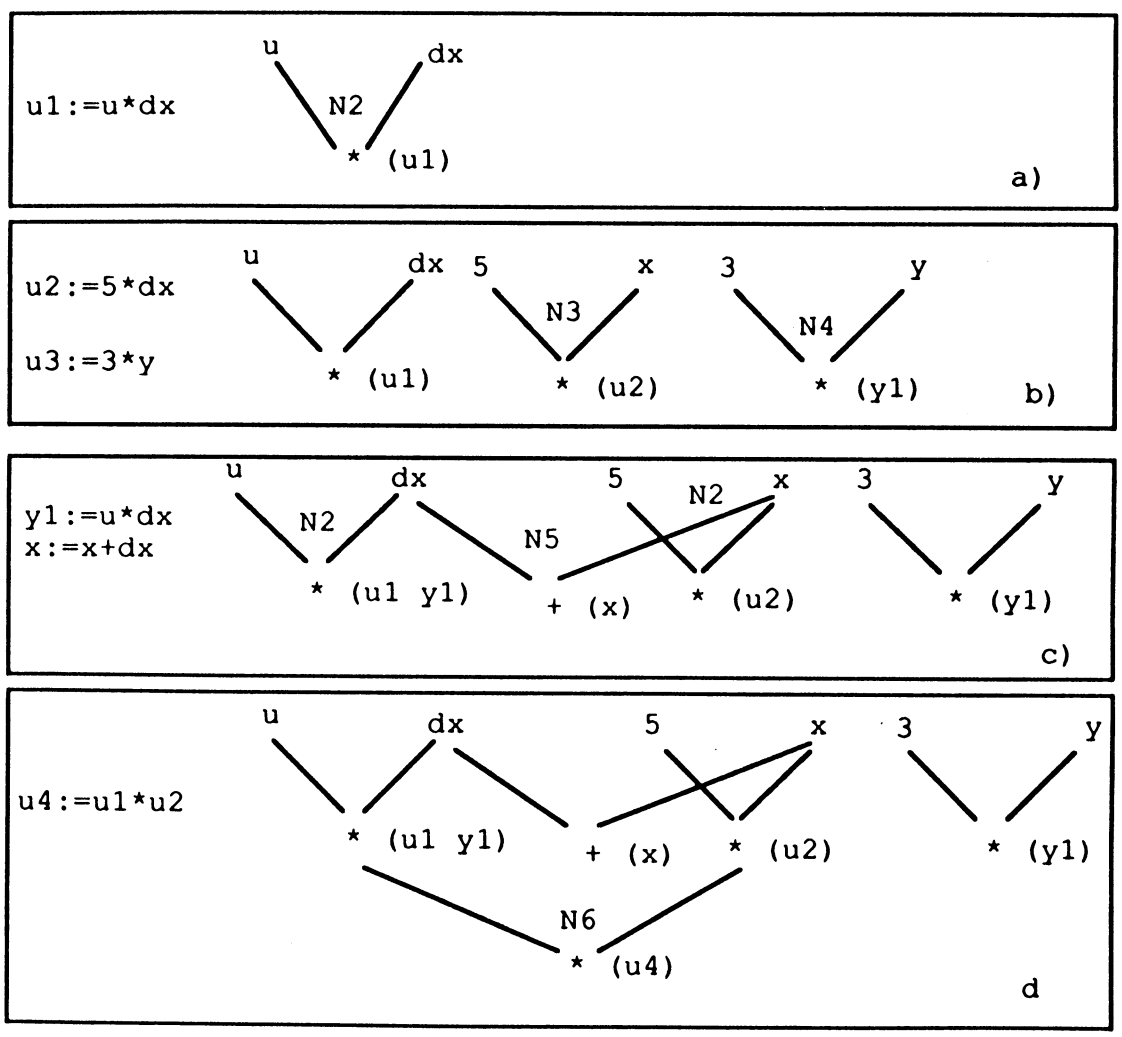


Figure 9 : construction d'un GAO

Un bloc de base est défini par une séquence d'instructions d'affectation de l'une des formes :

I)A :=B op C,

II) A :=op B,

III) A :=B,

dans lesquels *op* désigne une opération arithmétique ou de comparaison. Les expressions conditionnelles (*I<=90* par exemple) sont traitées comme les formes I) ou II) avec une partie gauche (*A*) indéfinie. En fait le membre droit peut être une expression arithmétique quelconque mais qui peut toujours se ramener à des séquences d'affectations ayant la forme ci-dessus définie. Nous illustrerons (cf figure 9) la construction d'un GAO en considérant les 6 premières instructions du BLOC 2 de la figure 3. Cette construction procède en séquence sur les instructions du bloc en trois étapes [Aho84] :

1 Soit *NODE(B)* le nœud du GAO en cours de construction comprenant *B* dans la liste des identificateurs qui lui est associé. Si un tel nœud n'existe pas (figure 9 a), on le crée en ajoutant au GAO, une nouvelle feuille ayant *B* dans sa liste d'identificateur. Dans le cas I), on procède de la même manière pour *C* ; soit *NODE(C)* le nœud résultant.

2 Dans le cas I), on cherche si dans le graphe un nœud *N* existe ayant pour fils gauche *NODE(B)* et pour fils droit *NODE(C)*. Dans le cas II), on se limite à un nœud ayant *op* comme opération et *NODE(B)* comme fils droit. Dans les deux cas, si un tel nœud n'existe pas, on en crée un. Soit *N* le nœud ainsi créé ou trouvé (cf figure 9 c et d)

3 On ajoute *A* à la liste des identificateurs du nœud trouvé au 2). Si *NODE(A)* est un nœud du GAO autre que *N* et contenant *A* dans sa liste d'identificateurs, on supprime *A* de cette liste.

Optimisation des blocs de base

Les transformations locales sont effectuées pendant la construction des graphes acycliques à travers les actions invoquées par l'analyseur suivant le protocole décrit précédement. Considérons le bloc 2 défini par la séquence (fig 3 page 78). Les feuilles du GAO (figure 10) correspondent aux valeurs initiales des variables utilisées dans le bloc. A chaque instruction *I* du bloc correspond un nœud *N* du graphe. Les nœuds fils de *N* correspondent aux instructions d'affectation les plus récentes avant *I* et définissant les opérandes de *I*. Le nœud *N* indique l'opération appliquée aux opérandes. En plus, il lui est associé une liste de toutes les variables pour qui il représente la dernière définition au sein du bloc.

Avant d'ajouter un nouveau nœud correspondant à une instruction *I* au graphe, on détecte si un nœud *M* définissant la même opération sur les mêmes opérandes existe dans le graphe. Dans ce dernier cas, on a une expression

commune et aucun nœud supplémentaire n'est ajouté au graphe. Seule la variable affectée par l'instruction *I* est ajoutée à la liste des dernières définitions associée à m. C'est le cas des deux instructions *u1 :=u*dx* et *y1 :=u*dx* (cf figure 9 c). Les deux instructions correspondent au nœud N2.

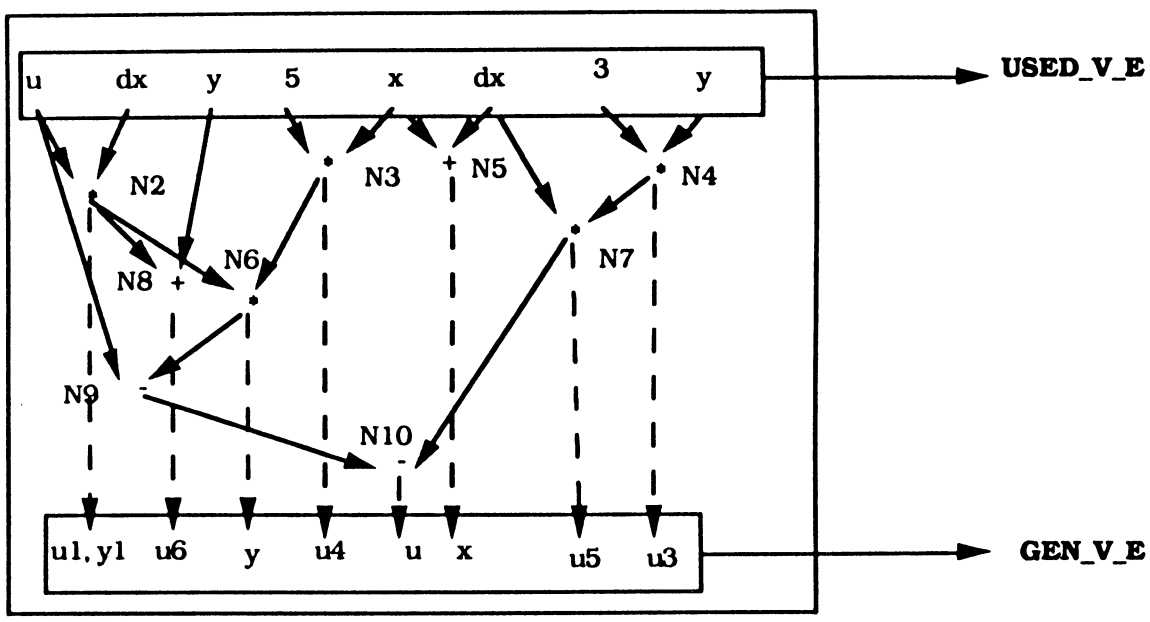


Figure 10 : graphe acyclique orienté du bloc 2

Pour détecter l'équivalence des expressions calculées (nœuds du graphe acyclique) par les instructions d'un bloc, on peut tirer parti des propriétés algébriques des opérations arithmétiques telles que la commutativité, l'associativité, les identités (exemple *2*a=a+a)*...Dans ce cas, pour les opérations commutatives par exemple, au lieu de tester la conformité des opérandes dans l'ordre indiqué, on les teste dans un ordre indifférent (*a+b=b+a*).

En même temps que les optimisations sont effectuées, certaines informations sont accumulées pendant la construction des GAO qui faciliteront l'analyse des flots. Pour chaque bloc, ASA détermine :

—la liste des variables effectivement utilisées dans le bloc. Ces variables correspondent aux identificateurs associés aux feuilles du GAO. cette liste est repérée par l'attribut *used_v_e* des objets *bloc* suivant le formalisme de ODSE(fig 10)

—la liste des variables dont les valeurs ont été modifiées dans le bloc. Ces variables correspondent aux identificateurs présents dans les nœuds intérieurs du graphe à la fin de la construction du GAO. Cette liste est décrite par l'attribut *gen_v_e* des objets de la classe *bloc* (fig 10).

Analyse des flots

En plus des transformations locales, on peut appliquer des transformations globales à l'ensemble des blocs. Dans ce cas, c'est le graphe de flot qui sert de support à ces transformations. Rappelons que les nœuds de ce graphe sont les blocs de base et les arcs décrivent les enchaînement entre ces blocs. Comme transformations globales sur ce graphe, on peut éliminer les expressions communes, propager les copies de variables (éliminer le plus possible les instructions de la forme a :=b en remplaçant toutes les occurrences de a par b dans les expressions subséquentes), déplacer les expressions invariantes en en-tête de boucle, éliminer les variables d'induction [Aho 84]. Ces dernières transformations reposent sur le concept de boucle. Autrement dit il faut détecter les portions du graphe de flot correspondant à des boucles. Cette tâche est rendue aisée pour toute une classe de graphes dits à flot réductible. Selon Aho, on peut partitionner les arcs d'un tel graphe en deux groupes :

—le premier groupe définit un arbre d'expansion c'est-à-dire un arbre permettant d'atteindre tous les nœuds du graphe à partir du nœud initial

—le deuxième groupe définit les arcs inverses. Ces arcs sont dirigés dans le sens inverse de celui défini par l'arbre d'expansion.

L'intérêt des graphes à flots réductibles pour l'analyse des boucles réside dans le fait que toute boucle naturelle est un sous-graphe du graphe d'expansion auquel on ajoute un arc inverse [Aho 84]. Ce qui simplifie le problème de la détermination de toutes les boucles naturelles du graphe de flot.

Les transformations globales utilisent un ensemble d'informations qu'il faut calculer à partir du graphe de flot. Ainsi, pour éliminer les sous-expressions communes globales, on a besoin de savoir pour chaque bloc (nœud du graphe) quelles sont les expressions disponibles en entrée et en sortie de chaque bloc, pour optimiser les boucles on a besoin de savoir quelles sont les expressions qui sont invariantes au fil des exécutions du corps de boucle...Ces informations sont définies par ce qu'il est convenu d'appeler les équations de flot.

Lorsque le graphe de flot est réductible, ces équations sont simplifiées et déterminées par la syntaxe des structures de contrôle du langage d'entrée. Dans notre cas le langage d'entrée (voir chapitre 2) est limité aux structures séquentielles, conditionnelles et itératives.

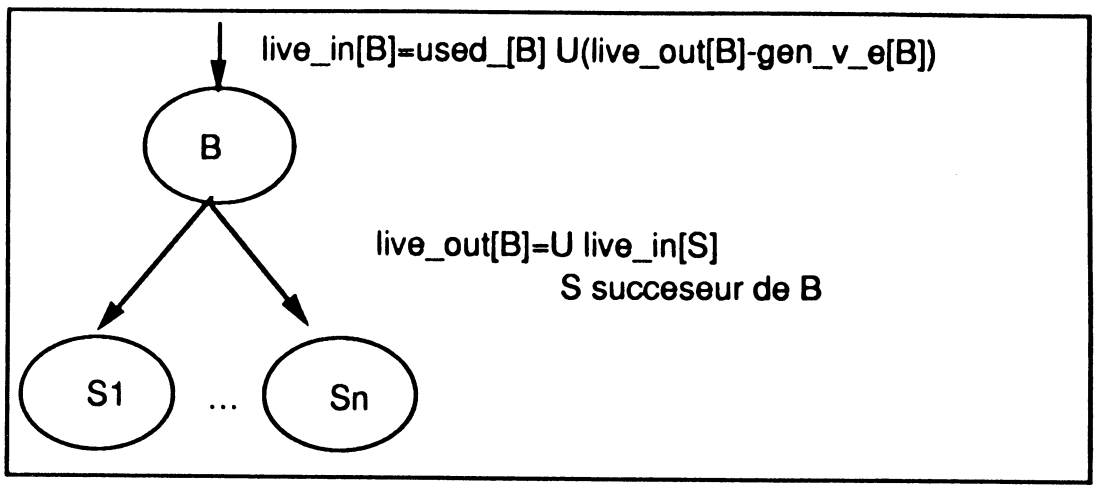


Figure 11 : Equations de flot définissant la vivacité des variables

Considérons le problème posé par l'analyse de la vivacité des variables. Il s'agit de déterminer pour chaque bloc, les variables vivantes (ou vivaces) en entrée et en sortie du bloc. une variable est dite vivante en un point (entrée ou sortie d'un bloc) du flot de contrôle, si elle est utilisée avant d'être modifiée en un point ultérieur du graphe de flot. La figure 11 donne les équations de flot permettant de calculer la liste des variables vivantes en entrée (*live_in*) et en sortie (*live_out*) d'un bloc. Dans un graphe à flot réductible, les blocs ont un seul point d'entrée et un seul point de sortie ce qui simplifie l'expression de ces équations et permet le calcul des solutions suivant l'ordre suggéré par la syntaxe des structures de contrôle et reflété par l'arbre d'expansion du flot de contrôle.

Par exemple pour calculer la liste des variables vivaces en entrée (*live_in*) et en sortie (*live_out*), on procède en deux étapes :

—lors de la construction d'un bloc (GAO), on calcule les listes des variables définies (*gen_v_e*) et utilisées avant toute définition (*used_v_e*) dans chaque bloc ;

—la vivacité des variables est calculée par application de l'operateur *lv_analysis* avec le concours des démons pour enchaîner les calculs. L'opérateur *lv_analysis* a pour effet d'ajouter la liste des variables utilisées à la liste des variables vivaces en entrée (*live_in*) :

```
to_do lv_analysis(bloc ?bloc1)
if (bloc ?bloc1 used_v_e /= null ) do
    (modify (bloc ?bloc1) (live_in [(bloc ?bloc1)live_in] or
                           [(bloc ?bloc1)used_v_e]))
end
```

```
to_do lv_analysis(bloc ?bloc1)
if (bloc ?bloc1 used_v_e = null ) do end
```

Les démons sont déclenchés à toute modification de l'attribut *live_in* ou *live_out* d'un bloc ?*bloc1* et effectuent les actions suivantes :

—à tout prédécesseur du bloc dont l'attribut *live_in* a été modifié, on ajoute à l'attribut *live_out* (liste des variables vivaces en sortie du bloc), la valeur de l'attribut *live_in* du bloc modifié :

```
when modified live_in (bloc ?bloc1)
if ?bloc2 in ( bloc in predecessors(bloc ?bloc1) )
do
    (modify (bloc ?bloc2) (live_out [(bloc ?bloc2)live_out] or
                    [(bloc ?bloc1)live_in]))
end
```

—lorsque l'attribut live_out d'un bloc (*?bloc1*) est modifié, le démon ci-dessous ajoute à l'attribut *live_in* du bloc modifié, les variables vivaces en entrées (*live_in*) mais qui ne sont pas définies dans le bloc c'est à dire qui n'appartiennent pas à l'attribut *gen_v_e* du bloc :

```
when modified live_out(bloc ?bloc1)
 if
[[(bloc ?bloc1)live_out] \([[(bloc ?bloc1) live_in]or
                          [(bloc ?bloc1)gen_v_e])|=null]
do
    (modify (bloc ?bloc1) (live_in [(bloc ?bloc1)live_in] or
                          ([(bloc ?bloc1)live_out] \
                           [(bloc ?bloc1)gen_v_e])))
end
```

N.B. Le symbole \ désigne la différence de deux ensembles.

Nous verrons plus tard comment ces informations sur la vivacité des variables sont exploitées pour réduire la quantité d'éléments de mémorisation alloués au circuit par ASA.

Nous venons de voir comment on pouvait calculer les solutions des équations de flots relatives à la vivacité des variables. Les transformations globales sont caractérisées par des équations de flots similaires. Les solutions

à ces équations facilitent l'application de la transformation correspondante. Nous ne traiterons pas davantage l'analyse des flots et invitons le lecteur intéressé à consulté l'ouvrage de Aho [Aho 84] qui traite avec plus de détails, le sujet. Dans le prototype, nous n'avons inclus que l'analyse de la vivacité des variables ; les autres transformations globales évoquées, pourront figurer dans une version ultérieure de ASA. Ces transformations peuvent extensivement être rajoutées au prototype.

De cette présentation, il ressort que les graphes acycliques orientés sont des structures bien adaptées à la représentation et à l'optimisation des blocs de base de l'algorithme d'entrée. Les algorithmes traitées par ASA ont un flot réductible ; ce qui facilite la détection des boucles et l'application des transformations globales pour optimiser la structure d'ensemble de l'algorithme d'entrée.

Acquisition des contraintes

Après lecture et transformation de l'algorithme d'entrée, ASA saisit les contraintes de l'utilisateur à travers un échange interactif guidé par des menus. Il demande à l'utilisateur le temps de cycle voulu pour le fonctionnement du circuit, et ensuite lui propose plusieurs alternatives pour optimiser l'architecture du circuit :

—exiger une vitesse de fonctionnement maximale pour une surface aussi faible que possible ;

—exiger une surface minimale pour une vitesse limite minimale imposée ;

—limiter le nombre d'opérateurs fonctionnels à utiliser : nombre maximal d'additionneurs, multiplieurs etc ;

–exiger une surface minimale pour une vitesse aussi rapide que possible.

Il ne faudrait pas se méprendre sur le sens des contraintes proposées : elles doivent être situées dans les limites imposées par la bibliothèque des opérateurs. Les expressions "surface minimale" ne doivent pas être prises à la lettre. En effet dans l'évaluation de la surface, ASA ne tient pas compte de la surface occupée par les interconnexions. Ses choix sont faits en ne considérant que la surface occupée par les opérateurs. Dans la suite, on se rendra compte que les stratégies suivies par ASA pour résoudre les problèmes d'optimisation posés par les contraintes ont un caractère arbitraire guidé par le bon sens. Ainsi dans le premier cas par exemple, ASA s'efforcera de minimiser le nombre des opérateurs suivant l'importance de leur encombrement. Par exemple il essaiera de minimiser les ressources fonctionnelles au détriment éventuel des éléments de mémorisation et d'interconnexion, il fera de même pour les éléments de mémorisation par rapport aux éléments d'interconnexion.

4.2.2. Ordonnancement des opérations

Au cours de la première tâche effectuée par ASA, l'algorithme d'entrée est décomposé en un ensemble de graphes acycliques orientés décrivant les blocs de base et un graphe de flot reflétant le transfert de contrôle entre ces blocs. Les graphes acycliques orientés représentent les diverses expressions calculées par les blocs de base. Les arcs du graphe traduisent des contraintes d'évaluation entre les expressions décrites par les nœuds du graphe. Un nœud père ne peut être évalué avant ses nœuds fils. L'ensemble des contraintes d'un GAO induisent une relation d'ordre partiel sur les nœuds du graphe qu'il faut absolument respecter pour ne pas trahir la sémantique de l'algorithme d'entrée. L'ordonnancement des opérations consiste à affecter les divers nœuds des graphes à des cycles d'horloge en respectant les contraintes d'évaluation. Les nœuds affectés à un même cycle sont destinés à être exécutés simultanément et de ce fait, nécessitent la disponibilité de plusieurs ressources matérielles. ASA essaiera dans tous les cas de minimiser la quantité de ressources matérielles nécessaire en dispersant au maximum les opérations tout en respectant les autres contraintes. En plus des contraintes topologiques exprimées par les graphes acycliques, l'utilisateur peut limiter la quantité de ressources à utiliser, imposer une vitesse de fonctionnement minimum, imposer une limite supérieure au temps d'exécution ou exiger une surface minimale. Une combinaison de ces contraintes est autorisée. Le problème d'ordonnancement est traité dans ASA par un ensemble de connaissances opératoires indiquant dans les divers cas cités, la démarche à suivre conformément au formalisme de ODSE. Ces connaissances constituent un opérateur au sens de ODSE de nom *schedule*. Nous expliquerons les stratégies préconisées par *schedule* dans les trois cas :

—minimisation des ressources utilisées pour une vitesse maximale recherchée ;

—minimisation des ressources pour une vitesse limite imposée ;

—maximum de vitesse pour un jeu de ressources contraint.

Toutes les stratégies proposées par l'opérateur *schedule* s'appuient sur une analyse des chemins critiques des GAO. Aussi, avant de décrire les trois stratégies citées, nous précisons la tâche effectuée par ASA pendant cette analyse. Cette analyse est faite sur chacun des blocs de l'algorithme d'entrée par l'opérateur *critical_path* :

```
to_do CRITICAL_PATH(chip ?chip)
if ?bloc in ( bloc / chip=?chip)
do
 (CRITICAL_PATH(bloc ?bloc))
end
```

Analyse du chemin critique d'un GAO

A chaque nœud du GAO est associé un délai correspondant au nombre de cycles qu'il faut à un opérateur de la bibliothèque pour effectuer l'opération du nœud. L'opérateur de la bibliothèque choisi pour calculer le délai du nœud dépend des contraintes. Lorsque la vitesse est le premier critère à optimiser, ASA considère les opérateurs les plus rapides ; par contre, si on est avant tout intéressé par une petite surface, ASA considère les opérateurs de moindre surface. Les nœuds du graphe acyclique étant pondérés par des délais, les chemins critiques sont ceux dont le délai cumulé des nœuds le constituant est maximal.

ASA effectue l'analyse du chemin critique d'un bloc par l'opérateur *critical_path* :

```
to_do critical_path(bloc ?bloc)
do
 (asap_schedule(bloc ?bloc))
 (asap_schedule(bloc ?bloc))
end
```

asap_schedule calcule les dates au plus tôt (attribut *asap_cycle*). La date au plus tôt d'un nœud *N* est défini par :

—si *N* est une feuille, *asap_cycle(N)*=1

—sinon

asap_cycle(N)=maximum(asap_cycle(M) +délai(M)/M prédécesseur de N).

asap_schedule déclenche le calcul des dates au plus tôt en affectant les attributs *asap_cycle* des nœuds racines du GAO à 1. Le calcul est poursuivi par des démons qui sont activés à toute modification de l'attribut *asap_schedule* d'un nœud *N*. Leur action porte sur tout nœud *M* successeur de *N* suivant les règles :

—l'attribut *asap_cycle* de *M* est contraint à une valeur supérieure ou égale à *asap_cycle(N)+délai(N)* ;

—si les dates au plus tôt (*asap_cycle*) de tous les nœuds prédécesseurs de *M* ont été alloués, on affecte la date au plus tôt de *M* à sa borne supérieure ;

—mise à jour du délai global du GAO repéré par l'attribut *nbcycles* du *bloc* considéré. *nbcycles* est initialisé à 0 avant l'ordonnancement du bloc. Si *asap_cycle(N)+délai(N) > nbcycles*, alors on affecte la valeur de *asap_cycle(N)+délai(N) à nbcycles*.

alap_schedule calcule les dates au plus tard (*alap_cycle*) des nœuds du GAO suivant une démarche inverse de *asap_schedule*. La date au plus tard d'un nœud *N* est définie par :

—si *N* est une feuille du graphe, *alap_cycle(N)*=délai maximal. Ce délai maximal est déterminé soit par une contrainte de vitesse exprimée par l'utilisateur, soit par *nbcycles* ;

—si *N* n'est pas une feuille, *alap_cycle(N) = minimum {alap_cycle(M)-délai(N)/ M successeur de N }*

A l'inverse de *asap_schedule*, *alap_schedule* commence le calcul des dates au plus tard par les nœuds feuilles dont les attributs *alap_cycle* sont affectées du délai maximal. Ici, les démons réagissent à toute modification de l'attribut *alap_cycle* d'un nœud *N* en effectuant sur tout nœud prédécesseur *M* de *N* les actions suivantes :

—l'attribut *alap_cycle* est contraint à prendre des valeurs inférieures ou égales à *alap_cycle(N)-délai(M)* ;

—si les dates au plus tard de tous les prédécesseurs de *M* ont été déterminés, on affecte à *M* une date au plus tard égale à la borne inférieure des valeurs possibles de *alap_cycle(M)*.

Après l'analyse du chemin critique, chaque nœud est caractérisé par une date au plus tard (*alap_cycle*) et une date au plus tôt (*asap_cycle*) ; la différence entre ces deux dates définit la mobilité ou le degré de liberté du nœud. Les diverses stratégies d'ordonnancement exploitent ces mobilités pour minimiser le jeu de ressources résultant de l'ordonnancement ou pour minimiser le délai d'exécution global découlant de l'ordonnancement lorsque le jeu de ressources est contraint. L'ordonnancement des opérations des GAO est conduite par la règle :

```
to_do schedulechip ?chip)
if ?bloc in (bloc/ chip=?chip)
do
```

```
  (schedule(bloc ?bloc))
end
```

qui ramène l'ordonnancement des opérations de l'algorithme d'entrée à l'ordonnancement des opérations des divers *bloc* de cet algorithme. Plusieurs stratégies sont préconisées pour ordonnancer les opérations d'un bloc en fonction :

—des buts recherchés : circuit rapide ou surface minimale,

—des contraintes imposées par l'utilisateur : limitation du jeu de ressources fonctionnelles à utiliser, vitesse limite imposée,

—des disponibilités de la bibliothèque d'opérateurs.

Chacune des stratégies proposées est assortie d'une précondition indiquant les conditions qui doivent être respectées par les objets de la base de faits pour qu'elle soit applicable. Plusieurs stratégies peuvent être applicables à une situation donnée auquel cas ASA sélectionne la stratégie dont la précondition est la plus sévère. L'heuristique d'un tel choix est que plus les préconditions sont restrictives, plus la solution proposée est spécifique et par là de meilleure qualité. Comme nous l'avons indiqué précédemment, nous analyserons les stratégies préconisées par ASA dans trois cas précis.

minimiser les ressources pour une vitesse maximale

```
to_do schedule(bloc ?bloc)
if [[(chip[(bloc ?bloc)chip])time]= constrained]
do
  (schedule_cp(bloc ?bloc))
  (schedule_ncp(bloc ?bloc))
end
```

La stratégie ci-dessus décrite permet à ASA de minimiser le jeu de ressources utilisé lorsque l'utilisateur exige un circuit à vitesse maximale. Il s'agit de trouver un ordonnancement des nœuds en un nombre minimal de cycles et utilisant le moins de ressources fonctionnelles possibles.

L'ordonnancement est fait en deux étapes : les nœuds critiques sont ordonnancés (opérateur *schedule_cp*) avant les autres (*schedule_ncp*).

(schedule_cp(bloc ?bloc)) ordonnance les nœuds critiques en les affectant à leur date au plus tôt. L'ordonnancement des nœuds du chemin critique induit une contrainte minimale sur le jeu de ressources à utiliser. Ce jeu de ressources doit permettre l'exécution simultanée de toutes les opérations affectées à un même cycle. Dans l'exemple *equadif*, ce jeu de

minimal ressources est constitué de deux multiplieurs, un additionneur, un soustracteur et un comparateur.

L'ordonnancement des nœuds non critiques commence par les nœuds dont les opérations sont les plus coûteuses en surface. Les opérations sont classées suivant l'ordre décroissant des coûts des ressources fonctionnelles pouvant les réaliser. L'heuristique de cette démarche consiste à exploiter les mobilités des nœuds non critiques pour disperser en priorité les nœuds qui requièrent les ressources fonctionnelles les plus coûteuses. Au fur et à mesure que les nœuds sont alloués à des cycles, les mobilités des nœuds restants sont réduites. Ainsi, on dispose d'une marge de manœuvre plus importante pour réduire le nombre des opérateurs les plus coûteux. Dans le cas de *equadif*, on commencera par les opérations de multiplication avant de s'occuper des autres opérations. Pour un type d'opération *op* à ordonnancer, ASA procède à partir du premier cycle jusqu'au dernier en trois étapes :

—il détermine les nœuds candidats à l'ordonnancement au cycle courant : ce sont les nœuds non ordonnancés, dont l'opération correspond à *op* et dont la date au plus tôt est inférieure au cycle courant ;

—si parmi les nœuds candidats il se trouve des nœuds devenus critiques c'est-à-dire dont la date au plus tard correspond au cycle courant, tous ces nœuds doivent être ordonnancés. Dans le cas où le jeu de ressource en considération ne permet pas un tel ordonnancement, on incrémente le nombre de ressources réalisant l'opération *op*, et on recommence l'ordonnancement de toutes les opérations *op*.

—lorsque tous les nœuds critiques ont été ordonnancés, on choisit des nœuds candidats en privilégiant ceux dont les opérandes figurent parmis les opérandes de nœuds déjà ordonnancés au cycle courant. Ce dernier choix traduit un souci de minimiser le nombre des éléments d'interconnexion. Les nœuds choisis sont affectés au cycle courant à concurrence du nombre d'opérateurs autorisés par le jeu de ressources en considération ;

—lorsqu'un cycle est saturé ou lorsqu'il n'y a plus de nœuds candidats, ASA passe au cycle suivant et le processus recommence jusqu'à l'ordonnancement de tous les nœuds.

La table 1 donne les résultats obtenus pour le circuit EQUADIF.

CYCLE	OPERATIONS
1	u1:=u*dx u2:=5*x x:=x+dx
2	u3:=3*y u4:=u1*u2 y:=y+u1
3	u6:=u-u4 u5:=dx*u3
4	u:=u6-u5

Table 1 :Ordonnancement effectué par ASA pour le circuit EQUADIFF

Ordonnancement pour un temps d'exécution plafonné

Lorsque le temps d'exécution du circuit ne doit pas dépasser une durée limite donnée, chaque *bloc* est contraint à un délai limite. Nous avons vu que ce délai limite était pris en compte dans la détermination des dates au plus tard des nœuds (cf analyse du chemin critique). Deux cas peuvent se présenter après l'analyse du chemin critique :

—s'il existe au moins un nœud dont la date au plus tard est inférieure à la date au plus tôt, la bibliothèque d'opérateurs disponible ne permet pas de satisfaire les besoins de l'utilisateur. Un message est envoyé à l'utilisateur pour l'informer de cette situation. Dans ce cas, on peut enrichir la bibliothèque avec des opérateurs plus rapides ou étendre les règles de ASA à des architectures plus efficaces (systoliques par exemple) ;

—s'il n'y a pas de nœud problème, la stratégie précédemment décrite s'applique au cas présent.

Ordonnancement pour un jeu de ressources contraint

Lorsque le jeu de ressources est contraint, les stratégies précédentes sont modifiées comme suit : ASA ordonnance l'ensemble des nœuds de manière

chronologique en commençant par le premier cycle. A chaque cycle, il applique les étapes suivantes :

—détermination des nœuds candidats : il s'agit des nœuds non ordonnancés dont le temps au plus tôt est inférieur au cycle courant ;

—les nœuds ordonnancés au cycle courant sont sélectionnés en considérant dans l'ordre, les critères suivants : les plus critiques sont choisis en priorité ; ensuite ceux dont les opérandes figurent déjà comme opérandes de nœuds déjà ordonnancés au cycle courant. Un nœud est plus critique qu'un autre si sa mobilité est plus faible ;

—lorsque le cycle courant est saturé ou lorsqu'il n'y a plus de nœud disponible, le processus est répété sur le cycle suivant jusqu'à l'ordonnancement de tous les nœuds.

4.2.3. Allocation des éléments de mémorisation

L'algorithme d'entrée décrit le fonctionnement du circuit à construire sous la forme d'un ensemble de traitements sur des données. Ces données doivent être stockées dans des éléments de mémorisation chaque fois que leur production est séparée de leur utilisation par un temps supérieur à la période d'horloge du circuit. Certaines informations sont produites et utilisées exclusivement à l'intérieur des blocs de base. Elles sont locales à ces blocs. D'autres, part contre, sont produites dans un bloc et utilisées dans d'autres blocs : on dit qu'elles sont globales puisque leur période d'utilisation excède le cadre d'un seul bloc. Le problème de l'allocation des éléments de mémorisation consiste à minimiser la quantité de ces éléments utilisés pour stocker les informations traitées par le circuit. Cette minimisation exploite le principe suivant lequel des informations dont les périodes de vie sont disjointes peuvent être stockées dans un même élément de mémorisation. Ainsi, un même ensemble d'éléments de mémorisation peut être utilisé pour stocker les données locales aux blocs de base puisque :

—ces données ne sont vivaces qu'à l'intérieur des blocs où elles sont définies ;

—les périodes d'exécution des blocs sont deux à deux disjointes.

Les éléments de mémorisation ainsi utilisés sont appelés temporaires. L'allocation des éléments de mémorisation par ASA est guidée par la règle :

```
to_do allocate(chip ?chip)
do
```

```
(GLOBAL_ALLOC(chip ?chip))
(LOCAL_ALLOC(chip ?chip))
end.
```

Elle procède en deux étapes : la première alloue des éléments de mémorisation aux données globales tandis que la deuxième affecte les temporaires aux données locales des blocs et les unités fonctionnelles aux opérations.

Allocation de mémoires aux valeurs globales

Avant d'allouer de la mémoire aux données globales, il faut déterminer ces données ainsi que leur vivacité. Une donnée est globale lorsque sa vivacité s'étend sur plusieurs blocs. Par exemple, si elle est utilisée dans au moins un bloc différent de celui dans lequel elle a été définie. Une telle donnée doit être mémorisée sur tous les chemins du flot de contrôle partant du point de sa génération aux points de son utilisation. Deux valeurs globales qui ont des vivacités disjointes peuvent partager le même élément de mémorisation. Cette condition, si elle est suffisante, ne tient pas compte de l'effet induit par la présence d'instructions conditionnelles dans l'algorithme d'entrée. Soit la portion d' un algorithme d'entrée suivante :

```
if c1 then
 a :=b+c ;
else
a :=c ;
end if ;
r :=a+b ;
```

On peut y voir que les valeurs définies par *b+c* et *c* ont des périodes de vivacité qui se recouvrent (instruction *r :=a+b*) ; malgré tout, ces deux valeurs peuvent et doivent être associées au même élément de mémorisation.

Pour minimiser le nombre d'éléments de mémorisation alloués aux valeurs globales, ASA procède en deux étapes :

—pendant la première, il essaie de constituer des groupes de valeurs mutuellement exclusives : ce sont des groupes de valeurs associées à une même variable par des instructions conditionnelles et dont les vivacités se recouvrent ;

—les éléments de mémorisation sont affectés à ces groupes suivant une démarche s'inspirant de l'algorithme de Hashimoto pour l'affectation optimale de pistes de routage aux fils de connexions d'un circuit.

Cette démarche s'exprime en ODSE comme suit :

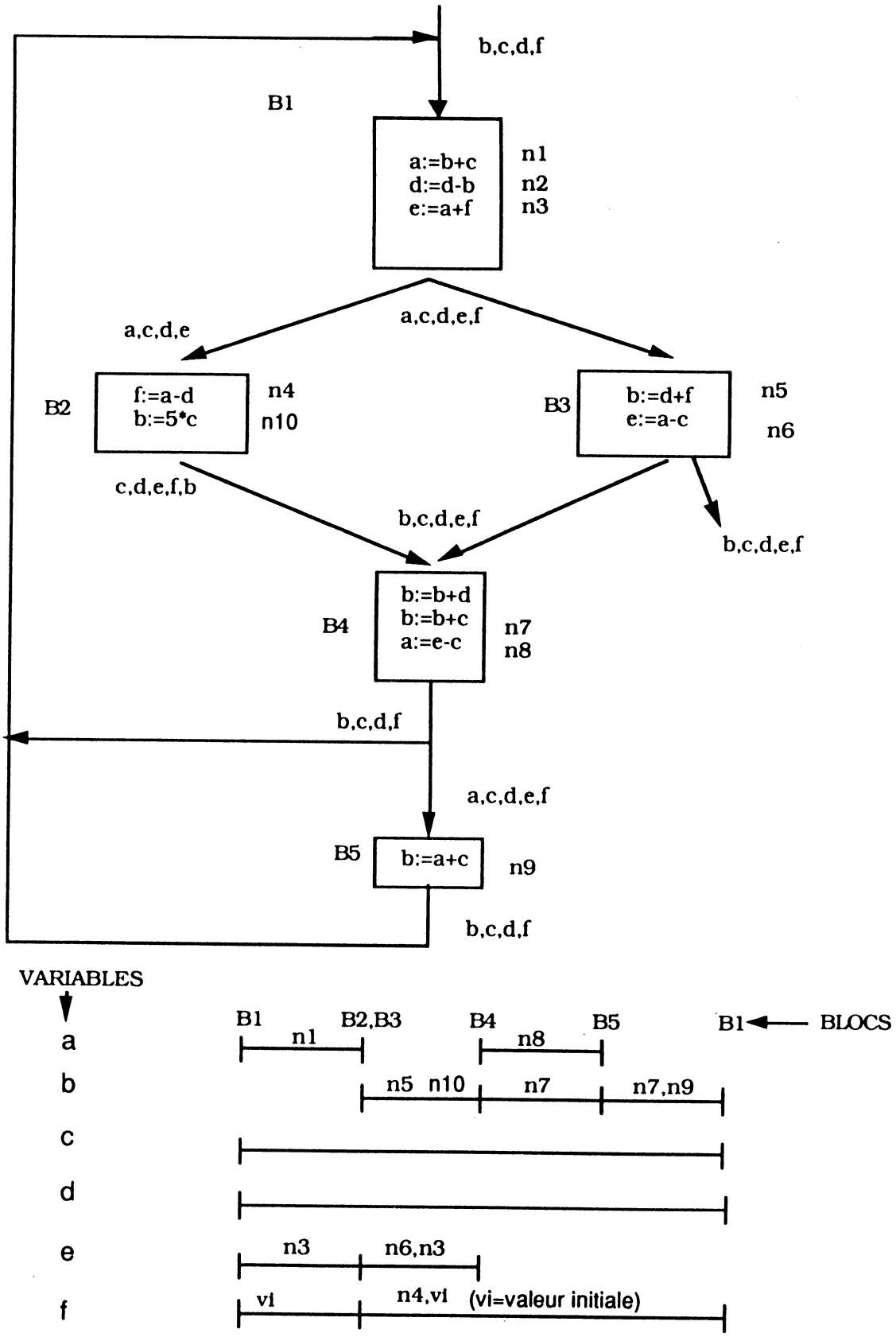


Figure 12 : Calcul de la vivacité des valeurs

```
to_do global_alloc(Chip ?chip)
do
 (group_nodes(Chip ?chip))
 (alloc_group(Chip ?chip))
end
```

Groupes de valeurs mutuellement exclusives

Afin de minimiser le nombre d'éléments de mémorisation utilisés dans le circuit, les valeurs à mémoriser doivent être partitionnées en groupes de compatibilité. Au sein d'un groupe, les valeurs ont des vivacités disjointes, ce qui leur permet de partager une même ressource de mémorisation. Le nombre de groupes distincts détermine le nombre d'éléments de mémorisation qu'il faut utiliser pour stocker les valeurs globales. On a donc intérêt à minimiser ce nombre. Dans ASA, le regroupement des valeurs mutuellement exclusives est fait après le calcul de la vivacité des valeurs.

Pour calculer la vivacité d'une valeur, ASA détermine les chemins du flot de contrôle le long desquels elle doit être mémorisée. Ces chemins partent du point de sa génération aux divers points de son utilisation. Le calcul est déduit des informations accumulées lors de l'analyse de la vivacité des variables. On sait que les blocs de base représentant l'algorithme d'entrée affectent un ensemble de valeurs aux variables. La vivacité d'une variable est la somme des vivacités des valeurs (noeuds) qui lui ont été affectés. La vivacité d'un noeud N affecté à une variable A est définie par la portion du flot de contrôle comprise entre le point de son affectation à A au point ultérieur de l'affectation d'une autre valeur à A.

Considérons l'exemple représenté à la figure 12. Nous y avons représenté un graphe de flot. Le résultat de l'analyse de la vivacité des variables y est présenté ainsi que la vivacité des noeuds.

La construction des groupes de valeurs mutuellement exclusives est faite en appliquant les étapes suivantes à chaque ensemble E de valeurs associées à une même variable V :

—un noeud quelconque N de l'ensemble de valeurs E est sélectionné ;

—si il existe un groupe construit G de nœuds inclus dans E tel que la vivacité de N recouvre la vivacité de G, N est ajouté à G sinon, un nouveau groupe est créé comprenant la seule valeur N. La vivacité du groupe G est définie par l'union des vivacités des valeurs contenues dans G.

—le processus est répété pour toutes les valeurs de E

Allocation aux groupes

Les groupes construits suivant la démarche précédente sont formés de valeurs mutuellement exclusives ; c'est à dire que tous les noeuds d'un même groupe peuvent partager le même emplacement de mémoire. Les groupes sont affectés à des éléments suivant une stratégie du type glouton : les groupes sont affectés à un même registre tant que leurs vivacités respectives sont disjointes. On essaie de maximiser l'occupation des éléments de mémorisation en préférant les groupes dont les vivacités sont contiguës à la vivacité du registre courant. La vivacité d'un élément de mémorisation est définie par la réunion des vivacités des groupes qui lui sont affectés. Lorsque la vivacité d'un groupe recouvre celle de tous les registres couramment utilisés, un nouveau registre est créé qui est alloué au groupe en question. Le processus recommence jusqu'à ce que tous les groupes soient alloués à des éléments de mémorisation. Toutefois, pendant cette allocation, lorsqu'un groupe G peut être affecté à plusieurs registres possibles, on essaie de privilégier le registre dont le choix conduirait à un travail d'interconnexion supplémentaire minimal. On préfèrera par exemple un registre déjà alloué à des groupes présentant le plus de similitudes avec G du point de vue des types d'opérations arithmétiques utilisées pour définir les valeurs du groupe. La figure 13 montre le résultat de l'allocation globale des éléments de mémorisation pour l'exemple de la figure 12.

4.2.4. Affectation locale des ressources

Lors de l'ordonnancement des opérations, ASA s'est simplement assuré que toutes les opérations d'un même cycle pouvaient être exécutées en parallèle sans conflit de ressources fonctionnelles. Il s'agit maintenant d'allouer individuellement les ressources fonctionnelles aux diverses opérations. En outre, les valeurs locales doivent être affectées à des temporaires. L'objectif de ASA pendant cette étape sera :

—d'allouer les ressources fonctionnelles en minimisant au mieux les besoins d'interconnexions ;

—de minimiser le nombre de temporaires utilisé pour stocker les valeurs locales.

Cette tâche est guidée par la règle :

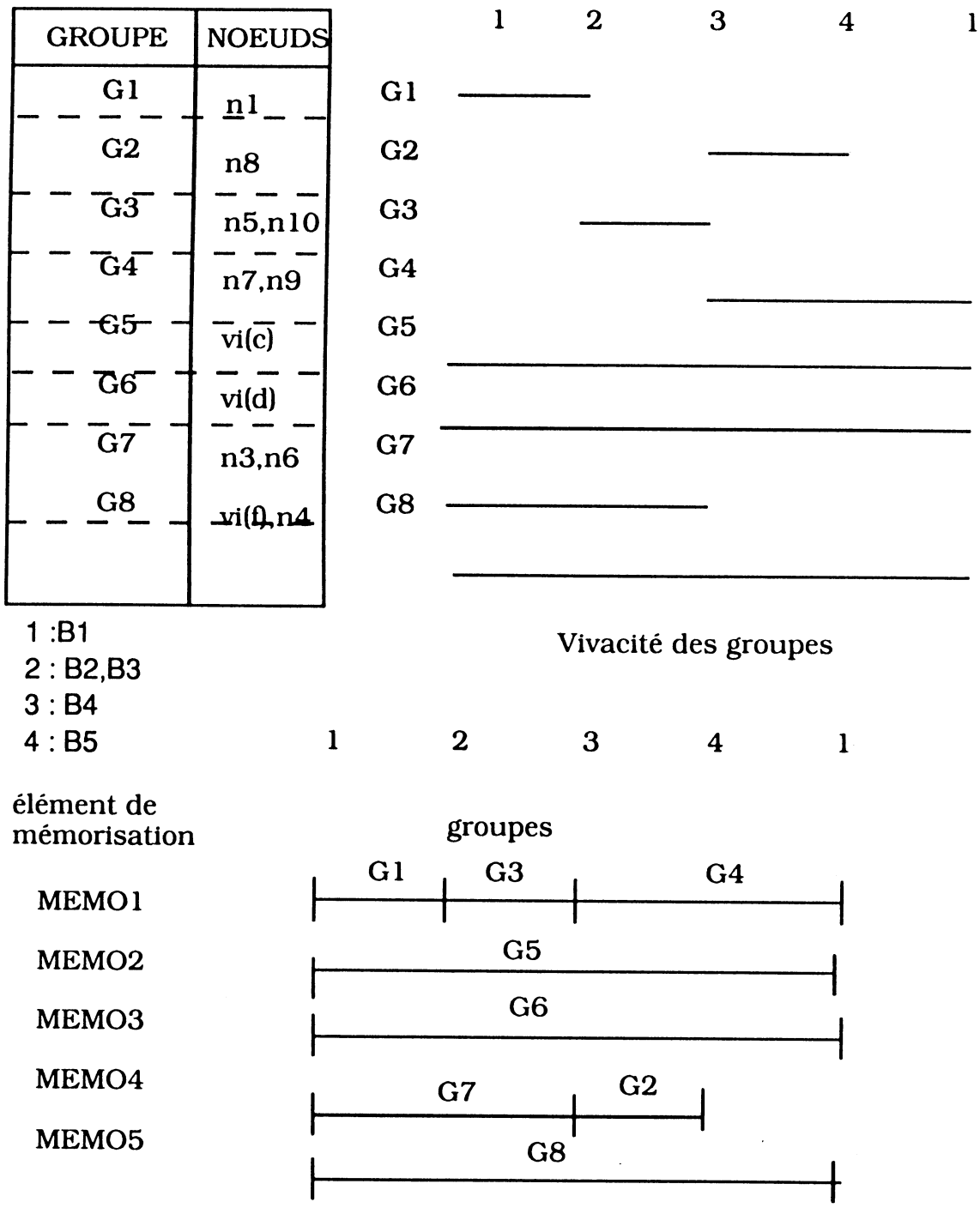


Figure 13 allocation des éléments de mémorisation aux valeurs globales de l'exemple figure12

```
to_do local_alloc(Chip ?chip)
if ?bloc in {bloc /chip =?chip}
do
  (allocate(bloc ?bloc))
end
```

(allocate(bloc ?bloc)) affecte les ressources aux opérations d'un même bloc. La démarche préconisée par *(local_alloc(chip ?chip))* consiste donc à traiter le problème de l'affectation locale des ressources, bloc par bloc.

Au niveau d'un bloc, les ressources sont allouées aux nœuds en respectant l'ordre chronologique des cycles auxquels ils ont été ordonnancés. Le processus commence par les nœuds du premier cycle. Pour chacun des nœuds d'un même cycle, l'allocation est dirigée par la règle :

```
to_do allocate(node ?node)
do
 (allocate_fu(node ?node))
 (allocate_s(node ?node))
end.
```

allocate_fu assigne une ressource fonctionnelle au noeud tandis que *allocate_s* lui affecte un temporaire s'il n'est pas une donnée globale.

Lorsqu'une unité fonctionnelle est allouée à un noeud, elle doit être connectée aux éléments de mémorisation contenant les opérandes de l'opération décrite par le nœud. *allocate_fu* s'efforce de réduire ces besoins d'interconnexion. Pour ce faire, parmi les opérateurs fonctionnels disponibles pour un nœud donné, il choisit en priorité ceux qui ont le plus de liens avec les opérandes du nœud. A cet effet, chaque fois qu'un opérateur est alloué à un nœud *N*, les attributs respectifs *input1* et *input2* sont mis à jour pour exprimer les besoins de communication avec les éléments de mémorisation contenant les opérandes respectives du nœud *N*.

Afin de limiter au mieux la quantité de mémoire, avant d'ajouter un élément de mémorisation au circuit, un effort est fait pour utiliser les mémoires déjà disponibles. Pour cela, on se sert aussi des informations sur la vivacité des éléments de mémorisation globaux. Chaque fois que l'on trouve un élément de mémorisation dont le contenu n'est pas vivace pendant un délai au moins égal à la vivacité de la donnée intermédiaire que l'on souhaite conserver, il est alloué à la donnée en question. D'autre part lorsqu'il y a le choix entre plusieurs temporaires, *allocate_s* s'efforce de choisir celui qui présente le plus de liens avec les opérateurs fonctionnels susceptibles d'être utilisés pour calculer des valeurs nouvelles à partir de la donnée pour laquelle on cherche une mémoire temporaire. Ces règles de conduite visent à minimiser le nombre de temporaires utilisés et les besoins d'interconnexion. Le détail de ces règles peut être consulté en annexe.

En marge des règles contrôlant les tâches d'allocation, d'autres règles sont utilisées pour mettre à jour les informations concernant la disponibilité des

ressources fonctionnelles. Les ressources comportent deux attributs : *available* qui indique la date à laquelle ils seront disponibles et *until* qui indique la date au delà de laquelle on ne peut plus utiliser la ressource. Au début de l'allocation des ressources aux nœuds d'un bloc, tous les attributs *available* de toutes les ressources fonctionnelles sont mises à 1 et les attributs *until* au numéro du dernier cycle du bloc. Les éléments de mémorisation contenant des données globales vivaces en entrée et en sortie du bloc en cours de traitement ne sont pas disponibles dans le bloc et de ce fait, leur attribut *until* est mis à 1 et leur attribut *available* au numéro du dernier cycle du bloc. Ainsi, on garantit qu'ils ne seront pas utilisés au sein du bloc pour conserver les résultats intermédiaires. Si des éléments de mémorisation contiennent des valeurs différentes vivaces en entrée et en sortie, leur attribut *available* est mis au cycle suivant le dernier dans lequel ils sont utilisés pour calculer une expression quelconque ; leur attribut *until* est affecté au numéro du cycle au cours duquel ils sont utilisées pour conserver une valeur vivace à la sortie du bloc. Cet artifice rend les mémoires disponibles entre le dernier instant de l'utilisation de leur contenu et le moment où elles sont utilisées pour la conservation des données globales.

Chaque fois qu'une ressource est allouée à un nœud, les informations *available* et *until* sont mises à jour de manière appropriée. Pour un détail de ces mises à jour, on peut consulter en annexe, les régles désignées par l'opérateur ODSE *update_cycle*.

4.2.5. Allocation des éléments d'interconnexion

Pendant l'allocation locale des ressources, nous avons vu que les règles exploitées par ASA cherchaient à réduire les besoins d'interconnexion entre les ressources fonctionnelles et les éléments de mémorisation. Ces besoins, même s'ils sont réduits, ne peuvent être totalement supprimés. C'est pourquoi après l'allocation locale des ressources, ASA établit des interconnexions entre les diverses ressources pour satisfaire les besoins révélés par l'allocation locale de ces ressources. Dans la base de règles, ces interconnexions sont guidées par l'opérateur (au sens de ODSE) *links_synt* suivant la règle :

```
to_do Links_synt(chip ?chip)
do
 (writeln"SYNTHESE DES INTERCONNEXIONS")
 (writeln " 1 --> Connexion point à point")
 (writeln" 2 --> Connexion par bus")
 (links_synt(Chip ?chip)(choice (ask number in 1..2))
end.
```

Deux possibilités sont offertes à l'utilisateur :

—la première consiste à faire des liaisons point à point. Des multiplexeurs sont utilisés pour lier une entrée d'une ressource à plusieurs autres ressources ;

—la deuxième propose des bus. Dans ce cas, le nombre de bus est imposé par le nombre maximal de transfert simultanés au cours d'un cycle. L'objectif de ASA dans ce cas sera de minimiser le nombre de connexions des ressources aux bus.

La première solution est triviale. les ressources sont liées suivant les besoins d'interconnexion. Chaque fois q'une entrée doit être liée à plusieurs sorties, un multiplexeur lui est aloué.

La liaison des ressources aux bus est faite en considérant les transferts d'un même cycle. Les cycles sont traités suivant le nombre décroissant de transferts différents qu'ils contiennent. Un transfert est identifié par : une source contenant la valeur à transférer et un ensemble de destinations qui doivent recevoir la valeur de la source. Pour un cycle donné, on affecte un bus à chaque transfert suivant les critères suivants :

—on sélectionne en priorité les bus qui sont déjà liés à la fois à la source et aux destinations possibles de la source. Par "destinations possibles", nous désignons l'ensemble des destinations de la source à travers tous les cycles du circuit ;

—si on n'en trouve pas, on sélectionne un bus parmi ceux qui sont connectés en entrée à un nombre maximal de destinations possibles ;

—si plusieurs bus vérifient la condition précédente, on en choisit une parmi ceux qui présentent le plus d'interconnexions avec les sources possibles des destinations du cycle courant.

Ces critères peuvent être rajoutés à peu de frais pour le temps d'interprétation des règles correspondantes si la base de règles est compilée (cf chapitre 3 paragraphe 3.3.3).

4.2.6. Optimisation générale

L'optimisation générale n'a pas été réalisée dans le prototype faute de temps. Dans cette section, nous donnons les indications générales des activités composant cette étape de la synthèse. Elle est conçue comme une phase au cours de laquelle ASA s'engage dans un processus itératif et interactif avec le concepteur visant à améliorer la qualité de l'architecture produite ou à explorer les caractéristiques d'architectures différentes mais voisines de celles générées.

Optimisation de l'architecture

L'optimisation de l'architecture consiste à appliquer au circuit, des transformations qui réduisent sa surface sans violer les contraintes imposées. Ces transformations essaient de remplacer autant que possibles des groupes de composants onéreux en surface par d'autres moins coûteux. Les composants incriminés peuvent être soit des unités fonctionnelles, soit des éléments de mémorisation. On peut distinguer deux sources principales d'optimisation : le remplacement de plusieurs unités fonctionnelles par des unités aritmétique et logique (UAL), le remplacement de plusieurs éléments de mémorisation par des RAM

Deux ressources quelconques sont considérées comme mutuellement exclusives si elles ne sont jamais utilisées en même temps. Le remplacement sera évidemment fait à trois conditions :

—que les ressources fonctionnelles aient des performances de vitesse voisines c'est à dire qu'elles nécessitent toutes un même nombre de cycles N pour effectuer leurs opérations respectives ;

—qu'une UAL soit disponible en bibliothèque pouvant réaliser chacune des opérations voulues avec un temps de fonctionnent d'au plus N cycles ;

—enfin, que la surface de l'UAL soit moindre que la somme des surfaces des unites fonctionnelles que l'on veut remplacer.

Les deux premiers critères sont nécessaires puisqu'on ne veut pas remettre en cause l'ordonnancement des opérations. Cet ordonnancement tient compte de la durée de fonctionnement des unités fonctionnelles exprimée en nombre de cycles.

On peut remplacer plusieurs éléments de mémorisation par des RAM simple accès chaque fois que :

—les éléments de mémorisation ont des vivacités mutuellement exclusives ;

—une RAM est disponible en bibliothèque avec la taille adéquate : au moins autant d'emplacements mémoire que le nombre d'éléments de mémorisation à remplacer ;

—la surface de la RAM est inférieure à la surface cumulée de tous les éléments de mémorisation.

Fusion d'unités fonctionnelles en UAL

La fusion de plusieurs unités fonctionnelles consiste à les remplacer par une UAL pouvant effectuer les opérations de toutes les unités impliquées. Cette fusion a une incidence directe sur l'allocation locale, aussi est-il souhaitable d'effectuer toutes les fusions possibles avant l'allocation locale. La fusion opère de manière itérative en deux étapes : recherche d'un sous-ensemble d'unités fonctionnelles candidates à la fusion, et remplacement de ce sous-ensemble par l'UAL appropriée. Le processus s'arrête lorsqu'il n'y a plus de fusion possible.

Le sous-ensemble d'unités fonctionnelles candidates à la fusion doit obéir aux trois critéres évoqués. La démarche que nous préconisons dans cette recherche consiste à se limiter aux UAL de la bibliothèque dont toutes les opérations sont utilisées dans la description fonctionnelle du circuit. Pour chaque UAL remplissant la condition précédente, on détermine s'il lui correspond dans le circuit un jeu de ressources fonctionnelles mutuellement exclusives dans leur période d'utilisation. La recherche est effectuée en commençant par les UAL les plus composites c'est-à-dire celles qui réalisent le plus grand nombre d'opérations. Cette stratégie s'exprime en ODSE par :

```
to_do merge_fu(Chip ?chip)
if ?lual is (UAL/ operations all_in [(Chip ?chip)operations])
do
 (Check_Merge(UALS '?lual)(chip ?chip))
end
to_do Check_Merge[(Uals ?uals)(Chip ?chip)]
if ?best_uals is (Ual in  (maximums (Ual in '?uals)nb_operations))
merge_possible[(Ual (first '?best_uals))(Chip ?chip)]
do
 (merge(UAL (first '?best_uals))(Chip ?chip))
 (Check_Merge(uals '?uals)(chip ?chip))
end
to_do check_Merge[(Uals ?uals)(Chip ?chip)]
if ?best_uals is (Ual in '?uals/(max nb_operations))
 not merge_possible[(Ual (first '?best_uals))(Chip ?chip)]
do
 (merge(uals (?uals \ ?best_uals))(chip ?chip))
end
to_do check_merge[(Uals ?uals)(Chip ?chip)]
```

```
if ['?uals=null]
do
end
```

Dans la base de faits, à chaque *ual* décrite est associé l'ensemble des opérations qu'elle peut réaliser. *merge_fu* restreint la recherche aux *ual* de la bibliothèque dont les opérations sont utilisées dans le circuit. la règle *check_merge* dirige l'opération de fusion comme un processus itératif. Ce processus explore en séquence la liste des *ual* sélectionnés par *merge_fu*. Lorsqu'il rencontre une *ual* dont les opérations correspondent à des unités fonctionnelles fusionnables, il invoque *merge* qui effectue la fusion. Le processus est itéré jusqu'à ce qu'il n'y ait plus de fusion possible. Remarquons que la liste des *ual* est explorée en commançant par celles qui réalisent le plus grand nombre d'opérations *(maximums (ual in '?uals)nb_operations)*. La détection de sous-ensembles d'opérateurs fonctionnels fusionnables est dévolue au théorème *merge_possible* qui vérifie deux conditions :

—que la surface de *l'ual* est moindre que la surface des unités fonctionnelles convoitées ;

—que les unités fonctionnelles sont fusionnables c'est à dire que deux quelconques d'entre elles ne sont jamais simultanément utilisées au fil des cycles d'exécution du circuit. Cette condition est exprimée par le théorème *compatible_fu* par :

```
if ['?fus=null] conclude Compatible[(f_units ?fus)(chip ?chip)] end
if Compatible[(f_unit (first '?fus))(f_units (rest '?fus))(chip
    ?chip)]
 Compatible[(f_units (rest '?fus))(chip ?chip)]
 conclude Compatible[(f_units ?fus)(chip ?chip)] end
```

qui définit inductivement la compatibilité d'un ensemble d'unités fonctionnelles. un ensemble vide d'unités fonctionnelles est compatible ; si une unité fonctionnelle est compatible à un ensemble de *N-1* autres unités, et si cet ensemble de *N-1* unités est compatible, on peut déduire que l'ensemble des *N* unités est compatible. Reste à définir la compatibilité d'une unité fonctionnelle par rapport à d'autres. On peut procéder négativement en posant :

```
if ?operation2 in (famille in '?fus)
 [(cycles/ ?operation1=[(Chip ?chip)?operation1],
 ?operation2=[(Chip ?chip)?operation2])≠null]
conclude
 not Compatible[(f_unit ?operation1)(f_units ?fus)(chip ?chip)]
end
if
 unknown Compatible[(f_unit ?operation1)(f_units ?fus)(chip ?chip)]
conclude
 Compatible[(f_unit ?operation1)(f_units ?fus)(chip ?chip)]
end
```

On détermine plus facilement la non compatibilité en cherchant parmi les cycles du circuit s'il n'en existe pas un qui utilise le nombre maximum d'unités fonctionnelles de la catégorie *?opération1* et pour une autre catégorie de la liste *?fus* distincte de la première. Dans l'affirmative, il y a recouvrement de toutes les unités fonctionnelles des deux catégories incriminées. Celà suffit à conclure que aucune unité fonctionnelle de la catégorie *?operation1* ne peut être compatible à un ensemble d'unités fonctionnelles de catégories respectives contenues dans la liste *?fus*.

merge est un opérateur ODSE qui fait la fusion en remplaçant un sous ensemble d'unités fonctionnelles fusionnables par une UAL. Deux cas sont envisagés :

—si l'allocation locale de ressources n'a pas été effectuée, la fusion se limite au remplacement des unités fonctionnelles candidates par l'UAL résultante ;

—dans le cas contraire, il faut revoir l'allocation locale. Une solution simple consisterait à la refaire.

Regroupement de mémoires en RAM

Chaque fois que des éléments de mémorisation sont accédés en mutuelle exclusion, ils peuvent être fusionnés en RAM. Mais la fusion n'est intéressante que s'il en résulte un gain de surface et si le temps d'accès à la RAM n'est pas trop important.

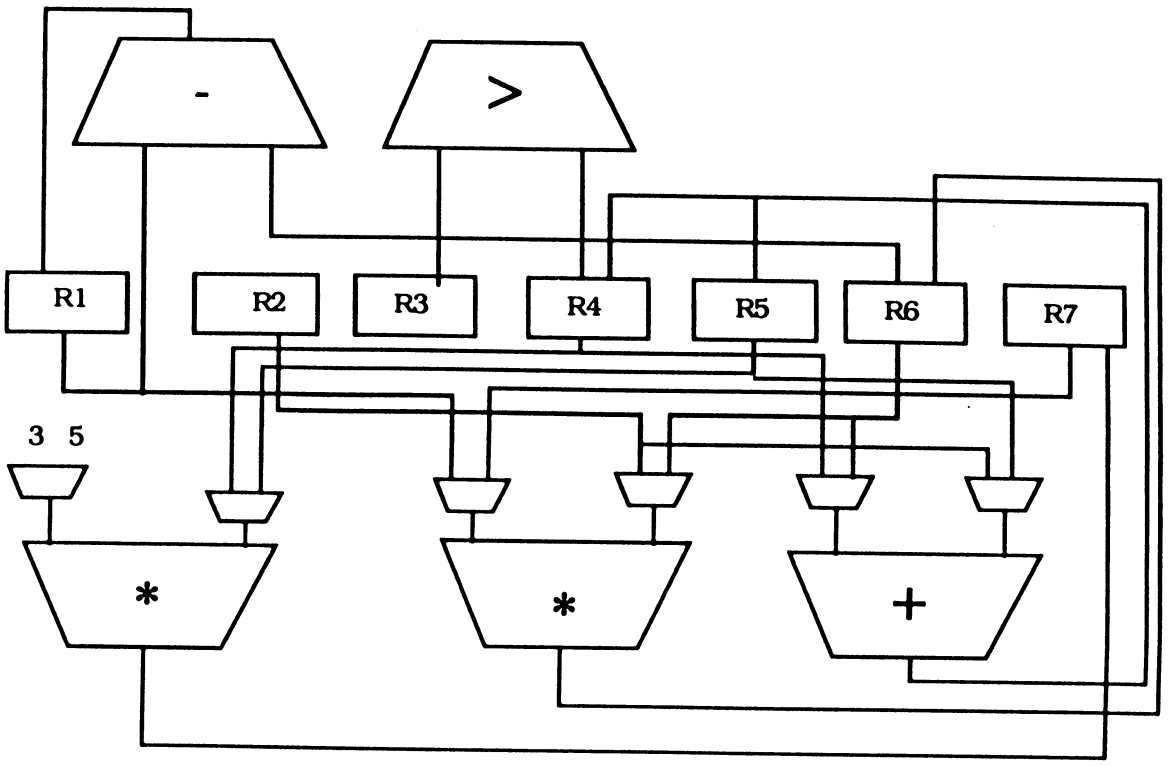
Exploration de l'espace des architectures

L'exploration de l'espace d'architectures est obtenue en relançant la synthèse avec des paramètres différents. Pour des paramètres légèrements différents, seule une partie de l'architecture a besoin d'être modifiée et il paraît

couteûx de faire une synthèse complète. Dans certains cas, il est possible de déterminer a priori les portions du circuit qui doivent être révisées. Dans ASA, cela est favorisé par le fait que la description fonctionnelle du circuit est éclatée en blocs distincts dont la réalisation peut être effectuée indépendamment les uns des autres. Si par exemple on réduit le nombre d'additionneurs de trois à deux, on n'a pas besoin de réordonnancer les opérations des blocs qui n'en comportent pas plus de deux par cycle. Pour contrôler les changements à apporter à une architecture en fonction de l'évolution des paramètres et des contraintes de l'utilisateur, nous proposons d'utiliser les techniques de maintenance de cohérence dans les systèmes ("truth maintenance system") ([Doy 79]). Ces systèmes découlent du "dependancy directed backtracking" utilisés dans les systèmes comme EL et TROPIC pour la gestion des échecs. Ils proposent des méthodes formelles permettant de calculer automatiquement les conséquences induites dans un ensemble d'assertions si certaines hypothèses changent de valeur de vérité.

4.2.7. Sortie des résultats

Pour le moment, l'architecture produite par ASA est décrite dans un fichier suivant la syntaxe de SPOT. SPOT est un générateur de partie opérative en tranches (bit-slice). La figure 14 décrit l'architecture générée pur le circuit *equadif* lorsque les contraintes imposent de minimiser le jeu de ressources utilisé pour une vitesse maximale. La durée du cycle de base choisi permet d'exécuter toutes les opérations sur un seul cycle. En annexe, nous donnons la description du fichier spot correspondant.



CONDITIONS :
- vitesse maximale
- toutes les opérations s'exécutent sur un cycle
- connexions point à point (multiplexeurs)

Figure 14 Partie opérative du circuit EQUADIFF

5. Comparaison de ASA à quelques systèmes existants

Dans le présent chapitre, nous comparons ASA à quatre systèmes automatiques de synthèse architecturale pris dans la littérature : FACET, HAL, ADAM(MAHA-REAL), et CHIPPE. Ces systèmes ont été choisis pour deux raisons principales :

—ils reposent sur des hypothèses assez voisines de ASA ce qui donne un sens à la comparaison des performances sur des exemples précis. En effet ils sont davantage orientés vers la synthèse de circuits spécifiques que celle des microprocesseurs. D'autre part, ils ne présupposent pas une structure d'architecture figée comme c'est le cas pour SYCO [Jer 86] par exemple. Comme ASA, ils supposent une bibliothèque d'opérateurs d'où sont tirés les modules constituant l'architecture. Puisqu'ils partent d'une description fonctionnelle du circuit, pour en produire l'architecture, ils sont confrontés aux mêmes problèmes que ASA : ordonnancement des opérations, allocation des ressources, allocation des éléments de mémorisation, synthèse des interconnexions ;

—Ces systèmes illustrent la tendance qui se dégage dans la synthèse automatique des circuits intégrés. Cette tendance s'affirme par un recourt croissant aux méthodes issues de l'intelligence artificielle. FACET, MAHA et REAL proposent une solution purement algorithmique ; HAL utilise la programmation par objets ; CHIPPE et ADAM sont encore plus voisins de ASA puisqu'ils suivent une approche système expert.

Dans les quatre premières parties, nous présentons les quatres systèmes choisis. L'accent est mis sur les solutions qu'ils proposent aux cinq problèmes de synthèse : représentation interne de la description fonctionnelle, ordonnancement, allocation de ressources fonctionnelles, allocation des éléments de mémorisation, synthèse des interconnexions. Chaque fois, nous comparons les solutions proposées à celles développées dans ASA. Chaque présentation est suivie d'une comparaison des résultats sur un exemple de description fonctionnelle. Bien évidemment ces exemples à eux seuls ne suffisent pas à établir une hiérarchie ou un classement des systèmes étudiés. Il faudrait une série de plusieurs exemples pour obtenir des conclusions plus fiables. Ce que nous ne pouvons faire puisque nous ne disposons pas des autres systèmes. Nous terminerons le chapitre en explorant les possibilités de ASA pour la conception d'un filtre Leapfrog.

5.1. FACET

FACET [Tse 83] a été développé à l'université de Carnegie-Mellon par Tseng et Siewiorek. Comme nous l'avons signalé, FACET suit une démarche purement algorithmique pour résoudre les problèmes de synthèse. Les problèmes d'allocation sont modélisés sous la forme d'un problème de partitionnement de graphe en cliques. Dans un graphe, une clique est un sous-graphe maximal (au sens de l'inclusion) et complet. Dans une allocation, des tâches (opérations dans le cas des unités fonctionnelles, valeurs dans le cas des éléments de mémorisation, transferts dans le cas des interconnexions) qui s'exécutent en des périodes de temps disjointes peuvent partager la même ressource. FACET modélise le problème d'allocation par un graphe dont les nœuds sont des tâches et les arcs représentent la relation de compatibilité : deux nœuds sont liés s'ils peuvent partager la même ressource. Le partitionnement du graphe en un nombre minimal de cliques corresond à une allocation optimale. Le partitionnement en cliques étant un problème np-complet, FACET propose une stratégie du type Glouton pour résoudre ce problème.

5.1.1. Représentation interne

La description fonctionnelle est traduite dans une forme interne appelée Value-Trace. C'est une forme dérivée des graphes acycliques orientés utilisés dans les compilateurs optimisés. Le Value-Trace étend le formalisme des GAO pour pouvoir exprimer des actions conditionnelles et des appels de procédure. FACET reçoit en entrée, une description fonctionnelle exprimée suivant le formalisme du Value-Trace. Chaque portion de cette description correspondant à un bloc de base est convertie en une séquence d'opérations.

5.1.2. Ordonnancement des opérations

Les opérations sont ordonnancés suivant leur date au plus tôt. Ce qui garantit un parallélisme maximal mais au prix de ressources fonctionnelles supplémentaires puisqu'aucun effort n'est fait pour disperser les opérations qui ne sont pas situées sur le chemin critique.

```
Circuit exemple_facet is
var v1,v2,v3,v4,v5,v6,v7,v8,v9,
v10,v11,v12,v13,v14,v15: integer;
begin
 loop
   v3:=v1+v2;
   v5:=v3-v4;
   v7:=v3*v6;
   v8:=v3+v5;
   v9:=v1+v7;
   v111:=v10/v5;
   v12:=100;
   v13:=v3;
   v12:=v1;
   v14:=v11 and v8;
   v15:=v12 or v9;
   v1:=v14;
   v2:=v15;
 end loop;
end exemple_facet;
```

Figure 1 exemple de circuit réalisé par FACET [Ts 83]

5.1.3. Allocation des ressources fonctionnelles

FACET décompose le problème d'allocation de ressources fonctionnelles en deux :

—minimiser le nombre d'opérateurs par type d'opération ;

—regrouper les opérations mutuellement exclusives en UAL.

Les deux problèmes sont modélisés dans FACET en construisant un graphe dont les nœuds sont les opérations du Value-Trace et les arcs représentent les possibilités de partage de ressource entre les nœuds compte tenu des cycles d'ordonnancement. Certains regroupements sont plus avantageux que d'autres : par exemple, il est plus avantageux de regrouper des opérations de même type que des opérations de types différents. Dans le premier cas, on fait l'économie d'une ressource fonctionnelle alors que dans le deuxième, on fusionne deux ressources distinctes en une UAL. C'est pourquoi les arcs du graphe sont répartis en 8 catégories suivant les avantages de regroupement que présentent les nœuds source et destination des arcs [Tse 83] :

catégorie 8 : les opérations et les trois paires de variables des nœuds sont identiques

catégorie 7 : les opérations sont différentes mais les 3 paires de variables sont identiques

catégorie 6 : les opérations et deux paires de variables sont identiques, la troisième paire est constituée de variables différentes

catégorie 5 : deux paires de variables sont identiques. La troisième paire et les opérations sont différentes

catégorie 4 : les opérations et une des paires de variables sont identiques ; les autres paires étant différentes

catégorie 3 : seules une paire de variables sont identiques

catégorie 2 : seules les opérations sont identiques

catégorie 1 : ni les opérations, ni les paires de variables ne sont identiques

Les cliques sont construites en préférant les arcs des catégories les plus intéressantes.

Le résultat de FACET pour l'exemple figure 1 définit 3 ressources fonctionnelles : la première est une UAL pouvant effectuer l'addition, la multiplication, la disjonction (ou logique) ; la deuxième comprend la soustraction, l'addition, et le et-logique ; la dernière est un diviseur. Dans ASA, nous n'avons pas réalisé la fusion de ressources fonctionnelles faute de temps. Pour le même exemple, on utilise deux additionneurs, un multiplieur, un diviseur, un soustracteur, un opérateur ou-logique, et un opérateur et-logique.

Le premier critère que cherche à optimiser FACET est déjà biaisé par l'ordonnancement. Comme nous l'avons vu au chapitre précédent, cet ordonnancement détermine un nombre minimal d'opérateurs distincts pour chaque type d'opérateurs utilisé. Le premier critère ne peut être optimisé que pendant l'ordonnancement. Néanmoins, la formulation est bien indiquée pour le regroupement d'opérations distinctes en UAL.

5.1.4. Allocation des éléments de mémorisation

Le problème de l'allocation des éléments de mémorisation est aussi formulé en termes de décomposition d'un graphe en cliques. Les variables sont les nœuds du graphe. Un arc entre deux variables traduit le fait que ces variables ont des vivacités disjointes.

Ainsi, la formulation du problème d'allocation de mémoire effectuée par FACET est np-complet. Pour l'exemple, FACET arrive à regrouper les variables en 8 éléments de mémorisation. ASA n'utilise que sept éléments de mémorisation.

5.1.5. Synthèse des interconnexions

La synthèse des interconnexions est également modélisée comme un problème de partitionnement de graphes en cliques. Cette fois, les nœuds modélisent les transferts de données entre les ressources et les arcs traduisent la

non simultanéité de ces transferts. Un arc relie deux transferts si ceux-ci ne sont pas effectués en même temps. Pour l'exemple, les interconnexions sont réalisés à l'aide de deux multiplexeurs deux entrées et un multiplexeur 3 entrées. Dans ASA nous n'avons utilisé qu'un seul multiplexeur à deux entrées. Mais ces résultats ne sont pas comparables puisque ASA n'utilise que des opérateurs mono-fonction ce qui dans le cas présent, réduit les besoins de communication.

5.2. HAL

Alors que FACET aborde la synthèse architecturale par une méthode purement algorithmique, HAL [Pau86] utilise les techniques de programmation par objets(LOOPS). L'ordonnancement et l'allocation sont effectués dans HAL par deux modules qui interagissent pour produire par itération successive, une architecture correspondant à des contraintes de temps. Le prototype initial a été étendu pour prendre en compte les architectures pipeline et les contraintes de surface.

5.2.1. Représentation interne

L'algorithme est représenté par un graphe de flot qui décrit à la fois les blocs de base et le flot de contrôle. La figure 2 schématise le graphe de flot utilisé par HAL pour représenter le circuit EQUADIFF.

On peut remarquer que contrairement à ASA, cette représentation interne n'est pas optimisée. L'expression "$u*dx$" y figure deux fois.

5.2.2. Ordonnancement des opérations du graphe

Comme ASA, HAL détermine le chemin critique du graphe de flot avant de s'attaquer à l'ordonnancement des opérations. Les opérations du chemin critique sont ordonnancées d'office. Ensuite, il s'agit de s'occuper des opérations non critiques en s'efforçant de réduire au mieux le jeu de ressources. HAL y procède en trois étapes :

la première étape vise la minimisation de la concurrence d'opérations semblables. A cet effet, il suppose une probabilité uniforme d'affectation d'une opération à l'un quelconque des cycles compris entre sa date au plus tôt et sa date au plus tard. Le graphe de distribution résultant guide l'affectation des

opérations non critiques aux divers cycles. Cette affectation procède de manière itérative.

Pendant la deuxième phase, MIDHAL (cf plus loin) fournit l'estimation d'une allocation de ressource basée sur l'ordonnancement de l'étape précédente.

La troisième étape utilise la même démarche que la première pour minimiser l'apparition simultannée d'opérations utilisant des ressources similaires.

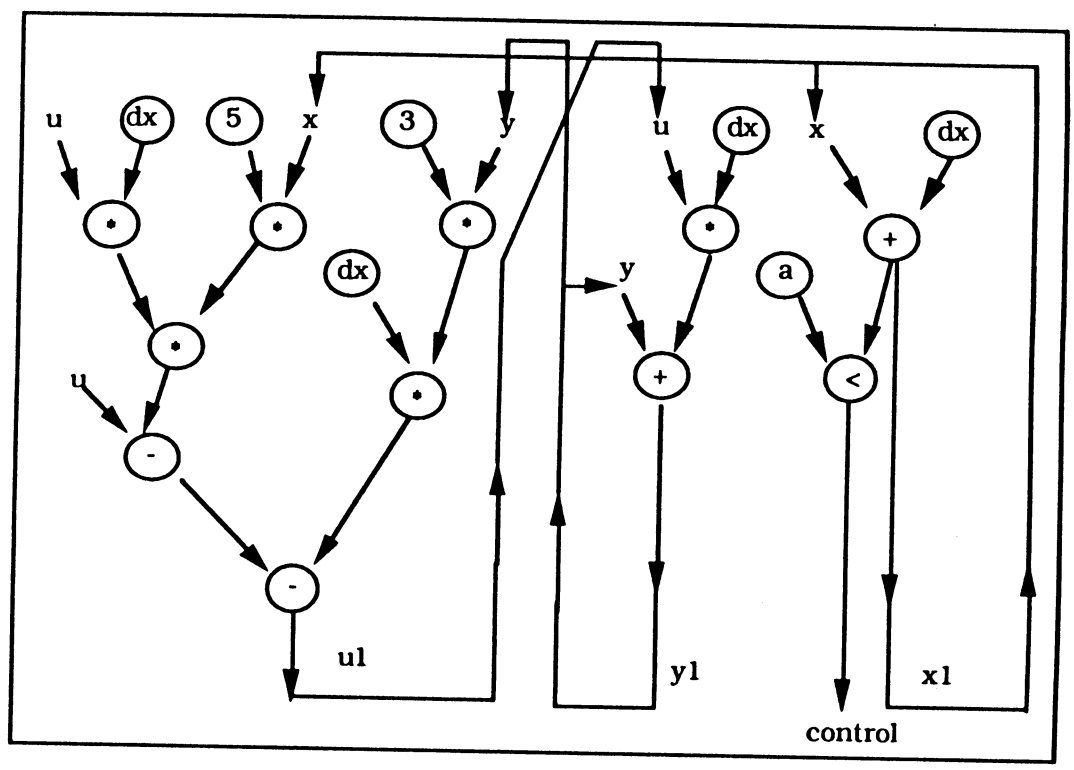


Figure 2 : graphe de flot produit par HAL pour le circuit EQUADIFF [Pau87]

5.2.3. Allocation globale des ressources fonctionnelles

Le module MIDHAL est conçu comme un système expert. Il exploite une base de règles pour effectuer l'allocation de ressources fonctionnelles pour un ordonnancement donné. Cette allocation tient compte des disponibilités d'une bibliothèque d'opérateurs. Elle procède aussi de manière itérative. Les unités fonctionnelles allouées ne sont pas affectées à des nœuds spécifiques du graphe. Cette affectation est le fait de EXHAL.

5.2.4. Allocation locale

Pendant l'allocation locale, le Module EXHAL effectue plusieurs étapes :

—affectation des ressources fonctionnelles aux nœuds du graphe ;

—allocation de registres. Cette allocation est modélisée comme un problème de partitionnement d'un graphe en cliques ;

—synthèse des interconnexions : les multiplexeurs sont alloués à la demande.

On peut remarquer certaines similitudes entre la stratégie d'allocation de ressources suivie par HAL et ASA. L'un comme l'autre affectent les ressources fonctionnelles en deux étapes. La première étape alloue globalement un ensemble de ressources fonctionnelles à un ensemble d'opérations. Cette étape définit le jeu de ressources fonctionnelles. La deuxième étape affecte localement les ressources fonctionnelles aux opérations de manière spécifique. Des différences néanmoins sont à noter entre les deux approches : tandis que HAL formalise le problème d'allocation de temporaires comme un problème de partitionnement d'un graphe en cliques (problème np-complet dans sa généralité), ASA suit une approche hiérarchique (affectation des registres globaux à l'ensemble des blocs, et ensuite allocation de temporaires au sein de chaque bloc) et inspirée du problème d'allocation optimale des pistes d'un canal de routage(problème dont une solution optimale a été proposée par [Has77]).

Les ressources fonctionnelles allouées par HAL au circuit EQUADIFF sont les mêmes que celles allouées par ASA : deux multiplieurs, un soustracteur, un comparateur, et un additionneur. Par contre, HAL utilise 6 registres alors que ASA n'en utilise que 5. HAL alloue 4 multiplexeurs à 2 entrées et 2 à trois entrées soit 14 entrées au total tandis que ASA se contente de 6 multiplexeurs à 2 entrées soit 12 entrées au total.

5.3. ADAM (MAHA-REAL)

MAHA et REAL font partie du projet ADAM ("Advanced Design Automation" [Kna 86]) de l'université de CAROLINE du SUD. ADAM est un ensemble de modules fondé sur le concept de planification. Comme ASA, il envisage la conception de circuits intégrés comme un processus de planification. Il s'agit de

chercher une séquence appropriée d'opérateurs dont l'application à une description abstraite et initiale du circuit, permet de le construire. Des différences existent néanmoins entre les voies suivies par ODSE et ADAM. Le chaînage arrière n'est pas utilisé dans ADAM. La construction de plan est effectuée dans ADAM comme un problème de recherche dans un graphe état-opérateurs suivant l' algorithme A*. ODSE au contraire, remplace le problème complexe de construction de plan par une sélection dans un catalogue de squelettes de plans éprouvés.

Dans ADAM, la synthèse architecturale est conduite par MAHA et REAL. MAHA et REAL sont vus comme des opérateurs de ADAM.

MAHA est un programme écrit en lisp [Par 86] pour la synthèse de circuits au niveau registre de transfert à partir d'un flot de donné. MAHA ne traite donc pas le problème de la traduction d'un algorithme en un flot optimisé de donnée. MAHA ne traite que le problème d'ordonnancement et d'allocation des unités fonctionnelles à un graphe de flot donné sous contraintes de temps et de resource. le problème d'allocation des registres est traité par REAL [Kur 87]. Le problème de la réduction des éléments d'interconnexion n'est pas abordé au cours des allocations de ressources.

5.3.1. Représentation interne

Alors que dans ASA un circuit est représenté par deux structures (un flot de contrôle et un ensemble de blocs de base), MAHA figure la description du circuit à réaliser par une seule structure : un graphe de flot dont les nœuds et les arcs décrivent indifféremment les expressions calculées et le transfert de contrôle. La figure 3 est un exemple de description fourni en entrée à MAHA. Les nœuds représentés par des carrés décrivent le transfert de contrôle. D (pour Distribute) signifie distribution et J (pour Jonction) signifie point de jonction. Ces nœuds permettent de décrires les transferts de contrôle (instructions conditionnelles).

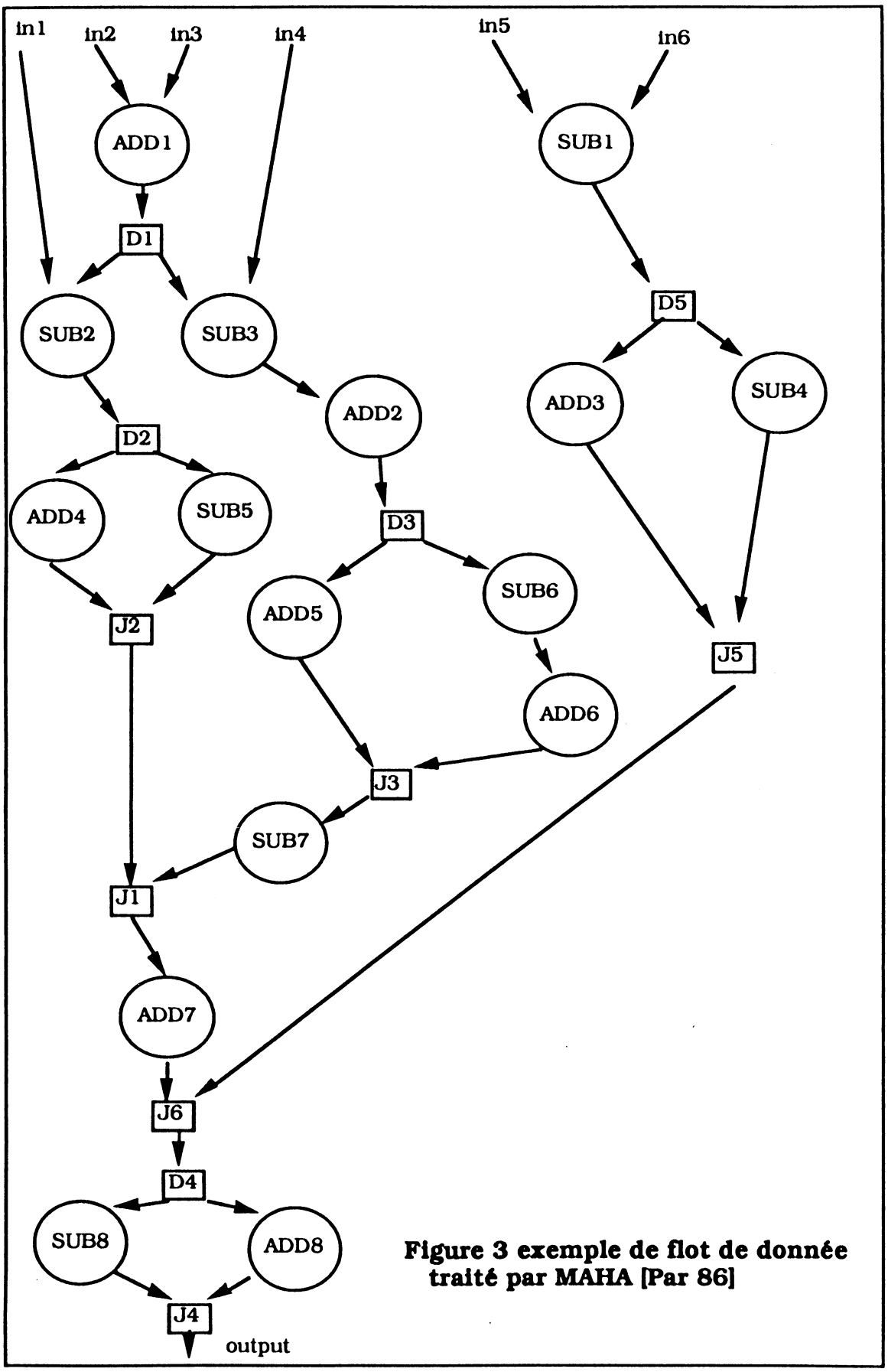


Figure 3 exemple de flot de donnée traité par MAHA [Par 86]

5.3.2. Allocation et ordonnancement de ressources fonctionnelles

MAHA s'efforce de trouver un ordonnancement des opérations du graphe satisfaisant des contraintes de temps et (ou) de surface données. L'allocation est effectuée en même temps que l'ordonnancement, des ressources supplémentaires n'étant ajoutées que lorsque les ressources courantes ne peuvent être partagées. L'ordonnancement des opérations, comme dans ASA, s'effectue en deux étapes : les nœuds critiques sont ordonnancés en priorité et ensuite les nœuds non critiques.

MAHA utilise un programme (CSSP) pour déterminer la structure temporelle du circuit à réaliser. A l'aide du CSSP, MAHA détermine d'abord le chemin critique et la période d'horloge à utiliser. Ensuite, le CSSP découpe le chemin critique en un nombre d'étapes inférieur ou égal au nombre de nœuds du chemin critique. Ce nombre d'étape est itérativement modifié pour s'adapter aux contraintes de l'utilisateur : si le circuit est lent, le CSSP essaie de réduire le nombre d'étapes (au prix d'un nombre de ressources hardware supplémentaires puisqu'il est conduit à enchaîner les opérations dans un cycle) ; si le coût en surface est dépassé, MAHA essaie de réviser à la hausse le partitionnement. En cas de conflit, l'utilisateur est consulté pour accorder la priorité à l'une des contraintes (temps ou surface).

Les nœuds non critiques sont alloués aux cycles en commençant par ceux qui ont la mobilité la plus faible. Le nœud sélectionné est alloué au premier cycle autorisé par sa mobilité qui lui permet d'utiliser une ressource fonctionnelle disponible. Lorsqu'aucune ressource n'est disponible pour exécuter l'opération du nœud, le nœud est arbitrairement affecté au premier cycle défini par sa mobilité. L'allocation est faite comme pour les nœuds critiques. Si les côuts sont dépassés, le flot de donnée doit être re-partitionné et l'allocation de toutes les ressources refaite. Après l'ordonnancement d'un nœud, MAHA recalcule la mobilité de tous les nœuds restants et le processus reprend jusqu'à ce que tous les nœuds soient ordonnancés.

Alors que ASA se limite aux schémas de séquencement ne comportant pas d'enchaînement de plusieurs opérations au cours d'un cycle, MAHA offre un éventail de structure de séquencement permettant l'enchaînement de plusieurs opérations au cours d'un cycle.

Comme ASA, MAHA effectue une analyse du chemin critique. Mais on se rend compte que MAHA dépense beaucoup d'effort pendant l'ordonnancement :

un dépassement du coût en surface le conduit à remettre en cause l'ensemble de l'allocation déjà effectuée ; d'autre part il recalcule la mobilité de tous les nœuds restant chaque fois qu'il traite un nœud. Dans ASA, les remises en cause sont localisées aux nœuds effectuant une même opération plûtôt qu'à tous les nœuds. D'autre part, la mise à jour de la mobilité des nœuds est prise en charge par des démons sous la forme d'une propagation de contraintes aux nœuds prédécesseurs et successeurs du nœud oronnancé.

Dans MAHA aucun compte n'est tenu du coût des opérateurs fonctionnels ou des besoins d'interconnexion dans le choix du nœud à ordonnancé et dans le choix du cycle à allouer. Nous avons vu au chapitre précédent que ASA s'efforçait d'exploiter les mobilités pour limiter en priorité le nombre d'unités fonctionnelles à utiliser et le nombre de transferts d'opérandes.

5.3.3. Allocation de registres

L'allocation des éléments de mémorisation dans MAHA est effectuée par REAL (REgister ALocation). REAL s'inspire d'un algorithme optimal d'allocation de pistes dans un canal de routages à un ensemble de segments de connexion, algorithme connu sous le nom "left-edge algorithm" [Has 77]. Cet algorithme procède comme suit (tiré de [Kur 87]) :

1- ordonner les segments suivant l'ordre croissant de leur bout gauche
2- allouer le premier canal au premier segment
3- trouver le premier segment dont le bout gauche est adjacent au bout droit du segment précédent et l'affecter au canal
4- si aucun autre fil ne peut être affecté au canal courant, recommencer à partir de l'étape 2 jusqu'à ce que tous les fils soient affectés à un canal.

Pour les graphes de flots ne contenant ni boucle ni d'instruction conditionnelle, le problème d'allocation des registres est identique au problème d'allocation des canaux. Les fils correspondent aux valeurs (nœuds du graphe), les bouts gauche et droit correspondent aux dates de naissance et de mort caractérisant la vivacité des valeurs ; les canaux modélisent les registres.

La présence de branches conditionnelles dans le graphe de flot rend mutuellement exclusives des valeurs dont les vivacités se recoupent pourvu que ces valeurs appartiennent à des branches conditionnelles distinctes. Pour tenir compte de ces perturbations, REAL colore les nœuds du graphe suivant un algorithme appelé MPC [Par 85]. Deux nœuds ont des couleurs distinctes s'ils

sont mutuellement exclusifs et dans ce cas, ils peuvent partager un registre. REAL exploite cette information pendant l'allocation de registres.

Kurdahi et Alice Parker pensent que REAL réalise une allocation optimale. Nous avons été un peu influencé par l'approche suivie par REAL (left-edge algorithm). Néanmoins on peut noter des différences entre REAL et l'approche suivie par ASA. REAL ne fait aucune distinction entre les variables globales et les temporaires. ASA au contraire décompose l'allocation en deux étapes : au cours de l'allocation globale, il ne s'intéresse qu'aux valeurs échangées entre les blocs (c'est à dire aux valeurs traversant les branches conditionnelles) ; l'affectation locale de temporaires est pratiquement identique au left-edge algorithm. Dans les deux cas, on se retrouve avec des problèmes de taille réduite. Alors que REAL aborde le problème des branches conditionnelles en déterminant la relation de mutuelle exclusion entre tous les nœuds du graphe, ASA partitionne les valeurs globales (celles dont la vivacité s'étend sur plusieurs blocs) en sous groupes de compatibilités. Toutes les valeurs mutuellement exclusives dont les vivacités se chevauchent deux à deux sont réunies en un groupe. L'allocation globale procède suivant le "left-edge algorithm" en assimilant chaque groupe à une valeur.

Pour l'exemple que nous avons présenté figure 3 avec un parallélisme maximal utilisant un jeu minimal de ressource, MAHA et ASA obtiennent une solution en 8 cycles utilisant un additionneur et un soustracteur. Pour une vitesse maximale, MAHA produit une solution en quatre cycles consommant deux additionneurs et trois soustracteurs. Ce dernier cas est hors de portée de ASA puisque ASA n'envisage pas les enchaînement d'opérations dans un cycle.

Pour cet exemple, nous ne disposons pas des performances de REAL.

5.4. CHIPPE

CHIPPE est un outil d'aide à la synthèse architecturale développé à l'université d'Illinois sous la direction de Gajski et F.D. Brewer. Il est conçu sous comme un ensemble de quatre systèmes experts communicant suivant le concept proposé par Gajski et Brewer [Bre 86]. Un de ces experts joue un rôle prépondérant : c'est le "design critic", seul habilité à effectuer les compromis et contrôlant l'activité des trois autres par le biais du renforcement ou du relâchement des contraintes. Les autres experts lui fournissent une évaluation des performances résultant des contraintes imposées, ce qui permet à l'audit d'ajuster ces contraintes pour la satisfaction des buts définis par l'utilisateur. La base de connaissances de l'audit est constituée de méthodes et stratégies à

suivre pour atteindre un objectif donné. Lorsqu'il y a une incertitude sur la stratégie à suivre, l'audit essaie plusieurs voies avant de décider de la stratégie à suivre. Les autres experts sont conçus pour rechercher des solutions optimales aux problèmes tout en respectant les contraintes imposées par l'audit. On retrouve dans cette approche, beacoup de similitudes avec ASA. Une place importante est accordée à la propagation des contraintes. le travail de synthèse est décomposé en plusieurs sous-tâches traitées par des portions de la base de connaissance bien distinctes. Néanmoins des différences notables subsistent. Alors que ASA décompose les tâches en sous-tâches à l'aide d'un catalogue de squelettes de plans, CHIPPE décompose plutôt les activités : plusieurs experts spécialisés décomposent l'activité de conception. Ainsi un expert s'occupe de la transformation de l'algorithme initial en une structure interne après une séquence d'optimisations adéquate, un autre effectue l'ordonnancement des opérations et le dernier l'allocation et la synthèse des interconnexions. La différence est mineure puisque à regarder de près, on se rend compte que la communication s'effectue dans une seule direction entre les trois experts (cf figure 4). Ce qui implique que les experts opèrent les uns après les autres sous la direction de l'audit. En fait, au moment de construire ODSE, nous avions étés attiré par le concept présenté par Gajski et Brewer mais nous avons reculé face à la complexité des problèmes de communication et de coopération entre plusieurs experts. Dans le cas présenté par CHIPPE, cette coopération est simple mais elle peut prendre des formes bien plus complexe[].

5.4.1. Représentation de l'algorithme d'entrée

Dans CHIPPE, langage d'entrée (inspiré de Pascal) est représenté sous une forme analogue aux blocs de base et flots de contrôle de ASA. La transformation de l'algorithme d'entrée dans cette structure interne est conduite par COMPILER.

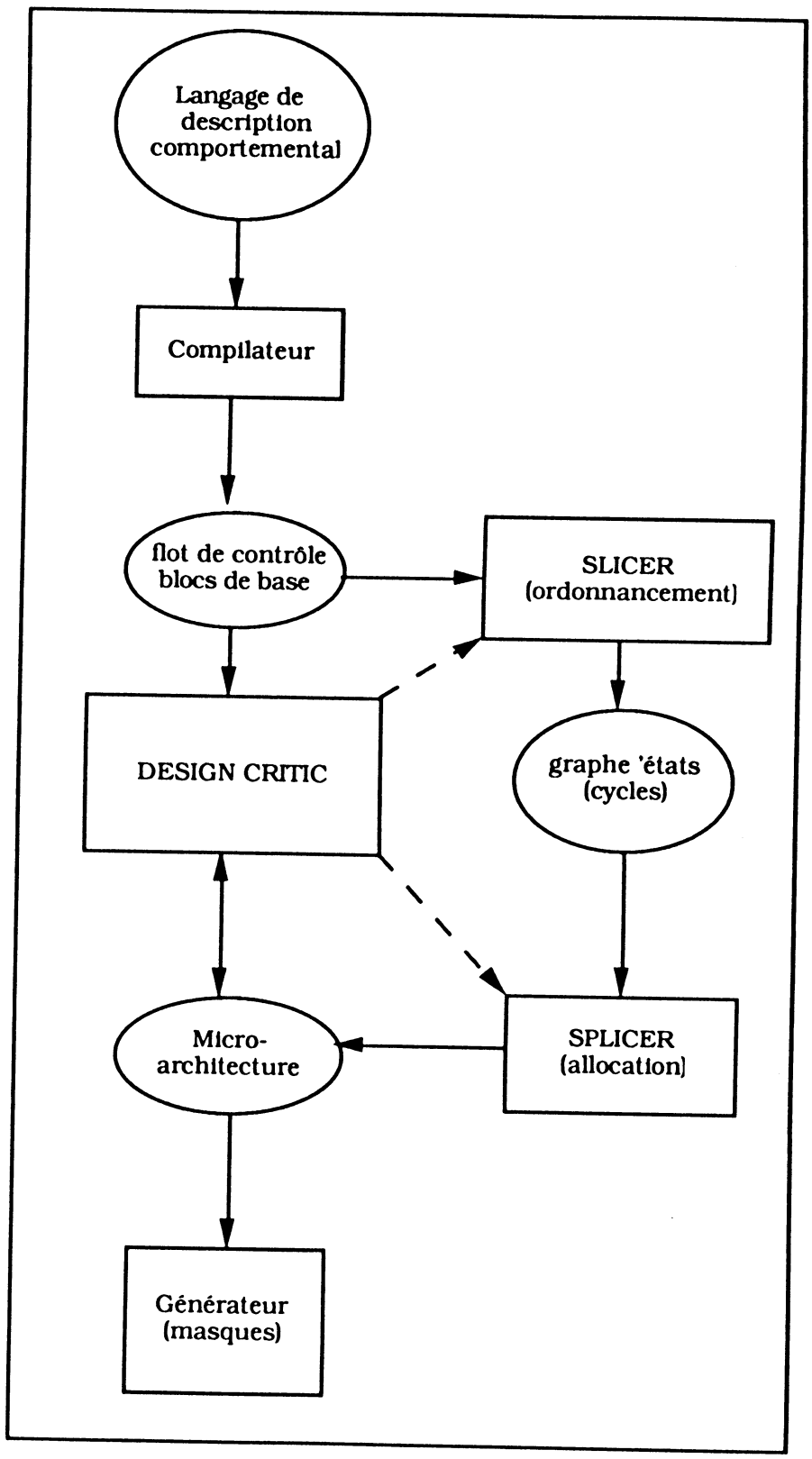


Figure 4 Structure générale de CHIPPE

5.4.2. Ordonnancement des opérations

L'ordonnancement des opérations est effectué par SLICER. Celui-ci détermine d'abord le chemin critique en calculant les dates au plus tard et les dates au plus tôt des nœuds du graphe de flot. Ensuite, il affecte les nœuds aux cycles à partir du premier jusqu'à ce que tous les nœuds soient ordonnancés. Les nœuds sont affectés aux cycles à concurence du nombre et des types d'unités fonctionnelles autorisées. Les nœuds ayant les plus faibles mobilités sont choisis en priorité. En cas de conflit, les nœuds ayant le maximum de successeurs sont retenus. Si le conflit persiste, des nœuds sont arbitrairement retenus dans ce dernier groupe. Comme ASA, SLICER tient compte du temps d'exécution des unités fonctionneles. Lorsque le temps d'exécution d'une unité fonctionnelle est plus grande qu'une période d'horloge, les opérations correspondantes sont ordonnancées sur un nombre suffisant de cycles (multicycle).

L'ordonnancement effectué par SLICER peut être remis en question par CRITIC qui peut modifier les contraintes (nombre d'opérateurs...) pour se conformer aux exigences de l'utilisateur.

On peut noter des différences avec la stratégie suivie par ASA. D'abord, l'ordonnancement s'effectue dans l'ordre chronologique des cycles. D'autre part les seuls critères retenus sont la mobilité des nœuds et le nombre de successeurs de ces nœuds. Ces critères accordent une importance égale aux opérations des nœuds et ignorent les problèmes de communication (transfert des opérandes) qui ont une incidence sur la complexité des connexions. ASA au contraire cherche en priorité à minimiser le nombre des unités fonctionnelles les plus coûteuses en temps et en surface, et s'efforce de réduire les besoins de communications en préférant pour un cycle donné, toutes choses égales par ailleurs, les nœuds dont les opérandes interviennent déjà dans les opérations des nœuds ordonnancés au cycle en question.

Il faut ajouter que SLICER intègre des particularités non encore implémentées dans ASA. Par exemple l'enchaînement de plusieurs opérations dans un même cycle lorsque la somme de leur temps d'exécution n'excède pas la durée du cycle.

5.4.3. Allocation des ressources

L'allocation des ressources ainsi que l'élaboration des interconnexions est effectuée par SPLICER qui explore suivant une stratégie ("branch and bound") en

profondeur d'abord avec retour arrière, un ensemble de solutions. Chaque solution est évaluée et la solution la moins coûteuse est retenue. L'utilisateur peut limiter le temps d'exécution de SLICER. Au terme du délai imparti, la meilleure solution retenue est proposée à l'utilisateur. Initialement, SPLICER sélectionne, dans la bibliothèque d'opérateurs, des unités fonctionnelles présentant les mêmes caractéristiques temporelles que celles supposées par SLICER au cours de l'ordonnancement. Le reste de la synthèse se déroule en quatre étapes pour chacun des cycles du circuit :

1 - allocation de bus pour le transfert des opérandes. Une approche du type glouton ("greedy") essaie d'affecter les registres aux bus déjà connectés à ces registres. Les coûts d'allocations sont évalués en termes de nombre de connexions, de bus ou de multiplexeurs supplémentaires qu'il faut pour que chaque registre soit connecté à l'unité fonctionnelle correspondante. Ce faisant, SPLICER enregistre des informations lui permettant d'éviter des configurations jugées médiocres.

2 - allocation des bus aux entrées des unités fonctionnelles. Une stratégie analogue au 1 affecte en priorité les bus aux unités fonctionnelles présentant des connexions appropriées avec les bus visés. C'est au cours de cette phase que les unités fonctionnelles sont affectées aux opérations. Les coûts d'allocation incluent les connexions et multiplexeurs supplémentaires utilisés pour lier les bus aux entrées des unités fonctionnelles.

3- allocation des bus aux sorties des unités fonctionnelles. Cette étape se déroule comme l'étape 1.

4- connexion des bus de sortie aux registres. C'est au cours de cette étape que des registres sont alloués pour conserver les informations intermédiaires. A chaque registre correspond une entrée dans une table comptant le nombre de nœuds restant utilisant le contenu du registre comme opérande. Comme précédemment, des multiplexeurs sont ajoutés suivant les besoins. Les coûts prennent en compte les temporaires, multiplexeurs et connexions supplémentaires.

Ces quatres étapes sont effectuées pour chaque cycle du circuit. Plusieurs configuration sont explorées. L'exploration s'effectue en profondeur d'abord avec retour arrière tout en évitant les configurations marquées pendant les explorations précédentes. La solution de plus faible coût est retenue. SPLICER est capable de traiter plusieurs cycles à la fois (anticipation). Un paramètre permet à l'utilisateur de fixer le nombre de cycles simultané que SPLICER doit traiter.

On peut noter plusieurs différences avec ASA. SPLICER (comme SLICER d'ailleurs) est conçu comme un programme plutôt qu'un système à base de règles. L'allocation des temporaires est faite de manière dynamique alors que ASA effectue une analyse de la vivacité des variables. L'espace de recherche exploré a une croissance exponentielle en fonction du jeu de ressources fonctionnelles utilisées. On peut craindre une baisse de performance pour des circuits de complexité importante.

Pour le circuit EQUADIFF et dans des conditions identiques, CHIPPE utilise le même jeu de ressources fonctionnelles que ASA, un registre de plus que ASA et six multiplexeurs pour un total de 14 entrées alors que ASA utilisait 6 multiplexeurs à deux entrées.

5.5. Conclusion

A travers les quelques exemples que nous avons vu, on remarque que ASA obtient des performances comparables (sinon meilleures) à celles de ces systèmes. Les exemples que nous avons présentés sont de faible complexité et ne suffisent pas pour porter un jugement définitif bien que ces exemples soient ceux proposés par les auteurs de ces systèmes. Néanmoins, les résultats obtenus sont encourageants pour la suite du prototype. Nous terminerons le chapitre en présentant les performances de ASA pour la synthèse d'un filtre LEAPFROG[Bal 89]. La figure 5 donne un schéma du filtre. La description ASA du filtre est donnée ci-dessous :

```
circuit leapfrog;
vin,v1,v2,v3,v4,v5,v6,v7,v8,v9,a1,a2,
      a3,a4,a5,a6,a7,a8,a9:integer;
begin
 loop
      v1:=v1+a1*(v1+v2);
      v3:=v3+a3*(v2+v4);
      v5:=v5+a5*(v4+v6);
      v7:=v7+a7*(v6+v8);
      v9:=v9+a9*(v8+v9);
      v2:=v2+a2(v1+v3);
      v4:=v4+a4*(v3+v5);
```

```
        v6:=v6+a6*(v5+v7);
        v8:=v8+a8*(v7+v9);
    end loop;
    end leapfrog;
```

Pour la synthèse, ASA dispose d'une bibliothèque contenant des registres, multiplexeurs à deux entrées, additionneurs et multiplieurs. Les additionneurs ont un temps de réponse de 30ns pour 32 bits et les multiplieurs 90ns.

Nous avons considéré 6 jeu de contraintes différentes :

—1) une période d'échantillonage du filtre de1200 ns pour une période d'horloge de 100ns ;

—2)une période d'échantillonnage du filtre de 1200ns pour une horloge de 50ns ;

—3) un jeu de ressources restraint à 1 multiplieur et un additionneur pour une horloge de 100ns ;

—4) un jeu de ressources restraint à 1 multiplieur et un additionneur pour une horloge de 50ns ;

—5) un jeu de ressources restraint à 1 multiplieur et 2 additionneurs pour une horloge de 100ns ;

—6) une fréquence de fonctionnement maximale avec une horloge de 100ns ;

—7)une fréquence de fonctionnement maximale avec une horloge de 50ns.

La figure 7 résume les caractéristiques obtenues pour les divers cas mentionnés.

Les solutions sont obtenues en un temps CPU de 20 minutes environ sur un ordinateur MICROVAX. Le circuit le plus rapide a une période d'échantillonnage de 450ns, et la plus lente 1900ns. La première utilise 5 multiplieurs et 5 additionneurs pour une horloge de 50ns tandis que la seconde se contente de 1 multiplieur et 1 additionneur pour une horloge de 500ns. Les besoins en éléments de mémorisation sont les mêmes pour toutes les hypothèses retenues : 15 registres dont 5 temporaires. On peut remarquer l'influence de la période d'horloge sur les performances de l'architecture produite. Avec un même jeu de ressources, on obtient une amélioration d'environ 70% en passant d'une horloge à 100ns à une horloge à 50ns.

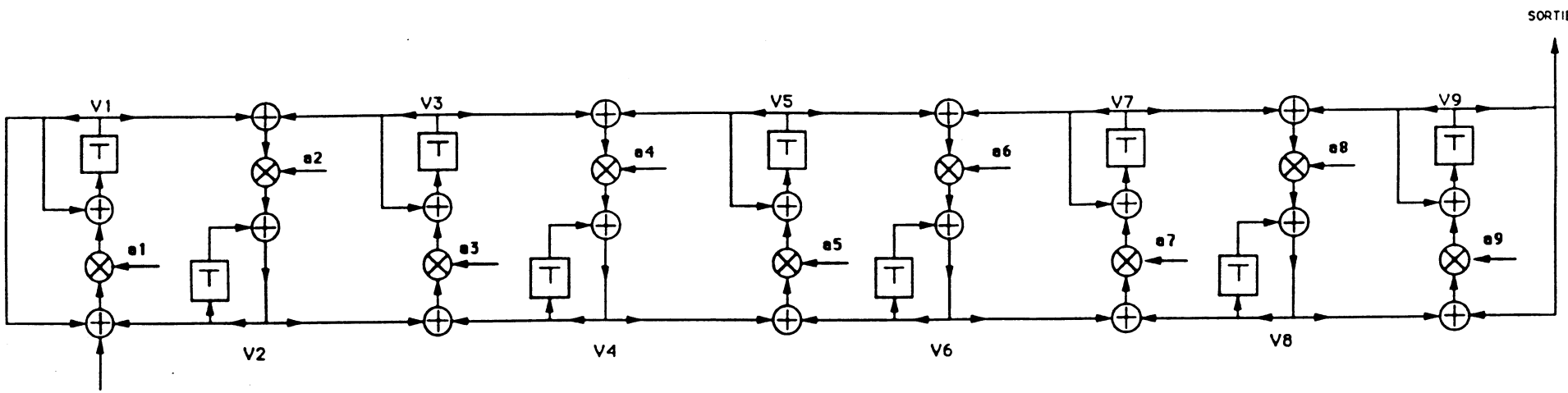


Figure 5 :Filtre Leapfrog(tiréde [Bal 89])

Contraintes	temps cpu	période horloge (ns)	Nb mult.	Nb reg	Nb add.	Nb mux	Nb cycles	temps total
temps<= 1200ns	18'35"	100ns	1	15 +9 constantes	2	24	12	1200ns
temps<= 1200ns	21'32"	50ns	1	15 +9 constantes	1	4 bus 6 MUX	20	1000ns
ressources : 1 multiplieur 1 additionneur	19'09"	100ns	1	15 +9 constantes	1	29	19	1900ns
ressources : 1 multiplieur 1 additionneur	19'10"	50ns	1	15 +9 constantes	1	27	21	1050ns
ressources : 1 multiplieur 2 additionneurs	16'12"	100ns	1	15 +9 constantes	2	27	11	1100ns
vitesse maximale	16'36"	100ns	3	15 +9 constantes	4	28	7	700ns
vitesse maximale	19'04"	50ns	5	15 +9 constantes	5	22	9	450ns

figure 6 :Solutions produites par ASA pour le filtre Leapfrog

6. Conclusion générale

Les principales contributions de cette thèse sur l'application des techniques d'intelligence artificielle à la synthèse architecturale de circuits intégrés peuvent se résumer par les points suivants :

—les concepts exposés dans cette thèse ont été validés par la construction d'un système expert d'aide à la synthèse architecturale à partir d'un moteur d'inférence que nous avons également construit ;

—s'inspirant des concepts de résolution de problème et de planification hiérarchique, le moteur d'inférence élabore la solution à un problème donné par affinements successifs en se conformant aux directives exprimées par des squelettes de plan judicieusement sélectionnés ;

—ces squelettes de plans constituent une hiérarchie de règles décrivant l'activité de synthèse architecturale sous la forme d'un ensemble de recommandations ou de stratégies inspirées par la démarche des concepteurs humains. Ainsi le moteur n'a pas besoin de chercher une solution : il lui suffit de sélectionner un plan approprié au contexte et d'en suivre les directives ;

—le prototype contient un ensemble de 500 règles environ permettant la synthèse d'une architecture de circuit à partir d'une spécification fonctionnelle assortie d'un jeu de contraintes. L'architecture est obtenue par assemblages de composants tirés d'une bibliothèque de modules. Dans la version actuelle, la partie opérative de cette architecture est générée par SPOT, un générateur de chemins de données en tranches développé au CNET Grenoble ;

—les premiers résultats obtenus sur les banc d'éssai de quelques systèmes concurrents sont encourageants

6.1. Perspectives à court terme

Les perspectives à court terme se divisent en deux catégories :

—en ce qui concerne moteur d'inférence, il s'agit de réaliser le module de compilation de la base de règles qui permettrait d'affranchir le temps d'évaluation d'un ensemble de règles du nombre de règles comme nous l'avons vu au chapitre 3.

—pour ce qui est du système expert ASA, il s'agira d'étendre la base de règles pour lui permettre d'effectuer plusieurs optimisations : fusion des unités fonctionnelles en UAL, regroupement de temporaires en RAM,etc.

6.2. Perspectives à long terme

A long terme, nous distinguerons également deux directions :

—on peut adjoindre un système de maintenance de la cohérence de la base de faits à ODSE. Comme nous l'avons vu au chapitre 4, c'est un système qui associe à chaque fait de la base, les raisons de sa présence. Lorsqu'un fait change, il peut entrainer la modification de la valeur de vérité de tous les faits qui dépendent de lui. Ce système serait très utile pour accélérer l'exploration de plusieurs solutions architecturales par la simple modification des contraintes.

—au niveau du système expert ASA, on peut envisager d'étendre ses compétences à la synthèse des architectures pipeline ou systoliques. Ici aussi, on peut s'inspirer des travaux effectués en analyse des flots de programmes pour déterminer les optimisations et les découpages à effectuer. Pour les architectures pipeline, nous pensons à des techniques comme le déroulement de boucles, l'optimisation des appels et des corps de procédures, la fusion de plusieurs blocs en un seul etc.

Bibliographie

[Aho 84] A.V. AHO, R. SEHTI,J.D. ULLMAN
"Compilers : Principles, Techniques and Tools"
Addison-Wesley, Reading,Mass., 1986

[Ali 87] F. ALIZON, G. HUET
"OSMOSE : Un Environnement de programmation SMALLTALK destiné à la Construction de Systemes Experts"
NOTE TECHNIQUE NT/LAA/SLC/280 Octobre 1987

[Anc 84] François ANCEAU
" The architecture of micro processors "
ADDISON-WESLEY 1984

[Bal 89] F.BALESTRO, G. PRIVAT, M.S. TAWFIK
"A Bit-Serial Approach to VLSI Implementation of Digital LDI Ladder Filter"

[Bar 83] G. BARBERYE, T.JOUBERT, M.MARTIN
"Manuel d'utilisation de PROLOG/CNET"
NOTE TECHNIQUE NT/PAA/CLC/LCS/1058

[Ber 84] J.M.BERGE ,L-O.DONZELLE, V.OLIVE,
J. ROUILLARD, D. ROUQUIER
"LOF : a Building Tool for Flexible Blocks Libraries"
Proc IEEE international conference on computer desugn
(ICCD 84)

[Ber 89] J.M. BERGE, L.-O. DONZELLE, V. OLIVE,
J. ROUILLARD
"ADA avec le sourire"
Presses Polytechniques ROMANDES 1989

[Boo 83] Grady BOOCH
"Software Engeneering with ADA"
The Benjamin/Cummings Publishing Company, Inc. 1983

[Bre 86] F.D. BREWER, D.D. GAJSKI
"An Expert-System Paradigm for Design"
23rd Design Automation Conference 1986

[Bre 87] F.D. BREWER, D.D. GAJSKI
"Knowledge Based Control in Micro-Architecture Design"
24th ACM/IEEE Design Automation Conference 1987

[Cha 84] E. CHARNIAK, D. V. McDERMOTT,
"Introduction to Artificial Intelligence"

[Cha 87] E. CHARNIAK, C. K. RIESBECK, D. V. McDERMOTT,
J. R. MEEHAN
"Artificial Intelligence Programming"
LAWRENCE ERLBAUM ASSOCIATES PUBLISHERS 1987

[Cam 89] R. CAMPOSANO, W. ROSENTIEL
"Synthesizing Circuits From behavioral Descriptions"
IEEE Transactions on CAD Vol. 8 No 2 Feb. 1989

[DeM85] H.De Man
"Evolution of CAD Tools towards third generation custom VLSI Design "

[Des 85] Y. DESCOTTE, J.C. LATOMBE
"Making Compromises among Antagonist Constraints in a Planner"
ARTIFICIAL INTELLIGENCE (1985) 27 pp183-217

[Doy 79] J. DOYLE
"Truth Maintenance System"
Artificial Intelligence 1979, vol 12

[Du 88] Yves DURAND
"Expériences en synthèse logique"
Thèse de docteur INPG. Grenoble 1988

[Haz 87] E.T. HAZEM
"FIDEL : Un langage de description et de simulation des circuits VLSI"
Thèse de docteur INPG, Grenoble 1987

[Far 87] Henri FARRENY, Malik GHALLAB
"Eléments d'Intelligence Artificielle"
HERMES 1987

[Fla-84] Eric FLAMAND
"Etude d'outils de synthèse automatique"
Rapport de D.E.A. informatique de l'I.N.P.G. Juin 84.

[For82] Charles L. FORGY
"Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem"
Artificial intelligence 19 (1982) pp 17-37

[Fuj 86] T. FUJITA, H. HAJIMU, K. MITSUMOTO, S. GOTO

" Artificial Intelligent Approach to VLSI Design "
Design Methodologies pp 441-464
Elsevier Science Publishers B.V. North-Holland 1986

[Gae-88] **Gaetano BORRIELLO, Edwald DIETJENS**
"High level Synthesis : Current Status and Future Directions"
25th ACM/IEEE Design Automation Conference 1988

[Gir 85] **E.F.GIRCZY, R.J.A. BUHR, J.P. KNIGHT**
"Applicability of a subset of ADA as an Algorithmic Hardware Description Langage for Graph-Based Hardware Compilation"
IEEE TRANS.ACTION C.A.D. vol.4, No.2, April 1985

[Gol 83] **Adele GOLBERG, David ROBSON**
"Smalltalk-80 The Language and its Implementation"
Addison-Wesley Publishing Company 1983

[Has 77] **A. HASHIMOTO, J. STEVENS**
"Wire Routing by Optimizing Channel Assignment Within Large Apertures."
Proceedings 8th DA workshop, pages 155-169, IEEE, 1971

[Hay 85] **F. HAYES-ROTH**
"Rule-Based Systems"
Communications of the ACM Sept. 85, Vol. 28, Number 9.

[Hen 85] **HENNION, P. SENN**
"ELDO : a new third generation circuit simulator using the one step relaxation method"
Proc. IEEE Int. Symp. on circuits and systems(ISCAS 85)

[Hor86] **E.Hörst**
" Logic Design and Simulation"
Advances in CAD for VLSI, Volume 2 North-Holland

[Ili 87] **Herman ILINE, Henry KANOUI**
"Extending Logic Programming to Object Programming : The system LAP"
IJCAII 87

[Kna 86] **D. W. KNAPP, A.C. PARKER**
"A Design Utility Manager : the ADAM Planning Engine"
23rd Design Automation Conference 1986

[Jer-86] **A. JERRAYA, P. VARINOT,R.JAMIER, B. COURTOIS**
"Principles of SYCO Compiler"
23rd Design Automation Conference p 715,721

[Kim 83] **J. KIM, J. McDERMOT**

"TALIB : An IC Layout Design Assistant"
Proc. AAAI, 1983, pp.197-201

[Kes 85] A.J.Kessler and A.Ganesan
"Standard cell VLSI Design : a Tutorial "
IEEE Circuits and Device Magazine Jan.1985

[Kor 87] R.E. KORF
"Planning as Search : A Quantitative Approach"
Artificial Intelligence 33(1987)pp65-88]

[Kow 85] T. J. KOWALSKI, D.J. GEIGER, W.H. WOLF, W. FICHTNER
" The VLSI Design Automation Assistant : From Algorithms to Silicon "
IEEE Design & Test August 1985

[Kna 86] D.W. KNAPP, A.C. PARKER
"A Design Utility Manager: the ADAM Planning Engine"
23rd IEEE Design Automation Conference

[Kur 87] F. J. KURDAHI, A.C. PARKER
"REAL : A Program for Register Allocation"
24th ACM/IEEE Design Automation Conference

[Lai 87] J. E. LAIR, A. NEWELL, P.S. ROSENBLOOM
"SOAR: An Architecture for General Intelligence"
Artificial Intelligence 33(1987)1-64

[Mar 88] S. MARCUS, J. STOUT, J. McDERMOTT
"VT : An Expert Elevator Designer That Uses Knowledge-Based Backtracking"
A.I. MAGAZINE Spring 1988 pp 95-111

[McF 88] M.C. McFARLAND A.C. PARKER, R. CAMPOSANO
"Tutorial on High Level Synthesis"
25th ACM/IEEE Design Automation Conference 1988

[McJ 82] John McDERMOTT
"R1: A Rule-Based Confogurer of Computer Systems"
Artificial Intelligence 19 (1982) pp 39-88

[Mea 79] C. MEAD, L. CONWAY
"Introduction to VLSI systems"
Reading, M.A : Addison-Wesley,1979

[Meh 83] Mehmet DINCBAS
"Contribution à l'étude des systèmes experts"

Thèse de docteur ingénieur Sup.Aero 1983

[Mea-80] Carver MEAD and Lin CONWAY
"Introduction to VLSI design"
Addison Wesley Editor, 1980

[Na 82] A.W. NAGLE, R.CLOUTIER, A.C. PARKER
"Synthesis of Hardware for the Control of Digital Systems"
IEEE Transactions on CAD of circuits and systems vol. 1, No 4 October 1982

[Obr 82] M. OBRESKA
"Etude comparative de différentes méthodes de conception des parties contrôle de mico-processeurs"
Thèse Docteur-Ingénieur, INPG, Grenoble 1982

[Oli 86] Vincent Olive
"Programmation d'automates sur silicium "
Thèse de doctorat I.N.P.G.86

[Pan 87] B. M. PANGRE, D.D. GAJSKI
"Design Tools for Intelligent Silicon Compilation"
IEEE TRANSACTIONS ON C.A.D. November 1987

[Par86] A.C. PARKER, J.T. PIZARRO, Mitch MLINAR
"MAHA: A program for Datapath Synthesis"
23rd IEEE Design Automation Conference

[Par 85] N.PARK.
"Synthesis of high speed digital systems"
PhD dissertation, University of Soputhern California

[Pau86] P.G. PAULIN, J.P. KNIGHT, E.F. GIRCZYC
"HAL : A Multi-Paradigm Approach to Automatic Datapath Synthesis"
23rd IEEE Design Automation Conference 1986

[Pau87] P.G. PAULIN, J.P. KNIGHT
" Force- Directed Scheduling in Automatic Datapath Synthesis "
24th IEEE Design Automation Conference 1987

[Pea 84] Judea PEARL
"HEURISTICS
Intellient Search Strategies for Computer Problem Solving"
Addison-Wesley Publishing Company 1984

[Rob 89] Anne ROBERT

"Générateur de Chemins de Données"
Note interne C.NE.T.

[Rou 88] J.ROUILLARD
"Manuel d'utilisation du générateur d'analyseur syntaxique SIL"
Note interne CNET

[Ryd 86] B.G. RYDER, M. C. PAUL
"Elimination Algorithms for Data Flow Analysis"
ACM Computing Surveys vol.3, No.3, September 1986

[Rey 85] Jean Frédéric REYSS-BRION
" IMHOTEP : Un générateur automatique d'architecture pour circuits intégrés de filtrage numériques "
Thèse de docteur de l'I.N.P.G. Mai 85

[Ryd 86] B. G. RYDER, M. C. PAULL
"Elimination Algorithms for Data Flow Analysis"
A.C.M. Computing Surveys, Viol.18,No.3, September 1986

[Sac 74] Earl D. SACCERDOTI
"Planning in a hierarchy of Abstraction Spaces"
ARTIFICIAL INTELLIGENCE 5 (1974), pp115-135

[Sah 80] Sarta SAHNI, Atal BHATT
"The complexity of Design Automation Problems"
Design Automation Conference 1980

[Sis 82] Jeffrey Mark Siskind,Jay Roger Southard
"Generating Custom High performance VLSI Designs from succint Algorithmic Descriptions"
1982 Conference on Advanced Research in VLSI, M.I.T.

[Ste 81a] Mark STEFIK
"Planning with Constraints (MOLGEN : part 1)"
ARTIFICIAL INTELLIGENCE 16 (1981) pp111-140

[Ste 81b] Mark STEFIK
"Planning and Meta-Planning"
ARTIFICIAL INTELLIGENCE 16 (1981) pp141-170

[Ste 86] Mark STEFIK, Daniel BOBROW
"Object-Oriented Programming: Themes and Variations"
The AI MAGAZINE 86

[Tag 87] A.-M. TAGANT, L.-O. DONZELLE, A. PUISSOCHET

"Rapport d'évaluation du système SMALLTALK sur machine TEKTRONIX 4405"
NOTE TECHNIQUE NT/CNS/CCI/62 février 1987

[Tri 87] **H. TRICKEY**
"Flamel : A High-Level Hardware Compiler"
IEEE Transactions on CAD CAD-6, 2 (March 1987)

[Tse 83] **C.J. TSENG, D. P. SIEWIOREK**
"FACET : A Procedure for the Automated Synthesis of Digital Systems"
20th Design Automation Conference 1983 IEEE

[Tse 88] **C.-J. TSENG, R.-S. WEI, G. ROTHWEILER, M.M. TONG,A.K. BOSE**
"BRIDGE : A Versatile Behavioral Synthesis System"
25th ACM/IEEE Design Automation Conference

[Sta 77] **Richard M. STALLMAN, Gerald J. SUSSMAN**
"Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis"
ARTIFICIAL INTELLIGENCE 9 (1977) pp135-196

[Ros 87] **Jean-Pière ROSEN**
"Méthode de Conception Orientée Objets"
GENIE LOGICIEL N°9 Novembre 1987

[Rou 89] **D. Rouquier**
"La Compilation de Silicium"
L'Echo des Recherches No 135 1er trimestre 1989

[Wec 86] **T. WECKER**
"Semi- Custom Design Systems"
Logic Design and Simulation ED. E. HORBST
ELSEVIER PUBLISHER B.U.
North Holland, 1986, pp 195-228

[Wil 84] **David E. WILKINS**
"Domain- Planning : Representation and Plan Generation"
ARTIFICIAL INTELLIGENCE 22 (1984) pp269-301

Annexe A : Syntaxe des commandes de ODSE

Dans cette annexe, nous décrivons la syntaxe complète des expressions, des actions primitives, des règles et des commandes acceptées par ODSE. Cette syntaxe est décrite sous la forme BNF :

les éléments non terminaux du langage sont encadrés par < et > exemple *<class>*

les symboles terminaux du langage sont exprimés en gras

[<element>] exprime zéro ou une occurence de *<élément>*

*(<élément>)** représente zéro une ou plusieurs occurences de *<élément>*.

A.1. Syntaxe des expressions

A.1.1. Valeur de l'attribut d'un objet de la base

Syntaxe :

<valeur attribut> **:** **:=** ***[(<class><nom>)<attr>]***

Description :

<class> désigne la classe de l'objet ;

<nom> désigne le nom de l'objet au sein de cette classe. *<nom>* peut être une expression dont l'évaluation doit fournir un identificateur ;

<attr> désigne le nom de l'attriobut dont on veut la valeur.

Lorsque l'attribut est défini, l'expression s'évalue à la valeur de cet attribut. Dans le cas contraire, l'évaluation fournit *()* désignant une expression vide.

restrictions :

l'évaluation de cette expression réussit si :

—la classe d'objet *<class>* est définie (cf commande ***def_obj***)

—*<nom>* représente le nom d'un objet de classe *<class>* présent en dans la base des faits.

Dans tout autre cas l'évaluation lève une exception qui interrompt l'opération en cours d'exécution au moment de l'évaluation. Cette exception signale la non existence de l'objet référencé.

exemples :

[(resource add1) surface]

retourne la valeur de l'attribut surface de l'objet *(resource add1)*.

[(famille[(node node1) operation]) time]

s'évalue à la valeur de l'attribut *time* de l'objet de la classe *famille* dont le nom est défini par l'attribut operation du nœud node1. Si *[(node node1)operation]= plus*, l'expression est égale à la valeur de *[(famille plus)time]*.

A.1.2. Expressions numériques

syntaxe :

<expression numerique> : :=

*<terme> {<operateur multiplicatif> <terme>}**

<operateur multiplicatif> : :=

*

/

<terme> : :=

*<facteur> { <operation additive><facteur>}**

<operation additive> : :=

+

-

<facteur> : :=

<nombre>

[-]<valeur attribut>

[-]<contrainte attribut>

(<expression>)

Description :

Le système est capable d'évaluer toute expression numérique satisfaisant à la syntaxe ci dessus donnée.

<valeur attribut> s'évalue à la valeur de l'attribut désigné et *<contrainte attribut>* a la valeur de la contrainte en question (cf § contraintes) ;

lorsqu'une erreur est détectée, l'opération en cours d'exécution au moment de l'évaluation est arrêtée et un message d'erreur transmis à l'utilisateur.

Restrictions :

<valeur attribut> et *<contrainte attribut>* doivent s'évaluer à une valeur numérique sinon une exception est levée qui interompt de manière radicale l'exécution de l'opération en cours ;

Exemples :

[(resource add1)surface] +[(resource add2)surface]

s'évalue à la somme des attributs surface des objets *add1* et *add2* de la classe *resource*

3[(bloc a)haut]*

donne le résultat de la multplication de [(bloc a)haut] par 3.

A.1.3. Liste de symboles

Syntaxe :

<liste> : :=

<valeur attribut>

<contrainte attribut>

'({<atome>})*

<specif ensemble>

<atome> : :=

<identificateur>

<nombre>

<specif ensemble> : :=

(<class> [in <expression symbolique>] [/<lconditions>])

<lconditions> : :=

*<condition attribut> {,<condition attribut>}**

<condition attribut> : :=

<attr> <condition> <expression>

<condition> : :=

=

≠

>

<

>=

<=

in_set

not_in_set

Description :

une expression est soit une liste énumérée de symboles, soit la valeur ou une contrainte du domaine des valeurs d'un attribut, ou encore un ensembles de symboles identifiant la liste des objets d'une classe donnée et satisfesant à des conditions précises. Dans ce dernier cas, l'évaluation se fait en recherchant dans la base des faits, la liste des noms de tous les objets de l'ensemble spécifié.

Restrictions :

<valeur attribut> doit désigner la valeur d'un attribut multivalué ;

<contrainte attribut> doit être une contrainte du type ***in_set*** ou ***not_in_set*** ;

exemples :

(bloc)

retourne les noms des objets de la classe *bloc*

'(a e i o u y)

désigne la liste des symboles entre les parenthèses.

(resource/ time<600)

s'évalue à la liste des noms des objets de la classe *RESOURCE* dont l'attribut ***time*** est strictement inférieur à 600.

A.1.4. Operations sur les listes

Syntaxe :

<expression sur listes> : :=

*<fusion listes>(**and** <fusion listes>)**

<fusion listes> ***xor*** *<fusion listes>*

<fusion listes>\<fusion listes>

<atome>

<liste>

(first *<liste>)*

(rest *<liste>)*

(préfix *<expression symbolique>* *<liste>)*

<fusion listes> : :=

*<liste>(**or** <fusionliste>)**

(<expression symbolique>)

Description :

dans la syntaxe, ***and*** et ***or*** désignent respectivement l'union et l'intersection de listes de symboles ;

xor désigne la différence symétrique de deux listes et \ la différence de deux listes ;

first et ***rest*** correspondent aux célèbres fonctions lisp car et crd s'évaluant respectivement au premier terme d'une liste ou la liste privée du premier élément ;

préfix permet de construire une liste en ajoutant un élément à en tête d'une autre liste.

Restrictions :

les opérations ***first rest or and*** portent nécéssairement sur des listes ; tout paramètre atomique provoque l'échec de l'opération en cours et l'évaluation est arrêtée ;

first requiert une liste non vide ;

Exemples :

*(**prefix** a '(b c d))*

s'évalue à la liste *(a b c d)*

*(**rest** '(b c d))*

retourne la liste *(c d)*.

'(a e i o u) \'(i o u c)

donne la liste *(a e)*

*'(a e i o u) **xor** '(i o u c)*

renvoie la liste *(a e c)*

expression symbolique

Syntaxe :

<expr> : :=

<expression listes>

<expression numerique>

***(#** <liste>)*

Description :

l'évaluation d'une expression doit fournir un symbole (numérique ou non), ou une liste de symboles ;

***(#** <liste>)* est une fonction qui retourne la longueur de la liste

A.1.5. Contraintes sur un attribut

Syntaxe :

<contrainte attribut> : :=

<constrainte>[(<class><nom>)<attr>]

<constrainte> : :=

in_set

not_in_set

lower_bound

upper_bound

Description :

s'évalue à la contrainte courante de type *<contrainte>* qui restraint le domaine des valeurs possibles de l'attribut ;

<contrainte> a les significations suivantes dépendant de sa forme :

in_set : domaine dans lequel *<attr>* doit prendre ses valeurs

not_in_set : ensemble des valeurs exclues du domaine de *<attr>*

lower_bound : borne inférieure du domaine de *<attr>*

upper_bound : borne supérieure de *<attr>*

lorsque la contrainte désigée n'est pas définie pour <attr>, l'expression s'évalue à l'ensemble vide ;

exemples :

upper_bound*[(bloc B)haut]*

retourne la borne supérieure de l'attribut haut de l'objet *(bloc B)*.

in_set*[(node node1)operation]*

retourne le domaine dans lequel l'attribut *operation* de l'objet *(node node1)* est astreint à prendre ses valeurs.

A.1.6. conclusion

Nous avons présenté les expressions acceptées par la version actuelle de ODSE. Cette liste n'est pas exhaustive, et nous comptons l'enrichir au gré des besoins ; dans la suite de cette annexe, *<exp>* pourra désigner indifféremment une expression numérique ou symbolique ou une liste ; le type de l'expression requis sera déduit du contexte ;

A.2. construction et de modification de la base de faits

A.2.1. definition des attributs

Syntaxe :

<def attr> : :=

DEF_ATTR *<identificateur> <mode> <domaine>*

<identificateur> : :=

*lettre (lettre +chiffre)**

<mode> : :=

singlevalued

multivalued

<domaine> : :=

<liste>

<nombre> <nombre>

atom

A.2.2. definition des classes d'objets

Syntaxe :

<definition classe> : :=

def_obj *<identificateur1> ((<identificateur2>)*)*

A.2.3. Opérations sur les objets

Dans cette partie, nous passerons en revue toutes les opérations permettant de créer, modifier et supprimer un objet. Certaines operations sur les objets sont susceptibles de lever des exceptions . Dans certains cas le système permet à l'utilisateur de traiter ces exceptions par l'exécutions de démons appropriés (cf démons).

creation : make

Syntaxe :

<création> : :=

(make *(<class><nom>) ((<attr><expression>)*)*

Description :

permet de créer un objet de classe *<class>* de nom *<nom>* . Les couples *(<attr><exp>)* représentent les valeurs initiales des attributs correspondants. Noter qu'un objet est identifié par sa classe et son nom.

restrictions :

Avant toute création d'objet, il faut que la classe correspondante de l'objet soit définie sinon l'exception ***undefined_class*** est levée.

deux objets d'une même classe ne peuvent avoir le même nom. Une exception est levée chaque fois que l'on essaie de creer un objet déja existant (cf exceptions).

les valeurs affectées aux attributs doivent satisfaire aux contraintes limitant leur domaine sinon l'exception correspondante est levée.

modification : modify

Syntaxe :

<modificationd'objet> : :=

***(modify** (<class><nom>) ((<attr><exp>))*)*

modifie les caractéristiques de l'objet *(<class><nom>)* en affectant les valeurs <exp> aux attributs *<attr>* correspondant. Les valeurs précédentes des attributs *<attr>* sont remplacées par les *<exp>*.

restrictions :

l'objet *(<class><nom>)* doit exister préalablement à toute modification sinon une exception est levée.

les valeurs affectées aux attributs doivent satisfaire aux contraintes limitant leur domaine sinon l'exception correspondante est levée.

modification d'attributs multivalués : augment

Syntaxe :

<ajout à un objet> : :=

***(augment** (<class> <nom>) ((<attr> <exp>))*)*

Description :

Ajoute à chaque attribut *<attr>* la valeur *<exp>* correspondante. Les valeurs précédentes de *<attr>* sont conservées.

Restrictions :

l'objet *(<class><nom>)* doit exister avant la modification ;

les attributs *<attr>* doivent être multivalués.

Exemples :

supprimer la valeur d'un attribut : retract

Syntaxe :

<retrait d'un objet> : :=

***(retract** (<class><nom>) ((<attr><exp>))*)*

Description :

Supprime les valeurs de *<exp>* de la liste des valeurs des attributs *<attr>* correspondant (si ces valeurs font partie de cette liste).

Restrictions :

l'objet *(<class><nom>)* doit évidement exister avant l'opération sinon une exception est levée.

Si un attribut est contraint à prendre une valeur précise, on ne peut supprimer cette valeur. Il faudrait au préalable relacher les contraintes (cf operations sur les contraintes)

supprimer un objet de la base : remove

Syntaxe :

<destruction objet> : :=

(remove (<class><nom>))

Description :

Supprime l'objet *(<class><nom>)* de la base.

Restrictions :

l'objet doit exister dans la base avant l'exécution de l'opération sinon une exception est levée.

Contraindre une variable : add_constraint

Syntaxe

<contraindre objet> : :=

(add_constraint (<class><nom>) ((<attr><cond><exp>))*)

<condition> : :=

<

<=

>

>=

=

≠

in_set

not_in_set

Description :

pour chaque triplet *(<attr><con><exp>)*, cette opération réduit le domaine des valeurs de l'attribut *<attr>* de l'objet *(<class><nom>)* dans le sens indiqué. La contrainte n'est prise en compte que si elle renforce les contraintes existantes pesant sur l'attribut. Il se peut qu'une contrainte astraigne un attribut à une valeur précise, auquel cas cette valeur est affectée à l'attribut en question.

Si *<cond>*= ***in_set***, si <attr> n'a aucune contrainte sur son domaine de valeurs, *<expr>* devient son domaine ; si *<attr>* est contraint, le nouveau domaine de <attr> est calculé par :

*in_set(<attr>) : = **in_set**(<attr>) * (<expr> - **not_in_set**(<attr>)).*

***in_set**(<attr>)* représente le domaine de *<attr>* c.a.d. l'ensemble des valeurs que peut prendre l'attribut *<attr>*.

***not_in_set**(<attr>)* represente l'ensemble des valeurs interdites à l'attribut *<attr>* au moment de l'exécution de l'opération. Cet ensemble des valeurs interdites est contruit par la contrainte **not_in_set** (cf § suivant) ;

* représente l'opération d'intersection de deux ensemble ;

- représente l'opération de différence de deux ensembles ;

si *<cond>* = ***not_in_set***, on a les modifications éventuelles suivantes :

not_in_set(<attr>) : = not_in_set(<attr>) + <expr>.

in_set(<attr>) : = in_set(<attr>) +<expr>.

+ représentant ici l'union de deux ensembles

Restrictions :

l'objet sur qui l'opération porte doit exister au préalable ;

selon les cas, *<exp>* doit s'évaluer à un entier ou à un ensemble de valeurs ;

toute nouvelle contrainte doit être compatible avec les contraintes existantes sinon la nouvelle contrainte est rejetée et une exception est levée ;

dans le cas où l'attribut a des valeurs, ces valeurs doivent être compatibles avec la nouvelle contrainte sinon la nouvelle contrainte est rejetée et une exception est levée ;

Exemples :

```
*(add_constraint (bloc machin) (longueur <= 24))!
*
```

contraint l'attribut longeur à prendre des valeurs <=24

```
*(add_constraint (bloc machin) (longueur < 50))
*
```

aucun effet sur *(bloc machin)* puisque la précédente contrainte implique celle-ci

```
*(add_constraint (bloc machin) (longueur >=24))!
*
```

cette nouvelle contrainte entraine l'affectation de la valeur 24 à l'attribut *longueur*.

```
(add_constraint (bloc machin) (coul in_set '(vert rouge blanc )))
    *
```

contraint l'attribut *coul* à prendre ses valeurs dans l'ensemble indiqué on a :

***in_set**(coul)=(vert, rouge, blanc)*(Pour la syntaxe des expressions, se référer au § correspondant).

***not_in_set**(coul)=()*

```
(add_constraint (bloc machin) (coul not_in_set '(rouge blanc)))
```

cette nouvelle contrainte impose la valeur `vert` à l'attribut *coul* ;

on a :

coul=vert

relacher les contraintes : relax_constraint

Syntaxe :

<relacher contrainte> : :=

(relax_constraint (<class><nom>)

((<attr><condition><exp>)*)

Description :

Pour chaque triplet *(<attr><condition><exp>)*, relache les contrainte de *<attr>* dans le sens indiqué. Si la contrainte exprimée dans la commande est plus forte que les contraintes de l'objet, il n'en n'est pas tenu compte.

lorsque la condition est ***in_set***, le domaine de l'attribut est étendu à l'ensemble des valeurs issu de *<exp>* (cf exemples).

Lorsque *<condition>=* ***not_in_set***, le relachement de la contrainte consiste à supprimer de l'ensemble des valeurs interdites de *<attr>*, les éléments de l'ensemble *<expr>* (cf exemples)

Restrictions :

l'objet sur qui l'opération porte doit exister au préalable ;

selon les cas, *<exp>* doit s'évaluer à un entier ou à un ensemble de valeurs ;

A.2.4. Les entrées-sorties

<entrée> :=

***(ask (file** <nomfich>) **(ext** <ext>)<type donnée>)*

***(ask** <type donnée>)*

<type donnée> :=

***integer** [in <numexp1>.. <numexp2>]*

***atom[in** <listexp>]*

<sortie> :=

***(write** (<data>)*)*

***(writeln** (<data>)*)*

***(write (file** <nomfich>) **(ext** <ext>) (<data>)*)*

*(**writeln** (**file** <nomfich>) (**ext** <ext>) (<data>)*)*

<data> :=
"<chaine>"
<exp>

description :

<entrée> est une expression qui s'évalue en demandant à l'utilisateur ou en lisant dans le fichier <nomfich>.<ext>, une valeur obéissant aux contraintes spécifiées par <type donnée>. <type donnée> peut spécifier soit un entier quelconque ou contenu dans un intervalle délimité par <numexp1> et <numexp2>, soit un atome quelconque ou appartenant à une liste de symbole. <listexp> est une expression qui doit s'évaluer à une liste d'atomes.

<sortie> est une action qui permet d'écrire à l'écran ou dans un fichier (<nomfich>.<ext>), une suite d'expressions. <chaine> désigne une chaine de caractère. **writeln** éffectue un retour à la ligne.

A.3. langage d'expression des regles

A.3.1. introduction

Dans le cas général, une règle est constituée de :

—une en-tête : prédicat lorsqu'il s'agit d'un théorème, évènement pour les démons et format pour les opérateurs ;

—une partie gauche constituée de formules de prédicat ou (et) de filtres ;

—une partie droite constituée d'opérations à exécuter ;

les § suivants présenteront la syntaxe des filtres théorèmes, démons et opérateurs.

A.3.2. les filtres

Syntaxe :

<filtre> : :=
<variable> **is** <specif ensemble>
<variable> **in** <specif ensemble>
(<class> <variable> <lconditions>)
[<expr1> <condition> <expr2>]

<specif ensemble> : :=

(<class> [in <expression symbolique>] [/ <lconditions>])

<lconditions> : :=

*<condition attribut> [,<condition attribut>]**

<condition attribut> : :=<attr> <condition> <exp>

formules

Syntaxe :

<formule> : :=

<formule déterminée>

unknown *<formule positive>*

<formule positive> : :=

<nom prédicat>(<class><expr>)

<nom prédicat>[[(<class><expr>)]]*

<formule déterminée> : :=

<formule positive>

not *<formule positive>*

Description :

<expr> doit pouvoir s'évaluer à une liste ou un identificateur selon les cas ; une formule s'évalue selon les cas :

—si c'est une *<formule déterminée>*, l'evaluation fournit la liste de toutes les instances de paramètres vérifiant la formule ; on rappèle qu'une formule est vérifiée pour un jeu de paramètres donné si il existe au moins un théorème dont la partie gauche est satisfaite pour les dits paramètres ;

—si la formule est introduite par not, elle s'evalue à vraie si aucun théorème ne permet de décider de la valeur de vérité de la *<formule positive>* qui suit le mot clé.

il n'y a aucune supposition à priori sur la valeur de vérité d'une formule ; ce n'est pas parce qu'on n' a pas pu démontrer qu'une propriété est vraie que cette propriété est fausse. C'est pourquoi pour un prédicat, il est souhaitable de définir les circonstances dans lesquelles elle est vraie et celle dans lesquelles elle est fausse. On peut définir des va leurs par défaut (cf exemples).

Restrictions :

de même les formules n'ont de sens que comme parties des condition d'une règle ;

Exemples :

cf théorèmes.

A.3.3. definition des theoremes

Syntaxe :

```
<théorème> : :=
if <partie gauche> conclude <formule determinée> end
<partie gauche> : :=
      <filtre>
      <formule>
      <partie gauche><partie gauche>
```

A.3.4. definition des operateurs

Syntaxe :

```
<operateur> : :=
      to_do <en tête> if <partie gauche> do <actions> end
      to_do <en tête> do <actions> end
      to_do <en tête> if <partie gauche> do end
      to_do <en tête> do end
<en tête> : :=
      <nom operateur> (<class parm><variable>)
      <nom operateur> [( (<class parm><variable>) )*]
<actions> : : =
      <action>
      <action><actions>
<action> : :=
      <primitive>
      (<nom operateur> ((<class parm><parm>) )*)
<primitive> : :=
      <creation objet>
      <modification objet>
      <retrait objet>
      <destruction objet>
      <contraindre objet>
      <relacher contrainte objet>
```

A.3.5. expression des attachements proceduraux

Syntaxe :

<attachement procédural> : :=

to_do *<en tête>* ***call_routine*** *<number>****end***

Description :

<en tête> est défini comme pour les opérateurs ;

<number> représente l'entrée de la routine fournie par l'utilisateur et correspondant à la branche d'une instruction "case" à exécuter ; une fois l'attachement procédural éffectuée, le système identifie l'opérateur ainsi défini comme un appel de procédure ; afin que la procédure utilisateur soit prise en compte par le système, il faut recompiler la procédure ***exe_routine*** en y rajoutant l'entrée correspondant ; ensuite refaire l'édition de lien du programme principal ODSE ; si cette manœuvre n'est pas obsevée, un message indique à l'utilisateur que la routine correspondante n'est pas implémentée et l'opération en cours échoue.

A.3.6. definition des demons

Syntaxe :

<demon> : :=

when *<évènement>* ***if*** *<partie gauche>* ***do*** *<actions>* ***end***

when *<évènement>* ***do*** *<actions>* ***end***

when*<exception>* ***do*** *<trait.excep>****end***

<évènement> : :=

created *(<class><varclass>)*

modified *<attr> (<class><varclass>)*

augmented *<attr> (<class><varclass>)*

retracted *<attr> (<class><varclass>)*

<exception> :=

undefined_class *<class>*

not_found *(<class><nom>)*

le_violation *<attr>(<class><nom>)*

lt_violation*<attr>(<class><nom>)*

gt_violation*<attr>(<class><nom>)*

ge_violation*<attr>(<class><nom>)*

in_set_violation*<attr>(<class><nom>)*

not_in_set_violation*<attr>(<class><nom>)*

<trait. excep> : :=

<actions>

(resume)

(retry)

A.4. interface de commande du systeme

Le système offre à l'utilisateur toute une game de commandes possibles allant de la construction de la des faits et l'enrichissement de la base de règles aux commandes d'exécution de tâches multiples. Parr commandes on distingue :

—les commandes de définition des attributs et des classes de la base ;

—les commandes d'édition de règles ;

—les commandes d'exécution de tâches dont la syntaxe générale est :

(<nomop>((<type parametre> <expr>)))!*

<nomop> est le nom d'une primitive ou d'un opérateur dont on veut l'exécution ; toute expression ayant cette syntaxe est interprétée par ODSE comme un ordre d'exécution de l'opérateur correspondant avec les paramètres indiqués.

—les commandes de mise au point.

Dans la section précédente, nous avons présenté les expressions et les commandes de consult reconnues par le système ; les chapitres suivants décriront les commandes de construction de la base de et d'enrichissement de la base de connaissance ; ce paragraphe se concentre sur les commandes utilitai de mise au point.

commandes de mise au point

description :

(trace (<class><expr>))!

description :

appliquée sur un objet, toute modification de la valeur d'un attribut quelconque de l'objet provoque l'affichage du nouvel état de l'objet à l'écran ; la modification des contraintes sur les attributs de l'objet ne provoque pas l'affichage des caractéristiques de l'objet sauf si cette modification conduit à une modification effective de la valeur d'un attribut ;

Restrictions :

l'argument de la commande doit être un objet présent en mémoire ;

exemples :

```
*(trace(bloc A))!
*(modify(bloc A)(haut 6))!
***************
bloc : A
------------------
haut : 6 constraints : ((ge, 0)(le,100))
------------------
```

place : P1 constraints : ((in_set,(P1, P2, P3, P4))

---

*

exécution des commandes d'un fichier

Syntaxe :

@<nomfich>

Description :

lit et exécute en séquence, les commandes du fichier *<nomfich>.**KBASE*** ; ces commandes doivent être écrites en respectant les règles de syntaxe ; a l'encontre d'un incident (exception ou autre) un message est envoyé à l'utilisateur, la commande est sautée et l'exécution reprend à la commande suivante ;

Restrictions :

avant d'éffectuer cette commande, il faut s'assurer de la présence du fichier "*.KBASE*" dans le directory courant ;

Exemples :

**@schedule*

lance l'exécution des commandes du fichier *schedule.**kbase***

chargement et sauvegarde d'une base de connaissance

<sauvegarde> :=

***(save(rulebase** <nomfich>))*

***(save(factbase** <nomfich>))*

<chargement> :=

***(load(rulebase** <nomfich>))*

***(load(factbase** <nomfich>))*

description :

<sauvegarde> permet la sauvegarde de la base de règle (***rulebase***) ou la base de fait (***factbase***) dans un fichier *<nomfich>.**kbs*** (base de règle) ou *<nomfich>.**fct*** (base de fait). la base est sauvegardée sous une forme intermédiaire qui permet un chargement rapide (puisqu'on évite de recompiler la base de connaissance).

<chargement> éffectue le chargement d'une base de connaissance (base de règles ou base de faits) à partir du fichier *<nomfich>.**kbs*** ou *<nomfich>.**fct***.

D'autres commandes de mise au point sont en cours d'étude (break sur les règles...).

Annexe B : Commandes ODSE de construction de ASA

Dans cette annexe, nous ne donnons qu'un échantillon des commandes qui ont servi à construire ASA.

```
; construction de la base de faits
;classe CHIP
def_attr plus singlevalued 0 10
def_attr minus singlevalued 0 10
def_attr mult singlevalued 0 10
def_attr div singlevalued 0 10
def_attr shiftr singlevalued 0 10
def_attr shiftl singlevalued 0 10
def_attr le singlevalued 0 10
def_attr ge singlevalued 0 10
def_attr gt singlevalued 0 10
def_attr lt singlevalued 0 10
def_attr eq singlevalued 0 10
def_attr neq singlevalued 0 10
def_attr operations multivalued (plus minus mult
   div shiftr shiftl et non ou)
def_obj chip (operations nbcycles plus minus mult
   div shiftr shiftl et non ou)
;definition des ressources
;classe FAMILLE
def_attr surface singlevalued atom
def_obj famille (time surface)
;classe RESOURCE
def_attr category singlevalued (flip_flop port
   storage bus multiplexer f_unit)
def_attr input1 multivalued atom
def_attr input2 multivalued atom
def_attr output multivalued atom
def_attr available singlevalued 0 50
def_attr until singlevalued 0 50
def_attr variables multivalued atom
def_attr sources multivalued atom
def_attr outputs multivalued atom
def_attr inputs multivalued atom
def_obj resource (category operation input1 input2
   output surface time
 available until variables cycles inputs)
;représentation interne de la description
   fonctionnelle du circuit
; classe VARIABLE
def_attr data_type singlevalued (boolean integer
   array_integer)
def_attr life multivalued atom
def_attr bitsize singlevalued 0 60
def_obj variable (bitsize data_type life)
;classe PARAMETER
def_attr mode singlevalued (input output)
def_obj parametre(mode data_type)
;classe CONSTANT
def_attr value singlevalued atom
def_obj constant (value)
; classe BLOC(GAO)
def_obj bloc (operations nodes leaves heads chip)
;classe NODE
def_attr operand1 singlevalued atom
def_attr operand2 singlevalued atom
def_attr bloc singlevalued atom
def_attr ready singlevalued atom
def_attr chip singlevalued atom
def_attr cycle singlevalued atom
def_attr variable singlevalued atom
def_attr group singlevalued atom
def_attr asap singlevalued atom
def_attr alap singlevalued atom
def_attr stored singlevalued atom
def_obj node (operation type_operand1 operand1
   ready cycle variable life type_operand2
   operand2 bloc group asap alap stored)
;classe GROUP(groupe de nœuds compatibles)
def_obj group(life variable operations)
;description du flot de contrôle
;classe TRANSITION
def_obj transition (from_bloc then_bloc else_bloc
   condition)
;classe CYCLE
def_attr resources_used multivalued atom
def_attr nbmoves singlevalued 0 20
def_obj cycle (bloc cycle plus minus mult div shiftr
   nbmoves shiftl le lt ge gt neq eq)
;BASE DE REGLES
;expression des regles de connaissances
to_do build (chip ?chip) do
(parse (file ?chip))
(lv_analysis(chip ?chip)) ; analyse de la
   vivacitédesvariables
(get_const(chip ?chip)) ; acquisition des contraintes
(critical_path(chip ?chip))
; analyse chemin chemin critique
(modify(chip ?chip)(nbcyc 0))
(schedule(chip ?chip))
(allocate(chip ?chip))
(links_synt(chip ?chip))
(optimize(chip ?chip))
; non implémenté
(Gen_spot(chip ?chip))
end

;Analyse du chemin critique
to_do critical_path(chip ?chip)
do
(asap_schedule(chip ?chip))
(alap_schedule(chip ?chip))
end

;determination des temps au plus tot
to_do asap_schedule(chip ?chip1)
if ?bloc1 in (bloc in [(chip ?chip1)blocs])
do
 (modify (bloc ?bloc1)(nbcycles 1))
(asap_schedule(bloc ?bloc1))
end
to_do asap_schedule(bloc ?bloc1)
if ?node in (node in [(bloc ?bloc1)leaves] ) do
(modify (node ?node) (asap 1))
end
to_do asap_schedule(bloc ?bloc1)
if (bloc ?bloc1 leaves= null )
do end
```

```
;demons pour asap_schedule
when modified asap(node ?node1)
if (node ?node1 operation not_in '(noop access))
 ?node2 in (node in successors(node
    ?node1) | asap=null,operation | =access)
 do
(constraint_asap(node ?node2)(after ?node1))
end
to_do constraint_asap[(node ?node2)(after
    ?node1)]
if (node ?node2 type_operand1=node,
    operand1=?node1)
 do
 (add_constraint(node ?node2)
(asap >= [(node ?node1)asap]+ [(famille [(node
    ?node1)operation])nbcycles]))
 (try_asap(node ?node2))
end

to_do constraint_asap[(node ?node2)(after
    ?node1)]
if (node ?node2 type_operand2=node,
    operand2=?node1)
 do
 (add_constraint(node ?node2)
(asap >= [(node ?node1)asap]+ [(famille [(node
    ?node1)operation])nbcycles]))
 (try_asap(node ?node2))
end
to_do constraint_asap[(node ?node2)(after
    ?node1)]
if (node ?node2
    operand1 | =?node1,operand2 | =?node1)
 do
 (add_constraint(node ?node2)
(asap >= [(node ?node1)asap]))
 (try_asap(node ?node2))
end

when modified asap(node ?node1)
if (node ?node1 operation=noop)
?node2 in (node in successors(node
    ?node1) | asap=null,operation | =access)
 do
 (add_constraint(node ?node2)
(asap >= [(node ?node1)asap]+1))
 (try_asap(node ?node2))
end
when modified asap(node ?node1)
if (node ?node1 operation not_in '(noop access),
 asap > [(bloc [(node ?node1)bloc])nbcycles]-
 [(famille [(node ?node1)operation])nbcycles]+1)
 do
(modify (bloc [(node ?node1)bloc])
(nbcycles [(node ?node1)asap]+ [(famille [(node
    ?node1)operation])nbcycles]-1))
end
when modified asap(node ?node1)
if (node ?node1 operation=noop,asap > [(bloc [(node
    ?node1)bloc])nbcycles])
 do
(modify (bloc [(node ?node1)bloc])
(nbcycles [(node ?node1)asap]))
end
to_do try_asap(node ?node1)
if
[(node in predecessors(node
    ?node1) | asap=null] | =null]
do
 end

 to_do try_asap(node ?node1)
 if [(node in predecessors(node
    ?node1) | asap=null]=null]
 do
 (modify (node ?node1) (asap lower_bound[(node
    ?node1)asap]))
 end
 Ordonnancement des opérations
 ;ordonnancement sous contraintes
 to_do schedule(chip ?chip)
 if ?bloc in (bloc in [(chip ?chip)blocs]) do
 (writeln "ordonnancement des noeuds du bloc
    "?bloc)
  (schedule(bloc ?bloc))
 end

 to_do schedule(bloc ?bloc)
 if [(node | bloc=?bloc,operation
  not_in '(non)]=null] do end
 ;ordonnancement sous contrainte de temps

 to_do schedule(bloc ?bloc)
 if (bloc ?bloc time=constrained)
  [(node | bloc=?bloc,operation
  not_in '(access)] | =null] do
 (modify(bloc ?bloc)(nbcyc 0)(cycles_cost 0))
 (schedule_cp(bloc ?bloc))
 (schedule_ncp(bloc ?bloc))
 end
 ;ordonnancement des operations critiques to_do
    schedule_cp(bloc ?bloc)
 if [(node | bloc=?bloc,asap=1,alap=1)=null]
 do end
 to_do schedule_cp(bloc ?bloc)
 if [(node | bloc=?bloc,asap=1,alap=1) | =null]
 ?node in (node | bloc=?bloc,operation not_in '(non),
    cycle=null)
 (node ?node asap=[(node ?node)alap]) do
 (schedule_cp(node ?node)(bloc ?bloc))
 end
 to_do schedule_cp[(node ?node)(bloc ?bloc)]
 if (node ?node operation not_in '(access store))
 do
 (writeln " noeud critique "?node " affecte au cycle
    "[(node ?node)asap])
 (check_incr
  (chip [(bloc[(node ?node)bloc])chip])
  (operation [(node ?node)operation])(bloc [(node
    ?node)bloc])
  (cycle [(node ?node)asap]))
 (schedule(node ?node)
  (cycle [(node ?node)asap]))
 end
 ;check_incr comptabilise les besoins en
 ;opérateurs du circuit
 to_do check_incr[(chip ?chip)
  (operation ?operation)(bloc ?bloc)
  (cycle ?cycle)]
 if [ [(chip ?chip)?operation]=0] do
 (modify(chip ?chip)
 (?operation [(chip ?chip)?operation]+1))
 end
 to_do check_incr[(chip ?chip)
  (operation ?operation)
  (bloc ?bloc)(cycle ?cycle)]
 if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
 [ [(chip ?chip)?operation]=
  [(cycle ?cycle1)?operation] ] do
 (modify(chip ?chip)
 (?operation [(chip ?chip)?operation]+1))
```

```
end
to_do check_incr[(chip ?chip)
 (operation ?operation)
 (bloc ?bloc)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
[ [(chip ?chip)?operation]>
 [(cycle ?cycle1)?operation] ] do end
to_do check_incr[(chip ?chip)
 (operation ?operation)(bloc ?bloc)
 (cycle ?cycle)]
if [(cycle | bloc=?bloc,cycle=?cycle)=null]
[ [(chip ?chip)?operation]>0 ] do end
 ;Ordonnancement des opérations non
; critiques
;schedule_ncp ordonnance les operations non
    critiques en s'efforcant de
;minimiser le nombre des operateurs les plus
    couteux--> on commence
;par les operations les plus couteuses qu'on s'efforce
    de disperser au
;maximum en ajoutant des operateurs que si besoin
    est.(branch and bound)
;precondition : les noeuds criques ont deja ete
    ordonnances
to_do schedule_ncp(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in
    '(non),cycle=null)=null]
do
end
to_do schedule_ncp(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in '(non ),
    cycle=null) | =null]
do
(schedule_ncp(operations [(bloc
    ?bloc)operations]\'(non noop))
 (bloc ?bloc))
(schedule_ncp(operation noop)(bloc ?bloc))
end
to_do schedule_ncp[(operations ?operations)(bloc
    ?bloc)]
if ['?operations=null]
do end
;on commence par les operations les plus couteuses
    en surface d'operateur
;les realisant
to_do schedule_ncp[(operations ?operations)(bloc
    ?bloc)]
if ['?operations | =null]
?operations2 is (famille in (maximums (famille in
    '?operations) surface))
do
(sched_ncp(operations '?operations2)(bloc ?bloc))
(schedule_ncp(operations
    '?operations\'?operations2)(bloc ?bloc))
end
to_do sched_ncp[(operations ?operations)(bloc
    ?bloc)]
if ?operation in (famille in '?operations)
do
(schedule_ncp(operation ?operation)(bloc ?bloc))
end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)]
if
[(node | bloc=?bloc,operation=?operation,cycle=null
    )=null]
do end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)]
if
?nodes is
    (node | bloc=?bloc,operation=?operation,cycle=n
    ull)
do
(schedule_ncp(operation ?operation)(bloc ?bloc)
 (cycle (min asap(node in '?nodes))))
end

to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)]
if
[?operation | =noop]
[[(chip [(bloc ?bloc)chip])?operation]=0]
?nodes is
    (node | bloc=?bloc,operation=?operation,cycle=n
    ull)
do
(writeln "ajout d'un operateur "?operation " au
    circuit "[(bloc ?bloc)chip])
(incr(chip [(bloc ?bloc)chip])(operation ?operation))
(schedule_ncp(operation ?operation)(bloc ?bloc)
 (cycle (min asap(node in '?nodes))))
end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
[?operation | =noop]
 ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
[(#(node | bloc=?bloc,operation=?operation,alap=?cy
    cle,cycle=null))
 <= [(chip [(bloc ?bloc)chip])?operation]-[(cycle
    ?cycle1)?operation]]
do
(schedule_max(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
(sched_next_ncp(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
 [(cycle | bloc=?bloc,cycle=?cycle)=null]
do
(create_cycle(cycle ?cycle)(bloc ?bloc))
(schedule_ncp(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
[?operation | =noop]
?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
[(#(node | bloc=?bloc,operation=?operation,alap=?cy
    cle,cycle=null))
 > [(chip [(bloc ?bloc)chip])?operation]-[(cycle
    ?cycle1)?operation]]
do
(writeln "ajout d'un operateur "?operation " au
    circuit "[(bloc ?bloc)chip])
(incr(chip [(bloc ?bloc)chip])(operation ?operation))
(reschedule_ncp(operation ?operation)(bloc ?bloc))
end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)]
if [?operation=noop]
[(node | bloc=?bloc,operation=noop,cycle=null)=null
    ]do end
to_do schedule_ncp[(operation ?operation)(bloc
    ?bloc)]
if [?operation=noop]
```

```
 ?node in
    (node | bloc=?bloc,operation=noop,cycle=null) do
(sched_noop(node ?node))
end
to_do sched_noop(node ?node)
if (node ?node type_operand1 | =parametre)do end
to_do sched_noop(node ?node)
if (node ?node type_operand1=parametre)
do
(schedule(node ?node)(cycle [(node ?node)asap]))
end
;schedule_max ordonnance autant d'operations que
    possibles compte tenu du
;nombre d'operateurs disponibles. les operations
    retenues sont celles
;dont la mobilite est minimale et
;dont le cout marginal (nbre de transferts
    d'operandes) est minimal
;creation d'un cycle dans un bloc
to_do schedule_max[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
do
(schedule_mcp(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
(schedule_bests(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
end


;ordonnancement sous contrainte de ressources
to_do schedule(bloc ?bloc)
if (bloc ?bloc time | =constrained)
 [(node | bloc=?bloc,operation
 not_in '(non)) | =null] do
(writeln "ordonnancement des operations du bloc "
    ?bloc)
(modify(bloc ?bloc)(nbcyc 0)(cycles_cost 0))
(init_ready(bloc ?bloc))
(schedule_max(bloc ?bloc))
end
to_do init_ready(bloc ?bloc)
if ?node in (node in [(bloc ?bloc)leaves] | operation
    not_in '(non))
do
(modify(node ?node)(ready 1))
end
;ordonnancement des noeuds sous contrainte de
    ressources
to_do schedule_max(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null) | =null]
do
(schedule_m(bloc ?bloc)(cycle 1))
end
to_do schedule_max(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null)=null]
do end
to_do schedule_m[(bloc ?bloc)(cycle ?cycle)]
if [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null)=null]
do end
to_do schedule_m[(bloc ?bloc)(cycle ?cycle)]
if [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null) | =null]
[(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null,ready<=?cycle)=null]
do
(schedule_m(bloc ?bloc)(cycle ?cycle+1))
end
to_do schedule_m[(bloc ?bloc)(cycle ?cycle)]
if
 [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null,ready<=?cycle) | =null]
 [(cycle | bloc=?bloc,cycle=?cycle)=null]
do
(create_cycle(cycle ?cycle)(bloc ?bloc))
(schedule_b(bloc ?bloc)(cycle ?cycle))
(schedule_m(bloc ?bloc)(cycle ?cycle+1))
end
to_do schedule_m[(bloc ?bloc)(cycle ?cycle)]
if
 [(node | bloc=?bloc,operation not_in '(non
    noop),cycle=null,ready<=?cycle) | =null]
do
(schedule_b(bloc ?bloc)(cycle ?cycle))
(schedule_m(bloc ?bloc)(cycle ?cycle+1))
end
to_do schedule_b[(bloc ?bloc)(cycle ?cycle)]
if ?operation in (famille in [(bloc
    ?bloc)operations]\'(noop non))
do
 (schedule_bests(operation ?operation)(bloc
    ?bloc)(cycle ?cycle))
end
;schedule_mcp ordonnance les operations dont la
    mobilite est devenu
;nulle (alap - cycle courant=0)
;precondition : il ya un nombre d'operateurs
    disponibles suffisants
;pour effectuer ces operations
to_do schedule_mcp[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
    [(node | bloc=?bloc,operation=?operation,alap=?c
    ycle,cycle=null)=null]
do end
to_do schedule_mcp[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
?node in
    (node | bloc=?bloc,operation=?operation,alap=?c
    ycle,cycle=null)
do
(writeln "noeud devenu critique "?node" affecte au
    cycle "?cycle)
(schedule(node ?node)(cycle ?cycle))
end

;schedule_best ordonnance autant d'operations
    qu'il reste d'operateurs
;disponibles capables de les executer; c'est ici qu'il
    faut bien decider
;des operations a prendre ou a differer pour un
    ordonnancement en un cycle
;ulterieur. On privilegie ceux dont les operandes
    participent deja a des
;operations deja ordonnances dans le cycle courant;
    ce faisant, on contribue
;a limiter le nombre de transferts distincts (donc de
    bus)requis pour le
; circuit final.
;prerequis : toustes les operations critiques ou de
    mobilite nulle ont deja
;ete ordonnances

to_do schedule_bests[(operation ?operation)(bloc
    ?bloc)(cycle ?cycle)]
if
```

```
[(node | bloc=?bloc,operation=?operation,cycle=null
   ,asap<=?cycle)=null]
do
end
to_do schedule_bests[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
(bloc ?bloc time | =constrained)
[(node | bloc=?bloc,operation=?operation,cycle=null
   ,
 ready<=?cycle)=null]
do
end
to_do schedule_bests[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
 (bloc ?bloc time=constrained)
[[(chip [(bloc ?bloc)chip])?operation]=[(cycle
   ?cycle1)?operation]]
do end
to_do schedule_bests[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
 (bloc ?bloc time | =constrained)
[upper_bound[(cycle ?cycle1)?operation]=[(cycle
   ?cycle1)?operation]]
do end
to_do schedule_bests[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
(bloc ?bloc time=constrained)
[[(chip [(bloc ?bloc)chip])?operation]>
 [(cycle ?cycle1)?operation]]
?nodes is (node in
(minimums
   (node | bloc=?bloc,operation=?operation,cycle=n
   ull,asap<=?cycle)alap) |
operand1 in [(cycle ?cycle1)operands],
operand2 in [(cycle ?cycle1)operands]]
do
(schedule_max(nodes '?nodes)(cycle ?cycle)
(number [(chip [(bloc ?bloc)chip])?operation]-[(cycle
   ?cycle1)?operation]))
(schedule_bests(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end

;ordonnancement d'un noeud a un cycle donne
;pre-requis: l'ordonnancement doit etre possible
   c.a.d asap<=cycle<=alap
to_do schedule[(node ?node)(cycle ?cycle)]
if ?cycle1 in (cycle | bloc=[(node
   ?node)bloc],cycle=?cycle)
do
(add_cycle(node ?node)(cycle ?cycle1))
(sequence(node ?node)(cycle ?cycle)
 (nbcycles [(famille [(node
   ?node)operation])nbcycles]))
end
to_do schedule[(node ?node)(cycle ?cycle)]
if [(cycle | bloc=[(node
   ?node)bloc],cycle=?cycle)=null]
do
(create_cycle(cycle ?cycle)(bloc [(node ?node)bloc]))
(add_cycle(node ?node)
 (cycle (first(cycle | bloc=[(node
   ?node)bloc],cycle=?cycle))))
(sequence(node ?node)(cycle ?cycle)
 (nbcycles [(famille [(node
   ?node)operation])nbcycles]))
end

to_do sequence[(node ?node)(cycle
   ?cycle)(nbcycles ?num)]
if [?num = 0 ]
do
end

to_do sequence[(node ?node)(cycle
   ?cycle)(nbcycles ?num)]
if [?num | =0]
[(cycle | bloc=[(node ?node)bloc],cycle=?cycle)=null]
do
(create_cycle(cycle ?cycle)(bloc [(node ?node)bloc]))
(incr(cycle (first (cycle | bloc=[(node
   ?node)bloc],cycle=?cycle)))
 (operation [(node ?node)operation]))
(sequence (node ?node)(cycle ?cycle+1)(nbcycles
   ?num-1))
end

to_do sequence[(node ?node)(cycle
   ?cycle)(nbcycles ?num)]
if [?num | =0]
?cycle1 in (cycle | bloc=[(node
   ?node)bloc],cycle=?cycle)
do
(incr(cycle ?cycle1)(operation [(node
   ?node)operation]))
(sequence (node ?node)(cycle ?cycle+1)(nbcycles
   ?num-1))
end
to_do decr[(cycle ?cycle)(operation ?operation)]
do
(modify(cycle ?cycle)(?operation [(cycle
   ?cycle)?operation]-1))
end
to_do unsequence[(node ?node)(cycle
   ?cycle)(nbcycles ?num)]
if [ ?num = 0 ]
do
end
to_do unsequence(node ?node)
if (node ?node cycle=null) do end
to_do unsequence(node ?node)
if (node ?node cycle | =null) do
(retract_operand(operand [(node ?node)operand1])
 (bloc [(node ?node)bloc])(cycle [(node ?node)cycle]))
(retract_operand(operand [(node
   ?node)operand2])(bloc [(node ?node)bloc])
 (cycle [(node ?node)cycle]))
(unsequence(node ?node)(cycle [(node ?node)cycle])
 (nbcycles [(famille [(node
   ?node)operation])nbcycles]))
(retract(node ?node)(cycle [(node ?node)cycle]))
(relax_asap(nodes (node in successors(node
   ?node) | operation=?operation)))
(relax_alap(nodes (node in predecessors(node
   ?node) | operation=?operation)))
 end

to_do unsequence[(node ?node)(cycle
   ?cycle)(nbcycles ?num)]
if [?num | =0]
?cycle1 in (cycle | bloc=[(node
   ?node)bloc],cycle=?cycle)
do
(decr(cycle ?cycle1)(operation [(node
   ?node)operation]))
```

```
(unsequence (node ?node)(cycle ?cycle+1)(nbcycles
   ?num-1))
end


to_do add_cycle[(node ?node)(cycle ?cycle)]
if [[(bloc[(node ?node)bloc])time]=constrained]
do
(modify(node ?node)(cycle [(cycle ?cycle)cycle]))
(update_alap(before ?node))
(update_asap(after ?node))
(add_cycle(operand [(node ?node)operand1])(cycle
   ?cycle))
(add_cycle(operand [(node ?node)operand2])(cycle
   ?cycle))
end
to_do add_cycle[(node ?node)(cycle ?cycle)]
if [[(bloc[(node ?node)bloc])time]!=constrained]
do
(modify(node ?node)(cycle [(cycle ?cycle)cycle]))
(add_cycle(operand [(node ?node)operand1])(cycle
   ?cycle))
(add_cycle(operand [(node ?node)operand2])(cycle
   ?cycle))
(modify(bloc [(node ?node)bloc])(cycles_cost [(cycle
   ?cycle)cycle]-
 [(node ?node)alap]))
(update_ready(node ?node))
end
to_do update_ready(node ?node1)
if [(node in successors(node ?node1))=null]
 do end
to_do update_ready(node ?node1)
if ?node2 in (node in successors(node ?node1))
 do
 (add_constraint(node ?node2)
(ready >= [(node ?node1)cycle]+ [(famille [(node
   ?node1)operation])nbcycles]))
 (try_ready(node ?node2))
end
to_do try_ready(node ?node2)
if [(node in predecessors(node
   ?node2)!cycle=null)!=null]do end
to_do try_ready(node ?node2)
if [(node in predecessors(node
   ?node2)!cycle=null)=null]
 do
(modify(node ?node2)(ready lower_bound[(node
   ?node2)ready]))
 end
to_do incr[(chip ?chip)(operation ?operation)]
do
(modify(chip ?chip)(?operation [(chip
   ?chip)?operation]+1))
end
to_do incr[(cycle ?cycle)(operation ?operation)]
do
(modify(cycle ?cycle)(?operation [(cycle
   ?cycle)?operation]+1))
end
to_do add_cycle[(operand ?operand)(cycle ?cycle)]
if
['?operand=null]do end
to_do add_cycle[(operand ?operand)(cycle ?cycle)]
if ['?operand!=null]
['?operand in [(cycle ?cycle)operands]]do end
to_do add_cycle[(operand ?operand)(cycle ?cycle)]
if ['?operand!=null]
['?operand not_in [(cycle ?cycle)operands]]
do
   (augment(cycle ?cycle)(operands ?operand))
   (modify(cycle ?cycle)(nbmoves [(cycle
      ?cycle)nbmoves]+1))
   end

   to_do sched_next_ncp[(operation ?operation)(bloc
      ?bloc)(cycle ?cycle)]
   if [(node!bloc
      =?bloc,operation=?operation,cycle=null]=null]
   do end
   to_do sched_next_ncp[(operation ?operation)(bloc
      ?bloc)(cycle ?cycle)]
   if ?nodes is (node!bloc
      =?bloc,operation=?operation,cycle=null)
   [(min asap(node in '?nodes))>?cycle]
   do
   (schedule_ncp(operation ?operation)(bloc ?bloc)
    (cycle (min asap(node in '?nodes))))
   end
   to_do sched_next_ncp[(operation ?operation)(bloc
      ?bloc)(cycle ?cycle)]
   if ?nodes is (node!bloc
      =?bloc,operation=?operation,cycle=null)
   [(min asap(node in '?nodes))<=?cycle]
   [(min alap (node in '?nodes))<=?cycle+[(famille
      ?operation)nbcycles]]
   do
   (schedule_ncp(operation ?operation)(bloc ?bloc)
    (cycle (min alap (node in '?nodes))))
   end

   to_do sched_next_ncp[(operation ?operation)(bloc
      ?bloc)(cycle ?cycle)]
   if ?nodes is (node!bloc
      =?bloc,operation=?operation,cycle=null)
   [(min asap(node in '?nodes))<=?cycle]
   [(min alap (node in '?nodes))>?cycle+[(famille
      ?operation)nbcycles]]
   do
   (schedule_ncp(operation ?operation)(bloc ?bloc)
    (cycle ?cycle+[(famille ?operation)nbcycles]))
   end
   ;reschedule_ncp recommence l'ordonnancement
      avec un operateur supplementaire
   ;on y est contraint lorsque la densite des operations
      ne permet pas de tous les
   ;ordonnancer sur le jeu d' operateurs existant en
      respectant les contraites de
   ;sequencement imposees par les dates au plus tot et
      dates au plus tard de ces
   ;operations. le re ordonnancement consiste a
      appliquer une strategie de type
   ;glouton visant le plein usage du jeu d'operateur
      existant.
   to_do reschedule_ncp[(operation ?operation)(bloc
      ?bloc)]
   do
   (writeln "re ordonnancement des operations "
      ?operation)
   (reschedule_ncp(operation ?operation)(bloc ?bloc)
    (cycle (min cycle
      (node!bloc=?bloc,operation=?operation,cycle!=
      null))))
   end
   to_do reschedule_ncp[(operation ?operation)(bloc
      ?bloc) (cycle ?cycle)]
   if ?cycle1 in (cycle!bloc=?bloc,cycle=?cycle)
    ?nodes is
   (node in
```

```
(minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle>?cycle,
 operand1 in [(cycle ?cycle1)operands],
 operand2 in [(cycle ?cycle1)operands])asap)or
(minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle=null,
 operand1 in [(cycle ?cycle1)operands],
 operand2 in [(cycle ?cycle1)operands])asap)}
do
(reschedule_ncp(node (first(minimums (node in
   ?nodes)asap)))(cycle ?cycle))
(resched_next(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end
to_do reschedule_ncp[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if
?cycle1 in (cycle | bloc=?bloc,cycle=?cycle)
 ?nodes is (node in
 (minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle>?cycle,
 operand2 in [(cycle ?cycle1)operands]) asap)or
 (minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle=null,
 operand2 in [(cycle ?cycle1)operands]) asap)or
 (minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle>?cycle,
 operand1 in [(cycle ?cycle1)operands]) asap)or
 (minimums
   (node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle=null,
 operand1 in [(cycle ?cycle1)operands]) asap)}

do
(reschedule_ncp(node (first (minimums (node in
   ?nodes)asap)))(cycle ?cycle))
(resched_next(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end
to_do reschedule_ncp[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if
 ?nodes is (node in
 (minimums
   (node | bloc=?bloc,operation=?operation,
 asap<= ?cycle,cycle>?cycle)asap)or
 (minimums
   (node | bloc=?bloc,operation=?operation,
 asap<= ?cycle,cycle=null)asap)}
do
(reschedule_ncp(node (first(minimums (node in
   ?nodes)asap)))(cycle ?cycle))
(resched_next(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end
to_do reschedule_ncp[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if

   [(node | bloc=?bloc,operation=?operation,asap<=
   ?cycle,cycle>?cycle)=null]
do
(resched_next(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end
to_do resched_next[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if
?nodes is
   (node | bloc=?bloc,operation=?operation,cycle>?
   cycle)
do
(reschedule_ncp(operation ?operation)(bloc ?bloc)
(cycle (min cycle (node in ?nodes))))
end
to_do resched_next[(operation ?operation)(bloc
   ?bloc)(cycle ?cycle)]
if
[(node | bloc=?bloc,operation=?operation,cycle>?cyc
   le)=null]
do
(schedule_ncp(operation ?operation)(bloc
   ?bloc)(cycle ?cycle))
end
to_do reschedule_ncp[(node ?node)(cycle ?cycle)]
do
(unsequence(node ?node))
(schedule(node ?node)(cycle ?cycle))
end
;rrecomp_asap ecalcule les dates au plus tôt
;recomp_alap recalcule les dates au plus tard
to_do recomp_asap(node ?node)
if
[(node in predecessors(node
   ?node) | cycle | =null)=null]do end
to_do recomp_asap(node ?node)
if
?node2 in (node in predecessors(node
   ?node) | cycle | =null)
do
(update_asap(node ?node)(after ?node2))
end
to_do update[(asap ?asap)(node ?node)]
if [?asap>[(node ?node)asap]]
do(modify(node ?node)(asap ?asap))end
to_do update[(asap ?asap)(node ?node)]
if [?asap<=[(node ?node)asap]]
do end

to_do relax_asap(nodes ?nodes)
if ['?nodes=null]do end
to_do relax_asap(nodes ?nodes)
if ?node in (node in '?nodes)
do
(relax_asap(node ?node))
 end
to_do relax_asap(node ?node)
do
(modify(node ?node)(asap lower_bound[(node
   ?node)asap]))
(recomp_asap(node ?node))
end



to_do relax_alap(nodes ?nodes)
if ['?nodes=null]do end
to_do relax_alap(nodes ?nodes)
if ?node in (node in '?nodes)
do
(relax_alap(node ?node))
 end
to_do relax_alap(node ?node)
do
```

```
(modify(node ?node)(alap upper_bound[(node
   ?node)alap]))
(recomp_alap(node ?node))
end



;description de l'allocation des unites fonctionelles
   et les elements de memo
to_do allocate(chip ?chip)
do
(modify(chip ?chip)(bitsize (max bitsize(variable))))
(global_alloc(chip ?chip))
(alloc_io_ports(chip ?chip))
(local_alloc(chip ?chip))
end
;allocation des ports d'entrées-sorties
to_do alloc_io_ports(chip ?chip)
if [(parametre)=null]do end
to_do alloc_io_ports(chip ?chip)
if ?param in (parametre)
do
(make (resource ?param)(category port)(mode
   [(parametre ?param)mode]))
(writeln " port "?param " alloue au circuit " ?chip)
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
end

; ALLOCATION DES ELEMENTS DE
   MEMORISATION GLOBAUX
; allocation des registres aux valeurs "globales"

to_do global_alloc(chip ?chip)
do
(modify(chip ?chip)(nbgroups 0)(nbres 0))
(compute_life(chip ?chip))
(group_varnodes(chip ?chip))
(alloc_groups(chip ?chip))
end


;calcul de la vivacite des valeurs

to_do compute_life(chip ?chip)
if [ (variable | life | =null)=null]
do end
to_do compute_life(chip ?chip)
if ?variable in (variable | life | =null)
do
(writeln "calcul vivacite des valeurs stockees dans
   la variable " ?variable)
(compute_life(variable ?variable))
end

;calcul de la vivacite des valeurs possibles d'une
   variable
to_do compute_life(variable ?variable)
if ?bloc in (bloc)
 [?variable in [(bloc ?bloc)gen_v_e]]
do
(modify(node [(variable ?variable)?bloc])(variable
   ?variable))
(compute_life(node [(variable ?variable)?bloc]))
end
to_do compute_life(variable ?variable)
if not bound_nodes(variable ?variable)
 do
(create_group(chip[(bloc (first[(variable
   ?variable)life]))chip])
 (variable ?variable))
 (add_life(variable ?variable)(group
    (first(group | nodes=null,life=null))))
 end
if ?bloc in (bloc)
[[(variable ?variable)?bloc] | =null]
conclude bound_nods(variable ?variable) end
if unknown bound_nods(variable ?variable)
 conclude not bound_nodes(variable ?variable) end
to_do add_life[(variable ?variable)(group ?group)]
if ?bloc in (bloc in [(variable ?variable)life]]
do
(augment(group ?group)(life ?bloc))
end
to_do compute_life(node ?node)
if ?bloc in (bloc in successors(bloc [(node
   ?node)bloc]))
do
(compute_life(node ?node)(bloc ?bloc))
end
to_do compute_life[(node ?node)(bloc ?bloc)]
if [ [(node ?node)variable] in [(bloc ?bloc)live_in]]
 [?bloc not_in [(node ?node)life]]
do
;le noeud est vivant dans le bloc en cours
(augment(node ?node)(life ?bloc))
(comp_next_life(node ?node)(bloc ?bloc))
end
to_do compute_life[(node ?node)(bloc ?bloc)]
if [ [(node ?node)variable] not_in [(bloc
   ?bloc)live_in]]
do end
to_do compute_life[(node ?node)(bloc ?bloc)]
if [?bloc in [(node ?node)life]]

do end
to_do comp_next_life[(node ?node)(bloc ?bloc)]
if ?bloc1 in (bloc in successors(bloc ?bloc))
do
(compute_life(node ?node)(bloc ?bloc1))
end
;groupe les valeurs d'une variable en noeuds
to_do group_varnodes(chip ?chip)
if ?var in (variable | life in [(chip ?chip)blocs])
do
(group_nodes(variable ?var))
end
to_do group_varnodes(chip ?chip)
if [(variable | life | =null)=null]
do
(writeln "no global variable in chip " ?chip)
end
to_do group_nodes(variable ?variable)
if ?bloc in (bloc in [(variable ?variable)life])
do
(group(variable ?variable)(bloc ?bloc))
end
to_do group_nodes(variable ?variable)
if
[(node | variable=?variable)=null]
do end

to_do group[(variable ?variable)(bloc ?bloc)]
if ?lnodes is
   (node | variable=?variable,life=?bloc,group=null)
 ?group in (group | variable=?variable,life=?bloc)
do
(augment(group ?group)(life ?bloc))
(add_group(nodes ?lnodes)(group ?group))
end
to_do group[(variable ?variable)(bloc ?bloc)]
```

```
if [ (group | variable=?variable,life=?bloc)=null]
 ?lnodes is
   (node | variable=?variable,life=?bloc,group=null)
do
(create_group(chip [(bloc ?bloc)chip])(variable
   ?variable))
(add_group(nodes '?lnodes)(group
 (first (group | life=null))))
end
to_do group[(variable ?variable)(bloc ?bloc)]
if
   [(node | variable=?variable,life=?bloc,group=null)
   =null]
do end
to_do create_group[(chip ?chip)(variable
   ?variable)]
do
(modify(chip ?chip)(nbgroups [(chip
   ?chip)nbgroups]+1))
(make (group (gensymbol group [(chip
   ?chip)nbgroups]))(chip ?chip)
 (variable ?variable))
end
to_do add_group[(nodes ?nodes)(group ?group)]
if ?node in (node in '?nodes | group=null)
do
(augment(group ?group)(operations [(node
   ?node)operation]))
(modify(node ?node)(group ?group))
(writeln " noeud " ?node " ajoute au groupe " ?group)
(add_life(node ?node)(group ?group))
end
to_do add_life[(node ?node)(group ?group)]
if ?bloc in (bloc in [(node ?node)life])
 [?bloc not_in [(group ?group)life]]
do(augment(group ?group)(life ?bloc))end
to_do add_life[(node ?node)(group ?group)]
do end
;allocation des groupes de valeurs globales
; generes a des elements de memorisation
to_do alloc_groups(chip ?chip)
if [(group | chip=?chip)=null]
do end
to_do alloc_groups(chip ?chip)
if ?group in (group | chip=?chip)
do
(writeln "allocation d un element de memo au
   groupe " ?group)
(allocate(group ?group))
end

;allocation d'un element de memorisation a un
; groupe de valeurs en mutuelle exclusion

to_do allocate(group ?group)
if ?resource in (resource | category=storage,
 life not_in [(group ?group)life])
best_storage[(resource ?resource)(group ?group)]
;s'il existe des elts de memo disponibles, choisir le
   meilleur
do
 (allocate(group ?group)(resource ?resource))
end


to_do allocate(group ?group)
if [(resource | category=storage,
 life not_in [(group ?group)life])=null]
;il n'y a pas d'element de memorisation disponible
do
(add_storage(chip [(group ?group)chip]))
(allocate(group ?group)
 (resource
   (first(resource | category=storage,life=null))))
end
to_do allocate[(group ?group)(resource ?resource)]
if (group ?group resource=null)
do
 (writeln "allocation de " ?resource " a "?group)
 (update_op(resource ?resource)(group ?group))
 (update_life(resource ?resource)(group ?group))
(affect_nodes(group ?group)(resource ?resource))
end
to_do allocate[(group ?group)(resource ?resource)]
if (group ?group resource | =null)
do end
;definition de best_storage[(resource
   ?resource)(group ?group)]
;un elt de memo est meilleur pour un groupe si il a un
   nombre maximal de
;connexions avec les operations du groupe
if ?res in (resource | category=storage,life not_in
   [(group ?group)life])
[(#[(group ?group)operations]\[(resource
   ?res)operations])<
 (#[(group ?group)operations]\[(resource
   ?resource)operations])]
conclude not_best_stor[(resource
   ?resource)(group ?group)]
end
if
unknown not_best_stor[(resource
   ?resource)(group ?group)]
conclude best_storage[(resource ?resource)(group
   ?group)] end

to_do update_op[(resource ?resource)(group
   ?group)]
do
(modify(resource ?resource)(operations
 [(resource ?resource)operations] or [(group
   ?group)operations]))
(modify (group ?group)(resource ?resource))
(augment(resource ?resource)(variables [(group
   ?group)variable]))
end

to_do update_life[(resource ?resource)(group
   ?group)]
if ?bloc in (bloc in [(group ?group)life])
do
(augment(resource ?resource)(life ?bloc))
end
to_do affect_nodes[(group ?group)(resource
   ?resource)]
if [ (node | group=?group)=null]
do end
to_do affect_nodes[(group ?group)(resource
   ?resource)]
if ?node in (node | group=?group)
do
(modify(node ?node)(stored ?resource))
end
to_do add_storage(chip ?chip)
do
(modify (chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol storage [(chip
   ?chip)nbres]))(category storage)
 (bitsize [(chip ?chip)bitsize]))
end
```

```
;ALLOCATION DES OPERATEURS ET DES
   TEMPORAIRES

to_do local_alloc(chip ?chip)
if ?bloc in (bloc | chip=?chip)
do
(allocate(bloc ?bloc))
end

;allocation des resources aux noeuds d'un bloc
to_do allocate(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in
   '(non),cycle | =null) | =null]
do
(writeln " affectation de temporaires aux noeuds du
   bloc " ?bloc)
(writeln "debut comput_life " ?bloc)
(comput_life(bloc ?bloc))
(writeln " compute_life termine " ?bloc)
(init_res(bloc ?bloc))
(allocate(bloc ?bloc)(cycle 1))
end
to_do allocate(bloc ?bloc)
if [(node | bloc=?bloc,operation not_in '(non))=null]
do
end
to_do comput_life(bloc ?bloc)
if ?nod in (node | bloc=?bloc,operation not_in
   '(non))
do
(writeln "compute_life "?nod)
(comp_life(node ?nod))
end
to_do comp_life(node ?node)
if (node ?node variable=null)
?nodes is ( node in
 (node in successors(node ?node) |
   type_operand1=node,operand1=?node) or
 (node in successors(node
   ?node) | type_operand2=node,operand2=?node))
do
(modify(node ?node)
(last_use (max cycle (node in '?nodes))))
end
to_do comp_life(node ?node)
if (node ?node variable=null)
[(node in successors(node ?node) |
   type_operand1=node,operand1=?node)=null]
[ (node in successors(node
   ?node) | type_operand2=node,operand2=?node)=
   null]
do
(modify(node ?node)(last_use [(node ?node)cycle]))
end
to_do comp_life(node ?node)
if (node ?node variable | =null)
do
(modify(node ?node)(last_use [(bloc[(node
   ?node)bloc])nbcyc]))
end
to_do init_res(bloc ?bloc)
if ?resource in (resource)
do
(init_available(bloc ?bloc)(resource ?resource))
(init_until(bloc ?bloc)(resource ?resource))
end

to_do init_available[(bloc ?bloc)(resource
   ?resource)]
if (resource ?resource variables not_in [(bloc
   ?bloc)live_in])
do
(modify(resource ?resource)(available 1))
end

to_do init_available[(bloc ?bloc)(resource
   ?resource)]
if (resource ?resource variables in [(bloc
   ?bloc)live_in],
variables not_in [(bloc ?bloc)live_out])
?lnodes is (node in (node | bloc=?bloc,
type_operand1=variable,operand1 in [(resource
   ?resource)variables])or
(node | bloc=?bloc,type_operand2=variable,
operand2 in [(resource ?resource)variables]))
do
(modify(resource ?resource)(available (max
   cycle(node in '?lnodes))))
end

to_do init_until[(bloc ?bloc)(resource ?resource)]
if (resource ?resource variables not_in [(bloc
   ?bloc)live_out])
do
(modify(resource ?resource)(until [(bloc
   ?bloc)nbcyc]))
end

to_do init_until[(bloc ?bloc)(resource ?resource)]
if (resource ?resource variables in [(bloc
   ?bloc)live_out]and
[(bloc ?bloc)gen_v_e])
?node in (node | bloc=?bloc,stored=?resource)
do
(modify(resource ?resource)(until [(node
   ?node)cycle]))
end
to_do init_until[(bloc ?bloc)(resource ?resource)]
if (resource ?resource variables in [(bloc
   ?bloc)live_out]\
[(bloc ?bloc)gen_v_e])
do
(modify(resource ?resource)(until 1))
end

to_do allocate[(bloc ?bloc)(cycle ?cycle)]
if [(cycle | bloc=?bloc,cycle=?cycle) | =null] do
(allocate(cycle
   (first(cycle | bloc=?bloc,cycle=?cycle))))
(allocate(bloc ?bloc)(cycle ?cycle+1))
end
to_do allocate[(bloc ?bloc)(cycle ?cycle)]
if [(cycle | bloc=?bloc,cycle=?cycle)=null]
do end


to_do allocate(cycle ?cycle)
if ?node in (node | bloc= [(cycle ?cycle)bloc],
 cycle=[(cycle ?cycle)cycle],
operation not_in '(non),f_unit=null)
do
(allocate(node ?node))
end
to_do allocate(cycle ?cycle)
if [ (node | bloc=[(cycle ?cycle)bloc],operation
   not_in '(non),
```

```
 cycle=[(cycle ?cycle)cycle],f_unit=null)=null] do
   end

to_do allocate(node ?node)
do
(allocate_fu(node ?node))
(allocate_s(node ?node))
(update_sd(node ?node))
end



to_do allocate_fu(node ?node)
if (node ?node
   operation!=noop,type_operand1=node,type_ope
   rand2=node)
do
(allocate_fu(node ?node)(input1 [(node[(node
   ?node)operand1])stored])
 (input2 [(node [(node ?node)operand2])stored]))
end
to_do allocate_fu(node ?node)
if (node ?node operation!=noop,
 type_operand1=node,type_operand2 not_in '(node
   variable))
do
(allocate_fu(node ?node)(input1 [(node[(node
   ?node)operand1])stored]))
end
to_do allocate_fu(node ?node)
if (node ?node operation!=noop,
 type_operand1=variable,type_operand2 not_in
   '(node variable))
?group in (group!variable=[(node
   ?node)operand1],life=[(node ?node)bloc])
do
(allocate_fu (node ?node)(input1 [(group
   ?group)resource]))
end

to_do allocate_fu[(node ?node)(input1
   ?input1)(input2 ?input2)]
if ?resources is
   (resource!category=f_unit,operation=[(node
   ?node)operation],
 available<=[(node ?node)cycle])
do
(allocate_fu(resources '?resources)(node
   ?node)(input1 ?input1)(input2 ?input2))
end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if ?res is (resource in
   '?resources!input1=?input1,input2=?input2)
do
(modify(node ?node)(fu_input1 ?input1)(fu_input2
   ?input2))
(augment(resource (first '?res))(input1
   ?input1)(input2 ?input2))
(allocate_fu(resource (first '?res))(node ?node))
end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if
[ (resource in
   '?resources!input1=?input1,input2=?input2)
   =null]
 commutative(operation [(node ?node)operation])
?res is (resource in
   '?resources!input2=?input1,input1=?input2)
do
(modify(node ?node)(fu_input1 ?input2)(fu_input2
   ?input1))
(augment(resource (first '?res))(input1
   ?input2)(input2 ?input1))
(allocate_fu(resource (first '?res))(node ?node))
end
if ?operation in (operation in '(plus mult et ou))
conclude commutative(operation ?operation) end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if
[ (resource in '?resources!
   input1=?input1,input2=?input2) =null]
 ?res is (resource in '?resources!input1=?input1)
do
(modify(node ?node)(fu_input1 ?input1)(fu_input2
   ?input2))
(augment(resource (first '?res))(input1
   ?input1)(input2 ?input2))
(allocate_fu (resource (first '?res))(node ?node))
end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if
[ (resource in '?resources!
   input1=?input1,input2=?input2) =null]
 ?res is (resource in '?resources!input2=?input2)
do
(modify(node ?node)(fu_input1 ?input1)(fu_input2
   ?input2))
(augment(resource (first '?res))(input1
   ?input1)(input2 ?input2))
(allocate_fu(resource (first '?res))(node ?node))
(augment(resource (first '?res))(input1 ?input1))
end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if
[ (resource in '?resources! input1=?input1) =null]
[ (resource in '?resources! input2=?input2) =null]
?res in (resource in '?resources)
good_choice[(resource ?res)(input1 ?input1)(input2
   ?input2)]
do
(alloc_fu(resource ?res)(node ?node)(input1
   ?input1)(input2 ?input2))
end
to_do allocate_fu[(resources ?resources)(node
   ?node)
 (input1 ?input1)(input2 ?input2)]
if
[ (resource in '?resources! input1=?input1) =null]
[ (resource in '?resources! input2=?input2) =null]
do
(modify(node ?node)(fu_input1 ?input1)(fu_input2
   ?input2))
(augment(resource (first '?resources))(input1
   ?input1)(input2 ?input2))
(allocate_fu(resource (first '?resources))(node
   ?node))
end


if
```

```
[(node | operation=[(resource
    ?resource)operation],f_unit=null,operand1=?inp
    ut1,
 operand2 in [(resource ?resource)input1]) | =null]
 conclude
 bad_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 end
 if unknown bad_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 conclude good_choice[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 end
 if unknown bad_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 conclude good_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 end

 if commutative(operation [(resopurce
    ?resource)operation])
  unknown bad_link[(resource ?resource)(input1
    ?input2)(input2 ?input1)]
 conclude good_choice[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 end

 if
 [(node | operation=[(resource
    ?resource)operation],f_unit=null,operand2=?inp
    ut2,
 operand1 in [(resource ?resource)input2]) | =null]
 conclude
 bad_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 end

 to_do alloc_fu[(resource ?resource)(node
    ?node)(input1 ?input1)(input2 ?input2)]
 if (node ?node f_unit | =null)
 do end
 to_do alloc_fu[(resource ?resource)(node
    ?node)(input1 ?input1)(input2 ?input2)]
 if (node ?node f_unit=null)
  good_link[(resource ?resource)(input1
    ?input1)(input2 ?input2)]
 do
 (modify(node ?node)(fu_input1 ?input1)(fu_input2
    ?input2))
 (augment(resource ?resource)(input1
    ?input1)(input2 ?input2))
 (allocate_fu(resource ?resource)(node ?node))
 end


 to_do allocate_fu[(node ?node)(input1
    ?input1)(input2 ?input2)]
 if
  [(resource | category=f_unit,operation=[(node
    ?node)operation],
  available<=[(node ?node)cycle])=null]
 do
 (add_fu(operation [(node ?node)operation])
  (chip [(bloc [(node ?node)bloc])chip]))
 (modify(node ?node)(fu_input1 ?input1)(fu_input2
    ?input2))
 (augment(resource (first
    (resource | category=f_unit,
    operation=[(node
    ?node)operation],available=null)))(input1
    ?input1)
   (input2 ?input2))
   (allocate_fu(resource (first
    (resource | category=f_unit,
   operation=[(node
    ?node)operation],available=null)))(node ?node))
   end
   to_do allocate_fu[(node ?node)(input1 ?input1)]
   if ?resources is
    (resource | category=f_unit,operation=[(node
    ?node)operation],
   available<=[(node ?node)cycle],
   input1=?input1)
   do
   (modify(node ?node)(fu_input1 ?input1))
   (allocate_fu(resource (first '?resources))(node
    ?node))
   (augment(resource (first '?resources))(input1
    ?input1))
   end
   to_do allocate_fu[(node ?node)(input1 ?input1)]
   if
   [ (resource | category=f_unit,operation=[(node
    ?node)operation],
   available<=[(node
    ?node)cycle],input1=?input1)=null]
   ?resources is
    (resource | category=f_unit,operation=[(node
    ?node)operation],
   available<=[(node ?node)cycle])
   do
   (modify(node ?node)(fu_input1 ?input1))
   (allocate_fu(resource (first '?resources))(node
    ?node))
   (augment(resource (first '?resources))(input1
    ?input1))
   end
   to_do allocate_fu[(node ?node)(input1 ?input1)]
   if
    [(resource | category=f_unit,operation=[(node
    ?node)operation],
   available<=[(node ?node)cycle])=null]
   do
   (add_fu(operation [(node ?node)operation])
    (chip [(bloc [(node ?node)bloc])chip]))
   (modify(node ?node)(fu_input1 ?input1))
   (augment(resource (first
    (resource | category=f_unit,available=null,
   operation=[(node
    ?node)operation],cycles=null)))(input1 ?input1))
   (allocate_fu(resource (first
    (resource | category=f_unit,
   operation=[(node
    ?node)operation],available=null,cycles=null)))(n
    ode ?node))
   end


   to_do allocate_fu[(resource ?resource)(node
    ?node)]
   do
   (write "allocation de l'operateur " ?resource " de
    fonction ")
   (writeln [(resource ?resource)operation] " au noeud "
    ?node)
   (modify(node ?node)(f_unit ?resource))
   (update_cycles(resource ?resource)(node ?node))
   (modify(resource ?resource)
```

```
(available [(node ?node)cycle]+[(famille [(node
   ?node)operation])nbcycles]))
end
to_do update_cycles[(resource ?resource)(node
   ?node)]
if ?cycle in (cycle | cycle>=[(node ?node)cycle],
 cycle<=[(node ?node)cycle]+[(famille [(node
   ?node)operation])nbcycles]-1)
do
(augment(resource ?resource)(cycles ?cycle))
end
;allocation de temporaires aux operandes
to_do allocate_s(node ?node)
if (node ?node stored | =null)
do
(augment(resource [(node ?node)stored])(nodes
   ?node))
(augment(resource [(node ?node)f_unit])(output
   [(node ?node)stored]))
end
to_do allocate_s(node ?node)
if (node ?node stored=null,operation=noop)
[ (resource | category=storage, available<=[(node
   ?node)cycle],
 until>=[(node ?node)last_use])=null]
do
(add_storage(chip [(bloc [(node ?node)bloc])chip]))
(update_alloc_s(node ?node)
(resource
   (first(resource | category=storage,available=null)
   )))
end
to_do allocate_s(node ?node)
if (node ?node stored=null,operation in '(gt ge le lt
   eq neq))
?res is (resource in (minimums
   (resource | category=flip_flop,
 available<=[(node ?node)cycle],
 until>=[(node ?node)last_use])until))
do
(allocate_s(resources '?res)(node ?node))
end
to_do allocate_s(node ?node)
if (node ?node stored=null,operation not_in '(gt ge
   le lt eq neq non noop))
?lres is (resource in (minimums
   (resource | category=storage,
 available<=[(node ?node)cycle],
 until>=[(node ?node)last_use])until))

do
(allocate_s(resources '?lres)(node ?node))
end
to_do allocate_s[(resources ?resources)(node
   ?node)]
if ?lres is (resource in '?resources | input1 =[(node
   ?node)f_unit])
do
(modify(node ?node)(stored (first '?lres)))
(modify (resource (first '?lres))(available [(node
   ?node)last_use]))
(augment(resource (first '?lres))(input1 [(node
   ?node)f_unit])(nodes ?node))
(augment(resource [(node ?node)f_unit])(output
   (first '?lres)))
end
to_do allocate_s[(resources ?resources)(node
   ?node)]
if
[(resource in '?resources | input1=[(node
   ?node)f_unit])=null]
do
(writeln " memoire "(first '?resources) " affectee au
   noeud " ?node)
(modify(node ?node)(stored (first '?resources)))
(modify(resource (first '?resources))(available
   [(node ?node)last_use]))
(augment(resource (first '?resources))(input1
   [(node ?node)f_unit])
(nodes ?node))
(augment(resource [(node ?node)f_unit])(output
   (first '?resources)))
end
to_do allocate_s(node ?node)
if (node ?node stored=null,operation not_in '(le lt
   ge gt eq neq noop))
[ (resource | category=storage,
available<=[(node ?node)cycle],until>=[(node
   ?node)last_use])=null]
do
(add_storage(chip [(bloc [(node ?node)bloc])chip]))
(update_alloc_s(node ?node)
(resource (first
   (resource | category=storage,available=null))))
end
to_do allocate_s(node ?node)
if (node ?node stored=null,operation in '(le lt ge gt
   eq neq))
[ (resource | category=flip_flop,
available<=[(node ?node)cycle],until>=[(node
   ?node)last_use])=null]
do
(add_storage(chip [(bloc [(node ?node)bloc])chip]))
(modify(resource
   (first(resource | category=storage,available=null)
   ))
 (category flip_flop))
(update_alloc_s(node ?node)
(resource
   (first(resource | category=flip_flop,available=nul
   l))))
end
;Synthèse des interconnexions
to_do links_synt(chip ?chip)
do
(writeln " synthese des interconnexions entre les
   operateurs ")
(writeln " 1 connexions point a point ")
(writeln " 2 bus ")
(writeln " faites votre choix ")
(links_synt(chip ?chip)(choice (ask number in
   1..2)))
end
to_do links_synt[(chip ?chip)(choice ?num)]
 if [?num=2]
 do
 (bus_synt(chip ?chip))
end

to_do links_synt[(chip ?chip)(choice ?num)]
 if [?num=1]
 do
(alloc_mux(chip ?chip))
end

to_do alloc_mux(chip ?chip)
 if ?resource in (resource | category in '(storage
   f_unit))
 do
```

```
 (alloc_mux(resource ?resource)(chip ?chip))
end
;solution bus
to_do bus_synt(chip ?chip)
 do
; (init_inputs(chip ?chip))
 (add_bus(chip ?chip)(nbus (max nbmoves
   (cycle | bloc in [(chip ?chip)blocs])))
 (allocate_bus(chip ?chip)
 (nbus (max nbmoves(cycle | bloc in [(chip
   ?chip)blocs])))
end
to_do init_inputs(chip ?chip)
if ?res in (resource)
do
 (delete(resource ?res)(input1 [(resource
   ?res)input1]))
 (delete(resource ?res)(input2 [(resource
   ?res)input1]))
end
to_do add_bus[(chip ?chip)(nbus ?nbus)]
if [?nbus=0]do end
to_do add_bus[(chip ?chip)(nbus ?nbus)]
if [?nbus>0]
do
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol bus [(chip
   ?chip)nbres]))(category bus)
(bitsize [(chip ?chip)bitsize])(chip ?chip))
(add_bus(chip ?chip)(nbus ?nbus-1))
end
to_do allocate_bus[(chip ?chip)(nbus ?nbus)]
if ?cycles is (cycle | bloc in [(chip
   ?chip)blocs],nbmoves=?nbus)
do
(allocate_bus(cycles '?cycles))
(allocate_bus(chip ?chip)(nbus ?nbus-1))
end
to_do allocate_bus[(chip ?chip)(nbus ?nbus)]
if
[?nbus>0]
[(cycle | bloc in [(chip
   ?chip)blocs],nbmoves=?nbus)=null]
do
(allocate_bus(chip ?chip)(nbus ?nbus-1))
end
to_do allocate_bus[(chip ?chip)(nbus ?nbus)]
if
[?nbus=0]
do end
to_do allocate_bus(cycles ?cycles)
if ?cycle in (cycle in '?cycles)
do
(allocate_bus(cycle ?cycle))
end
to_do allocate_bus(cycle ?cycle)
if ?node in (node | bloc=[(cycle
   ?cycle)bloc],cycle=[(cycle ?cycle)cycle])
do
(allocate_bus(input [(node ?node)fu_input1])
 (output1 [(node ?node)f_unit])(node ?node)(cycle
   ?cycle))
(allocate_bus(input [(node ?node)fu_input2])
 (output2 [(node ?node)f_unit])(node ?node)(cycle
   ?cycle))
(allocate_bus(input [(node ?node)f_unit])(output1
   [(node ?node)stored])
 (node ?node)(cycle ?cycle))
end

to_do allocate_bus[(input ?input)(output1
   ?output)(node ?node)(cycle ?cycle)]
if [?input | =null] [?output | =null]
 (resource ?input category=f_unit)
 ?lbus is (resource in [(resource
   ?output)input1] | category=bus,
 inputs=?input,phase2 | =?cycle)
do
(augment(resource (first '?lbus))(phase2
   ?cycle)(inputs ?input))
(modify(resource ?input)(?cycle (first '?lbus)))
end
to_do allocate_bus[(input ?input)(output1
   ?output)(node ?node)(cycle ?cycle)]
if [?input | =null] [?output | =null]
 (resource ?input category=f_unit)
 [(resource in [(resource
   ?output)input1] | category=bus,inputs=?input,
 phase2 | =?cycle)=null]
 ?fu is (resource | category=f_unit,cycles=?cycle)
 ?lbus is (resource in [(resource
   ?output)input1] | category=bus,
 inputs not_in '?fu, phase2 | =?cycle)
do
(augment(resource (first '?lbus))(phase2
   ?cycle)(inputs ?input) )
(modify(resource ?input)(?cycle (first '?lbus)))
(connect(bus (first '?lbus))(output1 ?output)(chip
   [(cycle ?cycle)chip]))
end

to_do allocate_bus[(input ?input)(output1
   ?output)(node ?node)(cycle ?cycle)]
if [?input | =null] [?output | =null]
 (resource ?input category=f_unit)
 [(resource in [(resource
   ?output)input1] | category=bus,inputs=?input,
 phase2 | =?cycle)=null]
?lbus is (resource in [(resource
   ?output)input1] | category=bus,phase2 | =?cycle)
do
(augment(resource (first '?lbus))(phase2
   ?cycle)(inputs ?input))
(modify(resource ?input)(?cycle (first '?lbus)))
(connect(bus (first '?lbus))(output1 ?output)(chip
   [(cycle ?cycle)chip]))
end

to_do allocate_bus[(input ?input)(output1
   ?output)(node ?node)(cycle ?cycle)]
if [?input | =null] [?output | =null]
 (resource ?input category=f_unit)
 [(resource in [(resource
   ?output)input1] | category=bus,
 phase2 | =?cycle)=null]
?lbus is (resource | category=bus,phase2 | =?cycle)
do
(augment(resource (first '?lbus))(phase2
   ?cycle)(inputs ?input))
(modify(resource ?input)(?cycle (first '?lbus)))
(connect(bus (first '?lbus))(output1 ?output)(chip
   [(cycle ?cycle)chip]))
end


to_do allocate_bus[(input ?input)(output1
   ?output)(node ?node)(cycle ?cycle)]
```

```
;il existe un bus contenant la source au cycle
   concerne
if [?input |=null][?output |=null]
 [(resource in [(resource ?output)input1] |

   category=bus,inputs=?input,cycles=?cycle,?cycle
   =?input) |=null]
 do
end

category=bus,inputs=?input,cycles=?cycle,?cycle=?i
   nput)=null]
 ?ibus is
   (resource |category=bus,inputs=?input,cycles=?c
   ycle,
 ?cycle=?input)
do
(connect(bus (first '?ibus))(output2 ?output)(chip
   [(cycle ?cycle)chip]))
end

to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input=null]do end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]

   [(resource |category=bus,inputs=?input,cycles=?
   cycle,
 ?cycle=?input)=null]
 ?bus1 is (resource in [(resource
   ?output)input2] |category=bus,
 inputs=?input,cycles |=?cycle)
 do
 (modify(resource (first '?bus1))(?cycle ?input))
 end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]
 [(resource in [(resource
   ?output)input2] |category=bus,
 inputs=?input,cycles=?cycle,?cycle=?input)=null]
 ?bus1 is
   (resource |category=bus,inputs=?input,cycles=?c
   ycle,?cycle=?input)
 do
 (connect(bus (first '?bus1))(output2 ?output)(chip
   [(cycle ?cycle)chip]))
 end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]
 [(resource |category=bus,
 inputs=?input,cycles=?cycle,?cycle=?input)=null]
 ?bus1 is (resource in [(resource
   ?output)input2] |category=bus,
 inputs=?input,cycles |=?cycle)
 do
 (modify(resource (first '?bus1))(?cycle ?input))
 (augment(resource (first '?bus1))(cycles ?cycle))
 end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]
 [(resource in [(resource
   ?output)input2] |category=bus,
 inputs=?input,cycles=?cycle,?cycle=?input)=null]
 [ (resource in [(resource
   ?output)input2] |category=bus,
 inputs=?input,cycles |=?cycle)=null]
 ?bus1 is
   (resource |category=bus,inputs=?input,cycles |=?
   cycle)
 do
 (modify (resource (first '?bus1))(?cycle ?input))
 (augment(resource (first '?bus1))(cycles ?cycle))
 (connect(bus (first '?bus1))(output2 ?output)(chip
   [(cycle ?cycle)chip]))
 end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]

   [(resource |category=bus,inputs=?input,cycles=?
   cycle,
 ?cycle=?input)=null]

   [(resource |category=bus,inputs=?input,cycles |=
   ?cycle)=null]
 ?res is (resource |category=storage,sources in
   [(resource ?input)sources]}
 ?bus is (resource |category=bus,cycles |=?cycle,
 inputs not_in '?res,output=?output)
 do
 (modify(resource (first '?bus))(?cycle ?input))
 (augment(resource (first '?bus))(cycles
   ?cycle)(inputs ?input))
 (augment(resource ?input)(output (first '?bus)))
 (connect(bus (first '?bus))(output2 ?output)(chip
   [(cycle ?cycle)chip]))
 end
 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]

   [(resource |category=bus,inputs=?input,cycles=?
   cycle,
 ?cycle=?input)=null]

   [(resource |category=bus,inputs=?input,cycles |=
   ?cycle)=null]
 ?res is (resource |category=storage,sources in
   [(resource ?input)sources]}
 ?bus is (resource |category=bus,cycles |=?cycle,
 inputs not_in '?res}
 do
 (modify(resource (first '?bus))(?cycle ?input))
 (augment(resource ?input)(output (first '?bus)))
 (augment(resource (first '?bus))(inputs
   ?input)(inputs ?input))
 (connect(bus (first '?bus))(output2 ?output)(chip
   [(cycle ?cycle)chip]))
 end

 to_do allocate_bus[(input ?input)(output2
   ?output)(node ?node)(cycle ?cycle)]
if [?input |=null]

   [(resource |category=bus,inputs=?input,cycles |=
   ?cycle)=null]
 ?res is (resource |category=storage,sources in
   [(resource ?input)sources]}
 [(resource |category=bus,cycles |=?cycle,inputs
   not_in '?res)=null]
 ?bus in (resource |category=bus,cycles |=?cycle}
 best[(resource ?bus)(input ?input)(output2
   ?output)(cycle ?cycle)]
 do
```

```
(alloc_bus(resource ?bus)(input ?input)(output2
   ?output)(cycle ?cycle))
 end
to_do alloc_bus[(resource ?bus)(input
   ?input)(output2 ?output)(cycle ?cycle)]
do
(modify(resource ?bus)(?cycle ?input))
(augment(resource ?bus)(inputs ?input)(cycles
   ?cycle))
 (augment(resource ?input)(output ?bus))
(connect(bus ?bus)(output2 ?output)(chip [(cycle
   ?cycle)chip]))
end
if
?bus1 in (resource | category=bus,cycles | =?cycle)
?res is (resource | sources in [(resource
   ?input)sources])
[(# [(resource ?bus1)inputs]and '?res)< (# [(resource
   ?bus)inputs]and '?res)]
conclude not_best[(resource ?bus)(input
   ?input)(output2 ?output)(cycle ?cycle)]
end
if
?bus1 in (resource | category=bus,cycles | =?cycle)
?res is (resource | sources in [(resource
   ?input)sources])
[(# [(resource ?bus1)inputs]and '?res)= (# [(resource
   ?bus)inputs]and '?res)]
[?bus1 in [(resource ?output)input2]]
[?bus not_in [(resource ?output)input2]]
conclude not_best[(resource ?bus)(input
   ?input)(output2 ?output)(cycle ?cycle)]
end

if
unknown not_best[(resource ?bus)(input
   ?input)(output2 ?output)(cycle ?cycle)]
conclude best[(resource ?bus)(input
   ?input)(output2 ?output)(cycle ?cycle)]
end

to_do connect[(bus ?bus)(output2 ?output)(chip
   ?chip)]
if (resource ?output in_link2=null)
do
(augment(resource ?output) (input2 ?bus))
(modify(resource ?output)(in_link2 ?bus))
(augment (resource ?bus) (outputs ?output))
end
to_do connect[(bus ?bus)(output2 ?output)(chip
   ?chip)]
if (resource ?output in_link2 | =null)
 [?bus in [(resource ?output)input2]]
do
end
to_do connect[(bus ?bus)(output2 ?output)(chip
   ?chip)]
if (resource ?output in_link2 | =null)
 [?bus not_in [(resource ?output)input2]]
do

(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol multiplexer
 [(chip ?chip)nbres]))(category multiplexer)
(bitsize [(chip ?chip)bitsize])(nbinputs 2)(output
   ?output))
(augment(resource (gensymbol multiplexer
 [(chip ?chip)nbres]))
 (input1 [(resource ?output)in_link2])(input2 ?bus))
(modify(resource ?output)(in_link2 (gensymbol
   multiplexer
 [(chip ?chip)nbres])))
(augment(resource ?output) (input2 ?bus))
(augment (resource ?bus) (outputs ?output))
end



:solution 2

to_do alloc_mux[(resource ?resource)(chip ?chip)]
 do
(alloc_mux_l1(resource ?resource)(chip ?chip))
(alloc_mux_l2(resource ?resource)(chip ?chip))
end


to_do alloc_mux_l2[(resource ?resource)(chip
   ?chip)]
if
[(# [(resource ?resource )input2])<=1]
 do end
to_do alloc_mux_l2[(resource ?resource)(chip
   ?chip)]
if
[(# [(resource ?resource )input2])>1]
 do
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol multiplexer
 [(chip ?chip)nbres]))(category multiplexer)
(bitsize [(chip ?chip)bitsize])(nbinputs 0)(output
   ?resource))
(alloc_l2(resource ?resource)(mux (gensymbol
   multiplexer [(chip ?chip)nbres]))
 (chip ?chip))
end
to_do alloc_l2[(resource ?resource)(mux
   ?mux)(chip ?chip)]
do
(update(inputs [(resource ?resource)input2])
 (resource ?mux)(chip ?chip))
(delete(resource ?resource)(input2 [(resource
   ?resource)input2]))
(augment(resource ?resource)(input2 ?mux))
end
to_do alloc_l1[(resource ?resource)(mux
   ?mux)(chip ?chip)]
do
(update(inputs [(resource ?resource)input1])
 (resource ?mux)(chip ?chip))
(delete(resource ?resource)(input1 [(resource
   ?resource)input1]))
(augment(resource ?resource)(input1 ?mux))
end
to_do alloc_mux_l1[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category | =storage)
[(# [(resource ?resource )input1])>1]
 do
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol multiplexer [(chip
   ?chip)nbres]))(category multiplexer)
(bitsize [(chip ?chip)bitsize])(nbinputs 0)(output
   ?resource))
(alloc_l1(resource ?resource)(mux (gensymbol
   multiplexer [(chip ?chip)nbres]))
```

```
(chip ?chip))
end
to_do delete[(resource ?resource)(input1 ?input)]
if ?res in (resource in '?input)
do
(retract(resource ?resource)(input1 ?res))
end
to_do delete[(resource ?resource)(input1 ?input)]
if ['?input=null]
do
end
to_do delete[(resource ?resource)(input2 ?input)]
if ['?input=null]
do
end
to_do delete[(resource ?resource)(input2 ?input)]
if ?res in (resource in '?input)
do
(retract(resource ?resource)(input2 ?res))
end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])=0]
do
(modify(resource ?resource)(nbinputs 1))
end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])=1]
 do
(modify(resource ?resource)(curinputs 0)(nbinputs
   1))
end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])=2]
 do
(modify(resource ?resource)(curinputs 0)(nbinputs
   2))
end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])=3]
 do
(modify(resource ?resource)(curinputs 0)(nbinputs
   3))
end

to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])>3]
 do
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol multiplexer
 [(chip ?chip)nbres]))(category multiplexer)
(bitsize [(chip ?chip)bitsize])(nbinputs 0)(output
   ?resource))
(update(inputs (rest(rest[(resource
   ?resource)input1]))))
 (resource (gensymbol multiplexer [(chip
   ?chip)nbres]))(chip ?chip))
(delete(resource ?resource)
 (input1 (rest(rest[(resource ?resource)input1])))))
(augment(resource ?resource)(input1 (gensymbol
   multiplexer
 [(chip ?chip)nbres])))
(modify(resource ?resource)(curinputs 0)(nbinputs
   3))
end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category|=storage)
[(# [(resource ?resource )input1])<=1]
do end
to_do alloc_mux_11[(resource ?resource)(chip
   ?chip)]
if
(resource ?resource category=storage)
[(# [(resource ?resource )input1])>0]
[(# [(resource ?resource )input1])<=3]
 do
(modify(resource ?resource)(curinputs 0)
(nbinputs (# [(resource ?resource )input1])))
end

to_do update[(inputs ?inputs)(resource
   ?resource)(chip ?chip)]
if
[(# '?inputs)=2]
do
(augment(resource ?resource)(input1 (first
   '?inputs))
(input2 (first (rest '?inputs))))
end
to_do update[(inputs ?inputs)(resource
   ?resource)(chip ?chip)]
if
[(# '?inputs)>2]
do
(modify(chip ?chip)(nbres [(chip ?chip)nbres]+1))
(make(resource (gensymbol multiplexer
 [(chip ?chip)nbres]))(category multiplexer)
(bitsize [(chip ?chip)bitsize])(nbinputs 0)(output
   ?resource))
(update(inputs (rest '?inputs))
 (resource (gensymbol multiplexer [(chip
   ?chip)nbres]))(chip ?chip))
(augment(resource ?resource)(input1 (first
   '?inputs))
(input2 (gensymbol multiplexer [(chip
   ?chip)nbres])))
end
to_do desc_op(chip ?chip)
do
(open_file(file ?chip)(ext dp)(mode output))
(writeln(file ?chip)(ext dp)" description de la partie
   operative de "?chip)
(writeln(file ?chip)(ext dp)" OPERATEURS |
   NOMBRE | ")
(desc(chip ?chip))
(writeln(file ?chip)(ext dp)" registres "
   (#(resource|category=storage)))
(writeln(file ?chip)(ext dp)" multiplexeurs
   "(#(resource|category=multiplexer)))
(writeln(file ?chip)(ext dp)" bus
   "(#(resource|category=bus)))
(close_file(file ?chip)(ext dp))
end
```

```
;génération du fichier SPOT
to_do desc(chip ?chip)
if ?op in (famille in [(chip ?chip)operations]\'(noop))
do
(writeln(file ?chip)(ext dp)" "?op" "[(chip ?chip)?op])
end
to_do gen_spot(chip ?chip)
 do
(open_file(file ?chip)(ext spot)(mode output))
(writeln(file ?chip)(ext spot)"-- file spot generated
   by ASA)
(writeln(file ?chip)(ext spot)"-- ASA conceived and
   implemented by")
(writeln(file ?chip)(ext spot)" -- Mr A. FONKOUA &
   al. from IMAG(LGI)-CNET ")
(writeln(file ?chip)(ext spot)"ASSEMBLY " ?chip"
   BIT 1 BIT "
[(chip ?chip)bitsize])
(writeln(file ?chip)(ext spot)" -- inputs / outputs
   description ")
(writeln(file ?chip)(ext spot)" INPUTS_OUTPUTS")
(gen_i_o(chip ?chip))
(writeln(file ?chip)(ext spot)" -- operators
   description ")
(writeln(file ?chip)(ext spot)" OPERATORS")
(gen_operators(chip ?chip))
(writeln(file ?chip)(ext spot)" -- wires description ")
(writeln(file ?chip)(ext spot)" WIRES")
(gen_wire(chip ?chip))
(gen_wire_io(chip ?chip))
(writeln(file ?chip)(ext spot)"-- end of the
   description ")
(writeln(file ?chip)(ext spot)" .")
(close_file(file ?chip)(ext spot))
end
to_do gen_i_o(chip ?chip)
if ?param in (parametre)
do
(gen_i_o(parametre ?param)(chip ?chip))
end
to_do gen_i_o[(parametre ?param)(chip ?chip)]
 if (parametre ?param data_type=integer)
 do
(writeln" which side would you like "?param " I/O ? ")
(writeln" answer one of : north south west east ")
(writeln(file ?chip)(ext spot) ?param" "
 [(parametre ?param)mode]" BIT 1 .. BIT "
[(parametre ?param)bitsize]" " (ask atom in
   '(NORTH SOUTH EAST WEST)))
end
to_do gen_i_o[(parametre ?param)(chip ?chip)]
 if (parametre ?param data_type=boolean)
 do
(writeln" which side would you like "?param " I/O ? ")
(writeln" answer one of : north south west east ")
(writeln(file ?chip)(ext spot) ?param " "[(parametre
   ?param)mode]
" BIT 1 "" "
 (ask atom in '(NORTH SOUTH EAST WEST)))
end
to_do gen_operators(chip ?chip)
if ?res in (resource)
do
(gen_operator(resource ?res)(chip ?chip))
end
to_do gen_operator[(resource ?resource)(chip
   ?chip)]
if (resource ?resource category=storage)
 do
 (writeln (file ?chip)(ext spot) ?resource " REGISTER
    (PARI "
 [(resource ?resource)nbinputs] " ) BIT 1 .. BIT "
 [(resource ?resource)bitsize])
 end
 to_do gen_operator[(resource ?resource)(chip
    ?chip)]
 if (resource ?resource category=f_unit)
  do
 (message(resource ?resource)(chip ?chip))
 (writeln (file ?chip)(ext spot) ?resource " ADDER 0
    BIT 1 .. BIT "
 [(resource ?resource)bitsize])
 end
 to_do message[(resource ?res)(chip ?chip)]
 if (resource ?res operation=minus)
 do
 (writeln(file ?chip)(ext spot)"-- "?res " should be a
    subtract operator ")
 end
 to_do message[(resource ?res)(chip ?chip)]
 if (resource ?res operation=mult)
 do
 (writeln(file ?chip)(ext spot)"-- "?res " should be a
    multiply operator ")
 end
 to_do message[(resource ?res)(chip ?chip)]
 do
 end
 to_do gen_operator[(resource ?resource)(chip
    ?chip)]
 if (resource ?resource category=multiplexer)
  do
 (writeln (file ?chip)(ext spot) ?resource "
    MULTIPLEXER 0 BIT 1 .. BIT "
 [(resource ?resource)bitsize])
 end
 to_do gen_wire(chip ?chip)
 if ?res in (resource | input1 | =null)
 do
 (gen_wire(resource ?res)(chip ?chip))
 end
 to_do gen_wire[(resource ?res)(chip ?chip)]
 if (resource ?res input2=null)
 do
 (gen_wire1(resource ?res)(chip ?chip))
 end
 to_do gen_wire[(resource ?res)(chip ?chip)]
 if (resource ?res input2 | =null)
 do
 (gen_wire1(resource ?res)(chip ?chip))
 (gen_wire2(resource ?res)(chip ?chip))
 end
 to_do gen_wire1[(resource ?resource)(chip ?chip)]
 if ?res in (resource in [(resource ?resource)input1])
 do
 (add_wire(output ?res)(input1 ?resource)(chip
    ?chip))
 end
 to_do gen_wire2[(resource ?resource)(chip ?chip)]
 if ?res in (resource in [(resource ?resource)input2])
 do
 (add_wire(output ?res)(input2 ?resource)(chip
    ?chip))
 end
 to_do add_wire[(output ?output)(input1
    ?input)(chip ?chip)]
 if (resource ?input category=storage,nbinputs>1)
  (resource ?output category in '(multiplexer f_unit))
 do
```

```
(modify(resource ?input)(curinputs [(resource
   ?input)curinputs]+1))
(writeln(file ?chip)(ext spot)?input" D"[(resource
   ?input)curinputs]
 " BIT 1 .. "[(resource ?input)bitsize]
" = " ?output" S BIT 1.. "[(resource ?output)bitsize] " ; ")
end
to_do add_wire[(output ?output)(input1
   ?input)(chip ?chip)]
if (resource ?input category=storage,nbinputs=1)
 (resource ?output category in '(multiplexer f_unit))
do
(modify(resource ?input)(curinputs [(resource
   ?input)curinputs]+1))
(writeln(file ?chip)(ext spot)?input" D BIT 1.. BIT "
[(resource ?input)bitsize]
 " = "?output" S BIT 1 .. "[(resource ?output)bitsize] " ; ")
end

to_do add_wire[(output ?output)(input1
   ?input)(chip ?chip)]
if (resource ?output category=storage)
 (resource ?input category in '(f_unit multiplexer))
do
(writeln(file ?chip)(ext spot)?input" A BIT 1.. "
[(resource ?input)bitsize]
 " = "?output" Q BIT 1 .. "[(resource ?output)bitsize] " ; ")
end
to_do add_wire[(output ?output)(input2
   ?input)(chip ?chip)]
if (resource ?output category=storage)
 (resource ?input category in '(multiplexer f_unit))
do
(writeln(file ?chip)(ext spot)?input" B BIT 1.. "
[(resource ?input)bitsize]
 " = "?output" Q BIT 1 .. "[(resource ?input)bitsize] " ; ")
end
to_do add_wire[(output ?output)(input1
   ?input)(chip ?chip)]
if (resource ?output category in '(multiplexer funit))
 (resource ?input category in '(f_unit multiplexer))
do
(writeln(file ?chip)(ext spot)?input" A BIT 1..
   "[(resource ?input)bitsize]
 " = "?output" S BIT 1 .. "[(resource ?output)bitsize] " ; ")
end
to_do add_wire[(output ?output)(input2
   ?input)(chip ?chip)]
if (resource ?output category in '(multiplexer
   f_unit))
 (resource ?input category in '(f_unit multiplexer))
do
(writeln(file ?chip)(ext spot)?input" B BIT 1..
   "[(resource ?input)bitsize]
 " = "?output" S BIT 1 .. "[(resource ?output)bitsize] " ; ")
end
to_do gen_wire_io(chip ?chip)
do
(writeln(file ?chip)(ext spot)"-- I/O wire not yet
   implemented")
end
```

Annexe C : Spécification de quelques paquetages ADA constituant ODSE

Dans cette annexe, nous donnons les sources des paquetages :

—OBJECTS

—FACTBASE (base de faits)

—RULES (base de règles)

—HIPLAN (interface de ODSE)

C.1 Objets : paquetage OBJECTS

```
------------------------------------------------------------------------
----
--package objects
--description : contains the specification of "object"
--              fundamental bloc of the  ODSE system. An object
--              is characterized by
--              - a set of attributes
--              - a set of constraints on its attributes
--              - its dependencies properties :
--                    - justiers : data on which it depends
--                    - justificands : a set of data on which it depends
--
------------------------------------------------------------------------
with symbolic_expression,text_io,integer_text_io;
use text_io,integer_text_io;

    package objects is
  package  Se renames Symbolic_Expression;
  package  Tio renames Text_Io;
  package  Itio renames Integer_Text_Io;

      subtype object is SE.S_Expr;
      Null_object : object renames se.null_s_expr;
------------------------------------------------------------------------

--    Function:     Is_object
--    Description:  Determines if s_expr_arg is an object
--    Exceptions Raised:  None.

      function Is_object (s_expr_arg : in se.s_expr) return Boolean
------------------------------------------------------------------------

--    Function:     Is_Null
--    Description:  Determines if object_Arg = Null_object.
--    Exceptions Raised:  None.

      function Is_Null (object_Arg : in object) return Boolean;

------------------------------------------------------------------------

--    Function: Is_Equal
--    Description:  Determines if two objects are equal by determining
```

```
--                  if their instantiations are equal.
--    Exceptions Raised:  None.

function Is_Equal (object1, object2 : in object) return Boolean;

-------------------------------------------------------------------------

--    Function:     Create_object
--    Description:  Creates a object. The symbolic expression forms the
--                  object's template and the object's variable binding
--                  context is set to null.
--    Exceptions Raised:  None.

      function Create_object (Template : in SE.S_Expr;
                        constraints,justifier,
                           justificand : in SE.S_Expr:= SE.Null_S_Expr)
       return object;
-------------------------------------------------------------------------

--    Function:     Get_Template
--    Description:  Returns the template of the given object: current set
--                  of attributes characterizing the object.
--    Exceptions Raised:  None.

      function Get_Template (object_Arg : in object) return SE.S_Expr;


      procedure Bind (Current_Value : in out object;  New_Value : in
object)
       renames SE.Bind;

      procedure Free (object_Arg : in out object) renames SE.Free;

      function Return_And_Free (object_Arg : in object) return object
       renames SE.Return_And_Free;


-------------------------------------------------------------------------

--    Procedure:    Get
--    Description:  Read a object from the specified input file.
--    Exceptions Raised: None.

      procedure Get (Input_File : in File_Type;
                object_Result : in out object);

-------------------------------------------------------------------------

--    Procedure:    Get
--    Description:  Read an object from the current default input file.
--    Exceptions Raised: None.

      procedure Get (object_Result : in out object);

-------------------------------------------------------------------------

--    Procedure:    Put
--    Description:  Print the structure of the input object
--                  to the specified output file.
--    Exceptions Raised:  None.
```

```
      procedure Put (Output_File : in File_Type; object_Arg : in
object);

-------------------------------------------------------------------------

--    Procedure:    Put
--    Description:  Print the structure of the input object
--                  to the current default output file.
--    Exceptions Raised: None.

      procedure Put (object_Arg : in object);
-------------------------------------------------------------------------
-- function get_name
-- description : returns the name of the object
 function get_name(object_arg: in object) return se.s_expr;
function set_name(name:in se.s_expr;obj:object) return object;
function set_class(idf:se.s_expr;obj:object) return object;
-------------------------------------------------------------------------
-
-- returns the class of the object
 function get_class(object_arg: in object) return se.s_expr;
-------------------------------------------------------------------------
-
-- the following functions help in debugging the expert in construction.
--function : set_trace_on
--description : mark the object_arg for displaying to the screen all
changes
-- occurring in any of its features.
function set_trace_on(object_arg:in object) return object;
-------------------------------------------------------------------------
-- function set_trace_off
-- disables object_arg trace option
function set_trace_off(object_arg:in object) return object;
-------------------------------------------------------------------
--procedure set_trace_off disables trace option for all objects
-- until the next set_trace_on is applied to an object in which case
-- all previously marked for trace are displayed again.
procedure set_trace_off;
-------------------------------------------------------------------------
--function get_attr
--description : returns the value of the attribute attr of the object
--              whose full description is provided as second argument
-------------------------------------------------------------------------
-
  function Get_Attr(Attr_arg : IN Se.S_Expr;
                    object_arg    : IN object) RETURN Se.S_Expr;
-------------------------------------------------------------------------
--
--function augment
--description : augment the attribute_arg of object object_arg with the
value
--              value_arg and returns the resulting object
--              modified obj.
-------------------------------------------------------------------------
-
  function set_attr( Attr, Val : IN Se.S_Expr;
                     object_arg : in object) RETURN object;
  function augment(Attr, Val : IN Se.S_Expr;
                     object_arg: in object) RETURN object;
  function retract(Attr, Val : IN Se.S_Expr;
                     object_arg:in object) RETURN object;
```

```
  function remove(Attr : IN Se.S_Expr;
                  object_arg:in object) RETURN object;
 function add_attr(attr,val:se.s_expr;obj:object) return object;
 function incr_attr(val,attr:se.s_expr;obj:object) return object;
 function decr_attr(val,attr:se.s_expr;obj:object) return object;
-------------------------------------------------------------------------
-
-- functions for manipulating graph structured objects
-------------------------------------------------------------------------
-
function add_successor(name:in se.s_expr; node: in object) return
object;
function retract_successor(name:se.s_expr; node:in object) return
object;
function add_predecessor(name:in se.s_expr; node: in object) return
object;
function retract_predecessor(name: in se.s_expr;node:in object)
                                 return object;
function set_successors(l_n:in se.s_expr;node:in object) return object;
function set_predecessors(l_n:in se.s_expr;node: in object)
                                 return object;
function get_successors(object_arg:in object) return se.s_expr;
function get_predecessors(object_arg:in object) return object;
function is_leave(node:in object) return boolean;
function is_head(node:in object) return boolean;
function is_node(obj_arg: in object) return boolean;
-------------------------------------------------------------------------
-
--
--               FUNCTIONS DEALING WITH CONSTRAINT
--
-------------------------------------------------------------------------
-
-------------------------------------------------------------------------
-
--function : get_constraint
--description : returns the constraint value field of the different
--              object_arg attributes
--              the description of the object whose full description is
--              given.
-------------------------------------------------------------------------
  function Get_constraints( Object_arg    : IN object) RETURN Se.S_Expr;
-------------------------------------------------------------------------
-
-- function : get_constraints
-- description : returns the constraint field of attribute attribute_arg
--               of object_arg
-------------------------------------------------------------------------
  function get_constraint(attribute_argument: in se.s_expr;
                          object_arg     : IN OBJECT)
                                           RETURN se.s_expr;
  function get_constraint(attr_arg: in se.s_expr;
                       constr:in se.s_expr;
                          object_arg     : IN OBJECT)
                                           RETURN se.s_expr;
-------------------------------------------------------------------------
--function : is_constrained
--description : returns true if any constraint field is not null
--              false otherwise
-------------------------------------------------------------------------
  function is_constrained ( object_arg     : IN object)
```

```
                                                  return boolean;
-------------------------------------------------------------------------
--function : is_constrained
--description : checks the attribute attr of object_arg for constraints
-------------------------------------------------------------------------
  function is_constrained ( attr:in se.s_expr;
              object_arg    : IN object) return boolean;
  function lower_bound(attr:in se.s_expr;obj:in object) return
se.s_expr;
  function set_lower_bound(attr,val:in se.s_expr;obj:object)
                    return object;
  function set_upper_bound(attr,val:in se.s_expr;obj:in object)
                    return object;
  function set_in_set(attr,val:in se.s_expr;obj: in object) return
         object;
  function set_not_in_set(attr,val:in se.s_expr;obj:in object)
                      return object;
  function upper_bound(attr:in se.s_expr;obj:in object) return
se.s_expr;
  function in_set(attr:in se.s_expr;obj: in object) return se.s_expr;
  function not_in_set(attr:in se.s_expr;obj:in object) return
se.s_expr;
-------------------------------------------------------------------------
-- function : augment
-- description : augment the global constraint field of object_arg with
--               the value constraint_arg
-------------------------------------------------------------------------
  function add_constraint(constraint_arg : in se.s_expr;
                              object_arg : in object)return object;
-------------------------------------------------------------------------
-- function : add_constraint
-- description : augments the attribute_arg constraint field of
object_arg with
--               value constraint_arg
-------------------------------------------------------------------------
  function add_constraint(constraint_arg : in se.s_expr;
                              attribute_arg :in se.s_expr;
                              object_arg : in object
                                                      ) return
object;

function add_constraint( attr,cond,val:se.s_expr;
                        object_arg:in object) return object;
procedure add_constraint( attr,cond,val:se.s_expr;
                        object_arg:in out object;modif:in out boolean);
-------------------------------------------------------------------------
-- function : set_constraint
-- description : sets the constraint_field of attr_arg to value
--               constraint_arg. The previous value is discarded
-------------------------------------------------------------------------
  function set_constraint(attribute_arg,constraint_arg
                      :in se.s_expr;object_arg:in object) return object;
  function set_constraint(attr_arg,const_arg,val
                      :in se.s_expr;object_arg:in object) return object;
-------------------------------------------------------------------------
-- function : relax_constraint
-- description : suppresses the constraint_arg from attr_arg
--               constraints set.
-------------------------------------------------------------------------
  function relax_constraint(attribute_arg,constraint_arg
                      :in se.s_expr;object_arg:in object) return object;
  function relax_constraint(attribute_arg,cond,val
```

```
                        :in se.s_expr;object_arg:in object) return object;
-------------------------------------------------------------------------
-- functions to manipulate data dependency field of objects
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-
--function : get_justificand
--description : returns the justificand field of object_arg.
--exception raised :none;
-------------------------------------------------------------------------
  function Get_justificand( object_arg     : IN object) RETURN
Se.S_Expr;
-------------------------------------------------------------------------
-
--function : get_justifier
--description : returns the justifier field of object_arg.
--exception raised :none;
-------------------------------------------------------------------------
  function Get_justifier( object_arg     : IN object) RETURN Se.S_Expr;
-------------------------------------------------------------------------
--function : is_justified
--description : returns true if justifier field of object_arg is not null
--              false otherwise
-------------------------------------------------------------------------
  function is_justified ( object_arg     : IN object) return boolean;
-------------------------------------------------------------------------
--function : is_justifying
--description : returns true if justificand field of object_arg is not
null
--              false otherwise
-------------------------------------------------------------------------
  function is_justifying ( object_arg     : IN object) return boolean;
-------------------------------------------------------------------------
-- function : add_justificand
-- description : augment the justificand field of object_arg.
-- exceptions raised : none.
-------------------------------------------------------------------------
  function add_justificand(justificand_arg : in se.s_expr;
                           object_arg : in object)return object;
-------------------------------------------------------------------------
-- function : add_justifier
-- description : augment the justifier field of object_arg.
-- exceptions raised : none.
-------------------------------------------------------------------------
  function add_justifier(justifier_arg : in se.s_expr;
                           object_arg : in object) return object;

-------------------------------------------------------------------------
-- function : set_justifier
-- description : sets the justifier field of object_arg to value
--               justifier_arg. The previous value is discarded
-------------------------------------------------------------------------
  function set_justifier(justifier_arg:se.s_expr;
                         object_arg:in object) return object;
-------------------------------------------------------------------------
-- function : retract_justifier
-- description : retracts  justifier_arg from justifiers set of
object_arg
-------------------------------------------------------------------------
  function retract_justifier(justifier_arg:se.s_expr;
                         object_arg:in object) return object;
-------------------------------------------------------------------------
```

```
-- function : set_justificand
-- description : sets the justificand field of object_arg to value
--               justificand_arg. The previous value is discarded
-------------------------------------------------------------------------
  function set_justificand(justificand_arg:se.s_expr;
                           object_arg:in object) return object;
-------------------------------------------------------------------------
-- function : retract_justificand
-- description : retracts  justificand_arg from justificands set of
object_arg
-------------------------------------------------------------------------
  function retract_justificand(justificand_arg:se.s_expr;
                           object_arg:in object) return object;
-------------------------------------------------------------------------
--FUNCTIONS FOR EDITING OBJECTS
-------------------------------------------------------------------
    procedure Print_Obj(Print_Arg : IN object);
    procedure GET_OBJECT(OBJ:IN OUT OBJECT);
-------------------------------------------------------------------
--generic functions for creating and manipulating objects
--
-------------------------------------------------------------------
--for generating function to create an object of  given class classname
--without name
generic
classname:in string;
function create_object2(template:se.s_expr) return object;
-------------------------------------------------------------------------
--for generating function to create an object of  given class classname
--with name objname
generic
classname:in string;
function create_object3(objname:string;template:se.s_expr) return
object;
-------------------------------------------------------------------------
--for generating fonction to get the value of attribute attr of a given
--object;
generic
attr:string;
function get_attr1(obj:object) return se.s_expr;
-------------------------------------------------------------------------
--for generating fonction to get the value of attribute attr of a given
--object;
generic
attr:string;
function retract1(val:se.s_expr;obj:object) return object;
-------------------------------------------------------------------------
--for generating fonction to get the value of attribute attr of a given
--object;
generic
attr:string;
function set_attr1(val:se.s_expr;obj:object) return object;
-------------------------------------------------------------------------
--for generating fonction to get the value of attribute attr of a given
--object;
generic
attr:string;
function augment1(val:se.s_expr;obj:object) return object;
-------------------------------------------------------------------
--for generating function to create an object of  given class classname
--with name objname
function add_pattern(patname:string;
```

```
                    arg:se.s_expr;template:se.s_expr) return se.s_expr;
-------------------------------------------------------------------------
-- particular features ob objects
function set_pattern(val:se.s_expr;obj:object) return object;
function get_pattern (obj:object) return se.s_expr;
function set_var_object(val:se.s_expr;obj:object) return object;
function get_var_object(obj:object) return se.s_expr;
function set_scope (val:se.s_expr;obj:object) return object;
function get_scope  (obj:object) return se.s_expr;
function set_object_class (val:se.s_expr;obj:object) return object;
function get_object_class  (obj:object) return se.s_expr;
function set_threshold (val:se.s_expr;obj:object) return object;
function get_threshold  (obj:object) return se.s_expr;
function get_weight  (obj:object) return se.s_expr;
function set_weight  (val:se.s_expr;obj:object) return object;
function get_max_weight  (obj:object) return se.s_expr;
function set_max_weight  (val:se.s_expr;obj:object) return object;
function get_min_weight  (obj:object) return se.s_expr;
function set_min_weight (val:se.s_expr;obj:object) return object;
function set_var_domain(val:se.s_expr;obj:object) return object;
function get_var_domain(obj:object) return se.s_expr;
function get_satisfied(obj:object) return se.s_expr;
function set_satisfied(val:se.s_expr;obj:object) return object;
function get_bindings(obj:object) return se.s_expr;
function set_bindings(val:se.s_expr;obj:object) return object;
function get_pathlength(obj:object) return se.s_expr;
function set_pathlength(val:se.s_expr;obj:object) return object;
function add_pathlength(val:se.s_expr;obj:object) return object;
function get_ready(obj:object) return se.s_expr;
function set_ready(val:se.s_expr;obj:object) return object;
function get_quality(obj:object) return se.s_expr;
function set_quality(val:se.s_expr;obj:object) return object;
function get_parameters(obj:object) return se.s_expr;
function remove_parameters(obj:object) return object;
function set_parameters(val:se.s_expr;obj:object) return object;
function get_predname(obj:object) return se.s_expr;
function set_predname(val:se.s_expr;obj:object) return object;
function get_opname(obj:object) return se.s_expr;
function set_opname(val:se.s_expr;obj:object) return object;
function get_falsepred(obj:object) return se.s_expr;
function set_falsepred(val:se.s_expr;obj:object) return object;
function get_truepred(obj:object) return se.s_expr;
function set_truepred(val:se.s_expr;obj:object) return object;
function set_attribute(val:se.s_expr;ob:object) return object;
function get_attribute(ob:object) return se.s_expr;
function set_event(val:se.s_expr;ob:object) return object;
function get_event(ob:object) return se.s_expr;
invalid_object_format,not_in_set_constraint_violation,
in_set_constraint_violation,le_constraint_violation,lt_constraint_violati
on,
ge_constraint_violation,gt_constraint_violation,
constraining_attribute_to_impossible_value: exception;
  end objects;

with  pattern,symbolic_expression,objects,class,text_io,assoc_pkg;
use   text_io;
```

C.2 Base de faits :paquetage FACTBASE

```
package  factbase is
subtype  index_range is natural range 0..16;
package  ob renames objects;
package  se renames symbolic_expression;
package  pat renames pattern;
package  co renames class;
subtype  list_iter is co.list_iter;

package  hp is new  assoc_pkg(co.collection);
subtype  factbase is hp.assoc_world;
subtype  names_iter is co.names_Iter;--| Bound names of
                                     --| objects in arbitrary order.
subtype  objects_iter is co.objects_Iter;--| Bound objects in arbitrary
order.
subtype  Bindings_Iter is co.bindings_iter;--| names,objects pairs
                                           --| in arbitrary order
function is_empty(fctbase:in factbase) return boolean;
function is_class(clname:se.s_expr;fctbase:in factbase) return boolean;
function size(fctbase: in factbase) return natural;
-------------------------------------------------------------------------

--    Function:     Free
--    Description:  Frees the given collection.
--    Exceptions Raised:  None.
      procedure Free (fctbase:in out factbase);
procedure add_collection(fctbase: in out factbase;collection:in
co.collection);
function get_class(collname:se.s_expr;fctbase:factbase) return
co.collection;
procedure add_collection(fctbase: in out factbase;collection:in
ob.object);
procedure bind_object(object1:in ob.object;
                      fctbase:in out factbase);
function get_object(obj_template:se.s_expr;fctbase:in factbase)
                                    return ob.object;
--  FUNCTION Make_names_iter(collection: in se.s_expr;
--                           fctbase:in factbase) RETURN names_iter;

  function More(Iter : IN objects_iter) RETURN Boolean renames
co.more;


--  PROCEDURE Next_name(Iter : IN OUT names_iter;
--                      name  : in OUT se.s_expr) renames co.next;


  function Make_objects_iter(collection: in se.s_expr;
         fctbase:in factbase) RETURN objects_iter;


  --FUNCTION More(Iter : IN objects_iter) RETURN Boolean renames
co.more;


  procedure Next(Iter : IN OUT objects_iter;
                 Val  : in OUT ob.object) renames co.next;
```

```
--  FUNCTION More(Iter : IN Bindings_Iter) RETURN Boolean renames
co.more;


  procedure Next(Iter : IN OUT Bindings_Iter;
                 name  : in OUT se.s_expr;
                 object1 :in OUT ob.object) renames co.next;

-------------------------------------------------------------------------

--     Procedure:     Get
--     Description:  Read a pattern from the specified input file.
--     Exceptions Raised: None.

       procedure Get (Input_File : in File_Type;
                      fctbase : in out factbase);

-------------------------------------------------------------------------

--     Procedure:     Get
--     Description:  Read a pattern from the current default input file.
--     Exceptions Raised: None.

       procedure Get (fctbase : in out factbase);

-------------------------------------------------------------------------

--     Procedure:     Put
--     Description:  Print the structure of the input pattern
--                   to the specified output file.
--     Exceptions Raised:  None.

       procedure Put (Output_File : in File_Type; fctbase : in
factbase);

-------------------------------------------------------------------------

       --     Procedure:     Put
       --     Description:  Print the structure of the input pattern
       --                   to the current default output file.
       --     Exceptions Raised: None.

       procedure Put (fctbase : in factbase);
-------------------------------------------------------------------
--description : takes a pattern as arguments and returns the complete
list
--              of bindings corresponding to successful matches of the
pattern
--              against the elements of the factbase;
function eval_set(patc:in ob.object;fctbase:in factbase) return
se.s_expr;
-------------------------------------------------------------------------
-----
-----------------   ACTIONS ON OBJECTS OF THE BASE -------------------
-------------------------------------------------------------------------
----
-- this generic procedure provides the means to apply actions described
in
```

```
-- ada to objects of a given collection. Any modification can be applied
-- to the object whose name is given through the user-defied function
func.
-- But it is the user responsibility to return an object of the
collection.
-- The object returned might be different from the object on which func
-- has been applied in which case by the end of apply_func, both objects
will
-- be present in the collection.
generic
--same the produced object is of the same collection!
with function func(obj:in ob.object) return ob.object;
procedure apply_func1( object_template:se.s_expr;
                       fctbase:in out factbase);
generic
--func should return a list of objects. all the objects returned should
-- have the attribute _class defined if they belong to a precise
collection.
with function func(list_arg:in ob.object) return se.s_expr;
procedure apply_func2( largs:se.s_expr;
                      fctbase:in out factbase);

------------------------------------------------------------------------

--    Function:     Get_Template
--    Description:  Returns the template of the given object: current set
--                  of attributes characterizing the object.
--    Exceptions Raised:  None.
--function get_attr
--description : returns the value of the attribute attr of the object
--              whose full description is provided as second argument
------------------------------------------------------------------------
-
  function Get_Attr(Attr_arg : IN Se.S_Expr;
                    object_template    : IN se.s_expr;
                    fctbase: in factbase) RETURN Se.S_Expr;
------------------------------------------------------------------------
--
--function augment
--description : augment the attribute_arg of object object_arg with the
value
--              value_arg and returns the resulting object
--              modified obj.-----------------------------------------
------------------------------
  procedure set_attr( Attr, Val : IN Se.S_Expr;
                      object_template : in se.s_expr;
                       fctbase: in out factbase);
  procedure augment(Attr, Val : IN Se.S_Expr;
                    object_template: in se.s_expr;
                    fctbase: in out factbase);
  procedure retract(Attr, Val : IN Se.S_Expr;
                    object_arg:in se.s_expr;
                    fctbase:in out factbase);
  procedure remove(object_arg:in se.s_expr;
                    fctbase:in out factbase);
  procedure make(object_arg:in se.s_expr;
                    fctbase:in out factbase);
------------------------------------------------------------------------
-
--
--                FUNCTIONS DEALING WITH CONSTRAINT
--
```

```
-------------------------------------------------------------------------
-
-------------------------------------------------------------------------
-
--function : get_constraint
--description : returns the constraint value field of the different
--              object_arg attributes
--              the description of the object whose full description is
--              given.
-------------------------------------------------------------------------
  function Get_constraints( Object_template    : IN se.s_expr;
                            fctbase: in factbase) RETURN Se.S_Expr;
-------------------------------------------------------------------------
-
-- function : get_constraints
-- description : returns the constraint field of attribute attribute_arg
--               of object_arg
-------------------------------------------------------------------------
  function get_constraint(attribute_argument: in se.s_expr;
                              object_template    : IN se.s_expr;
                               fctbase: in factbase)
                                                   RETURN se.s_expr;
  function get_constraint(attr_arg: in se.s_expr;
                          constr:in se.s_expr;
                               object_template    : IN se.s_expr;
                                fctbase:in factbase)
                                                   RETURN se.s_expr;
-------------------------------------------------------------------------
--function : is_constrained
--description : returns true if any constraint field is not null
--              false otherwise
-------------------------------------------------------------------------
  function is_constrained ( object_template    : IN se.s_expr;
                            fctbase: in factbase)
                                          return boolean;
-------------------------------------------------------------------------
--function : is_constrained
--description : checks the attribute attr of object_arg for constraints
-------------------------------------------------------------------------
  function is_constrained ( attr:in se.s_expr;
              object_template    : IN se.s_expr;
               fctbase: in factbase) return boolean;
-------------------------------------------------------------------------
-------------------------------------------------------------------------
  procedure add_constraint(constraint_arg : in se.s_expr;
                               object_template : in se.s_expr;
                                fctbase:in out factbase);
-------------------------------------------------------------------------
-- function : add_constraint
-- description : augments the attribute_arg constraint field of
object_arg with
--               value constraint_arg
-------------------------------------------------------------------------
  procedure add_constraint(constraint_arg : in se.s_expr;
                               attribute_arg :in se.s_expr;
                               object_template : in se.s_expr;
                                fctbase: in out factbase);
procedure add_constraint( attr,cond,val:se.s_expr;
                          objname:in se.s_expr;fctbase:in out factbase);
procedure add_constraint( attr,cond,val:se.s_expr;
                          objname:in se.s_expr;
                          fctbase:in out factbase;modif:in out boolean);
```

```
-------------------------------------------------------------------------
-- function : set_constraint
-- description : sets the constraint_field of attr_arg to value
--               constraint_arg. The previous value is discarded
-------------------------------------------------------------------------
  procedure set_constraint(attribute_arg,constraint_arg:in se.s_expr;
                                          object_template:in se.s_expr;
                                          fctbase: in out factbase);
-------------------------------------------------------------------------
-- function : relax_constraint
-- description : suppresses the constraint_arg from attr_arg
--               constraints set.
-------------------------------------------------------------------------
  procedure relax_constraint(attribute_arg,constraint_arg
                     :in se.s_expr;object_template:in se.s_expr;
                      fctbase: in out factbase);
  procedure relax_constraint(attr,cond,val :in se.s_expr;
              objname:in se.s_expr;
               fctbase: in out factbase);
-------------------------------------------------------------------------
-- functions to manipulate data dependency field of objects
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-
--function : get_justificand
--description : returns the justificand field of object_arg.
--exception raised :none;
-------------------------------------------------------------------------
  function Get_justificand( object_template    : IN se.s_expr;
                            fctbase : in factbase) RETURN Se.S_Expr;
-------------------------------------------------------------------------
-
--function : get_justifier
--description : returns the justifier field of object_arg.
--exception raised :none;
-------------------------------------------------------------------------
  function Get_justifier( object_template     : IN se.s_expr;
                          fctbase: in factbase) RETURN Se.S_Expr;
-------------------------------------------------------------------------
--function : is_justified
--description : returns true if justifier field of object_arg is not null
--              false otherwise
-------------------------------------------------------------------------
  function is_justified ( object_template    : IN se.s_expr;
                          fctbase : in factbase) return boolean;
-------------------------------------------------------------------------
--function : is_justifying
--description : returns true if justificand field of object_arg is not
null
--              false otherwise
-------------------------------------------------------------------------
  function is_justifying ( object_template     : IN se.s_expr;
                           fctbase : in factbase) return boolean;
-------------------------------------------------------------------------
-- function : add_justificand
-- description : augment the justificand field of object_arg.
-- exceptions raised : none.
-------------------------------------------------------------------------
  procedure add_justificand(justificand_arg : in se.s_expr;
                            object_template : in se.s_expr;
                            fctbase: in out factbase);
-------------------------------------------------------------------------
```

```
-- function : add_justifier
-- description : augment the justifier field of object_arg.
-- exceptions raised : none.
-------------------------------------------------------------------
  procedure add_justifier(justifier_arg : in se.s_expr;
                          object_template : in se.s_expr;
                          fctbase:in out factbase);

-------------------------------------------------------------------
-- function : set_justifier
-- description : sets the justifier field of object_arg to value
--               justifier_arg. The previous value is discarded
-------------------------------------------------------------------
  procedure set_justifier(justifier_arg:se.s_expr;
                          object_arg:in se.s_expr;
                          fctbase: in out factbase);
-------------------------------------------------------------------
-- function : retract_justifier
-- description : retracts  justifier_arg from justifiers set of
object_arg
-------------------------------------------------------------------
  procedure retract_justifier(justifier_arg:se.s_expr;
                          object_template:in se.s_expr;
                          fctbase: in out factbase);
-------------------------------------------------------------------
-- function : set_justificand
-- description : sets the justificand field of object_arg to value
--               justificand_arg. The previous value is discarded
-------------------------------------------------------------------
  procedure set_justificand(justificand_arg:se.s_expr;
                          object_template:in se.s_expr;
                          fctbase: in out factbase);
-------------------------------------------------------------------
-- function : retract_justificand
-- description : retracts  justificand_arg from justificands set of
object_arg
-------------------------------------------------------------------
  procedure retract_justificand(justificand_arg:se.s_expr;
                          object_template:in se.s_expr;
                          fctbase: in out factbase);
-------------------------------------------------------------------
-------------------------------------------------------------------
-- MANIPULATING GRAPH STRUCTURED OBJECTS
-------------------------------------------------------------------
-------------------------------------------------------------------
-
-- functions for manipulating graph structured objects
-------------------------------------------------------------------
-
procedure add_successor(name:in se.s_expr; node: in se.s_expr;
                          fctbase:in out factbase);
procedure retract_successor(name:se.s_expr; node:in se.s_expr;
                          fctbase: in out factbase);
procedure add_predecessor(name:in se.s_expr; node: in se.s_expr;
                          fctbase:in out factbase);
procedure retract_predecessor(name: in se.s_expr;node:in se.s_expr;
                          fctbase: in out factbase);
procedure set_successors(l_n:in se.s_expr;node:in se.s_expr;
                          fctbase : in out factbase);
procedure set_predecessors(l_n:in se.s_expr;node: in se.s_expr;
                          fctbase : in out factbase);
function get_successors(object_arg:in se.s_expr;
```

```
                                    fctbase : in factbase) return
se.s_expr;
function get_predecessors(object_arg:in se.s_expr;
                                    fctbase : in factbase) return
se.s_expr;
function is_leave(node:in se.s_expr;fctbase : in factbase) return
boolean;
function is_head(node:in se.s_expr;fctbase : in factbase) return
boolean;
function is_node(obj_arg: in se.s_expr;
                        fctbase: in factbase) return boolean;
function make_successors_iter(node:in se.s_expr;fctbase: in factbase)
                                        return list_iter;
function make_predecessors_iter(node:in se.s_expr;
                            fctbase: in factbase) return list_iter;
function make_list_iter(list_names:in se.s_expr;
                            fctbase: in factbase) return list_iter;
function more(iter: in list_iter) return boolean renames co.more;
procedure next(iter:in out list_iter;obj:in out ob.object) renames
co.next;
-------------------------------------------------------------------------
-- DEBUGGING FUNCTIONS
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-
-- the following functions help in debugging the expert in construction.
--function : set_trace_on
--description : mark the object_arg for displaying to the screen all
changes
-- occurring in any of its features.
procedure set_trace_on(object_arg:in se.s_expr;
                    fctbase: in out factbase);
-------------------------------------------------------------------------
-- function set_trace_off
-- disables object_arg trace option
procedure set_trace_off(object_arg:in se.s_expr;
                        fctbase : in out factbase);
-------------------------------------------------------------------
--procedure set_trace_off disables trace option for all objects
-- until the next set_trace_on is applied to an object in which case
-- all previously marked for trace are displayed again.
procedure set_trace_off renames ob.set_trace_off;
procedure print_template(col:se.s_expr;fctbase:factbase);
--FUNCTION Eval (exp:se.s_expr;fctbase:factbase) RETURN Se.S_Expr;
unknown_collection:exception;
end factbase;


with  symbolic_expression,objects,text_io,integer_text_io,factbase,
     class;
use  text_io,integer_text_io;
```

C.3 Base de règles : paquetage RULES

```
package  rules is
package  se renames symbolic_expression;
package  ob renames objects;
package  co renames class;
```

```
package  fct renames factbase;
type rulebase is private;
type operators_iter is private;
procedure add_deamon(trig,rule:in ob.object;
          rbase:in out rulebase);
procedure add_theorem(trig,rule:ob.object;rbase:in out rulebase);
procedure add_operator(trig,rule:ob.object;rbase:in out rulebase);
procedure add_routine(trig:ob.object;rbase:in out rulebase);
function fetch_theorem(trig:ob.object;rbase:rulebase) return ob.object;
function fetch_operator(trig:ob.object;rbase:rulebase)return ob.object;
function is_routine(format:se.s_expr;rbase:rulebase)return boolean;
function routine_num(format:se.s_expr;rbase:rulebase)return natural;
-------------------------------------------------------------------------

--     Procedure:     Get
--     Description:  Read a pattern from the specified input file.
--     Exceptions Raised: None.

       procedure Get (Input_File : in File_Type;
                 rbase : in out rulebase);

-------------------------------------------------------------------------

--     Procedure:     Get
--     Description:  Read a pattern from the current default input file.
--     Exceptions Raised: None.

       procedure Get (rbase : in out rulebase);

-------------------------------------------------------------------------

--     Procedure:     Put
--     Description:  Print the structure of the input pattern
--                   to the specified output file.
--     Exceptions Raised:  None.

       procedure Put (Output_File : in File_Type; rbase : in rulebase);

-------------------------------------------------------------------------

--     Procedure:     Put
--     Description:  Print the structure of the input pattern
--                   to the current default output file.
--     Exceptions Raised: None.

       procedure Put (rbase : in out rulebase);

-------------------------------------------------------------------
--function eval_deamons
--description : evaluates the set of deamon rules triggered by event
--             and return instantiations of all rhs of rules found
--             firable.
function eval(exp:in se.s_expr;
              fctbase:in fct.factbase;
              rbase:in rulebase)return se.s_expr;
function get_active_deamons(event:in ob.object;fctbase:in fct.factbase;
                    rbase:in rulebase) return se.s_expr;
function find_way(op:in se.s_expr;
    fctbase:fct.factbase;rbase:in rulebase) return se.s_expr;
-----------------------------------------------------
procedure set_asleep(rname:se.s_expr;rbase: in out rulebase);
procedure set_alive(rname:se.s_expr;rbase:in out rulebase);
```

```
procedure set_break_on(rname:se.s_expr;rbase:in out rulebase);
procedure set_break_off(rname:se.s_expr;rbase:in out rulebase);
procedure set_break_off(rbase:in out rulebase);
procedure set_break_on(rbase:in out rulebase);
function is_break_active_on(rbase:in rulebase) return boolean;
function is_asleep (act:in ob.object) return boolean;
function is_asleep (rname:in se.s_expr;rbase:in rulebase) return
boolean;
function is_break_active_on(act:ob.object) return boolean;
function is_break_active_on (rname:in se.s_expr;
                             rbase:in rulebase) return boolean;
procedure init_rulebase(rbase:in out rulebase);
-------------------------------------------------------
function get_number_of_theorems(rbase:in rulebase) return natural;
function get_number_of_deamons(rbase:in rulebase) return natural;
function get_number_of_operators(rbase:in rulebase) return natural;
unknown_operator,out_of_my_skills,invalid_condition,no_more_operators,
not_allowed:exception;
private
type operators_iter is
record
operators:ob.object;
more:boolean:=false;
end record;
type rulebase is
record
deamons,operators,theorems:co.collection;
nbdeamons,nbtheorems,nboperators:natural:=0;
break_mode:boolean:=false;
end record;
end rules;


with  Symbolic_Expression, Objects, Text_Io, Integer_Text_Io, Factbase,
Rules;
  use  Text_Io, Integer_Text_Io;
```

C.4 interface de ODSE: paquetage HIPLAN

```
-----------------------------------------------------------------------
-- This package contains the specifications of the plan inference
-- engine
-- description : the package operates on four rulebases:
-- PBTCBASE: contains knowledge about various ways of achieving
--           complex actions. It represent the skills of the
--           system in plan generation. (used by planhandler
--           to find ways of accomplishing a given task.see below)
--
-----------------------------------------------------------------------
package  Hiplan IS
  package  Se renames Symbolic_Expression;
  package  Ob renames Objects;
  package  Fct renames Factbase;
  package  Rul renames Rules;
  subtype  Tasktype IS Ob.Object;
-------------------------------------------------------------
--procedure :achieve
--consult the different knowledge and fact bases to achieve the
```

```
--task .
--exceptions raised: illegal_definition_of_task,
--                   task_out_of_my_skills
-------------------------------------------------------

  procedure Achieve(Task_Arg : IN Tasktype;
                    Fctbase  : IN OUT Fct.Factbase;
                    Rbase    : IN OUT Rul.Rulebase;
                    Success  : IN OUT Boolean);
-------------------------------------------------------------------
-- procedure : do_it
-- description : executes the primitive action and records the effects
--               in the specified fctbase
-- exceptions raised : primitive_task_failure,illegal_specification,
--                     is_not_a_primitive_task,
--                     primitive_task_is_not_implemented
-------------------------------------------------------------------
  procedure Doit(Primitive_Action : IN Tasktype;
                 Fctbase          : IN OUT Fct.Factbase;
                 Rbase            : IN OUT Rul.Rulebase);
-------------------------------------------------------------------
-- procedure :planhandler
-- description : monitors the implementation of the plan and casts the
--               effects in fctbase
-- uses global variables: KDGEBASE,DEMNSBASE,THMBASE
-- exceptions raised : plan_failure,illegal_specification_of_plan,
--                     is_not_a_plan
-------------------------------------------------------------------
-------------------------------------------------------------------
-- procedure :MACROHANDLER
-- description : implements the macroaction specified by a call
--               to a specific module
--exceptions raised : improper_macroaction_specification,
--                    macro_action_not_yet_implemented
-------------------------------------------------------------------
  procedure Macrohandler(Macroaction : IN Tasktype;
                         Fctbase     : IN OUT Fct.Factbase;
                         Rbase       : IN OUT Rul.Rulebase;
                         Success     : IN OUT Boolean);
-------------------------------------------------------------------
-- procedure :execution_module
-- description : call the specified procedure on the
--               specified list of arguments
-------------------------------------------------------------------
--  PROCEDURE Execution_Module(Procedure_Name : IN Routine_Index;
--                             List_Arg       : IN Tasktype);
-------------------------------------------------------------------
-- INTERACTIONS WITH THE DATA BASE
-------------------------------------------------------------------


-------------------------------------------------------
--procedure :make
--description : creates an object having the features specified
--              in the make_arg.
--             make_arg is of the form
--            (object_class object_name {(attr val)}). For efficient
--            access to the object, it is recomended to figure
--            object_class in the index of the fact base. The new
--            object created will have name object_name and type
--            object_class. the couples (attr val) represent the
--            initial set of values assigned to the object and as thus
```

```
--            are treated as augmentations to be added to the data
structure
--            of the objects.
-------------------------------------------------------------------
  procedure Make(Make_Arg : IN Se.S_Expr;
                 Fctbase  : IN OUT Fct.Factbase;
                 Rbase    : IN OUT Rul.Rulebase);

-------------------------------------------------------------
--procedure : modify
-- description : modifies the object referenced in the argument.
--               argument is of the form :
--               (object_class object_name {(attr val)}). an object
--               named object_name of class object_class should exist
--               in the fact base. all pre-existing values of the
--               attributs mentioned in {(attr val)} are erased and
--               replaced by those indicated as second components of
(attr obj)
--               pairs.
-----------------------------------------------------------
  procedure Modify(Modify_Arg : IN Se.S_Expr;
                   Fctbase    : IN OUT Fct.Factbase;
                   Rbase      : IN OUT Rul.Rulebase);
-------------------------------------------------------------------
-- procedure : augment
-- description : add the features {(attr val)} to the object obj_name
--               argument :((object_class,obj_name) {(attr val)})
-------------------------------------------------------------
  procedure Augment(Augment_Arg : IN Se.S_Expr;
                    Fctbase     : IN OUT Fct.Factbase;
                    Rbase       : IN OUT Rul.Rulebase);

-------------------------------------------------------------------
-- procedure   : retract
-- description : retracts the value val from attribute attr of the
specified
-- objectfact base.
--              argument : ((object_class,name),{(attr,val)})
-------------------------------------------------------------------
  procedure Retract(Retract_Arg : IN Se.S_Expr;
                    Fctbase     : IN OUT Fct.Factbase;
                    Rbase       : IN OUT Rul.Rulebase);
-------------------------------------------------------------------
--procedure : remove
-- description: retracts the specified object from the fact base.
--              argument : (object_class,name)
-------------------------------------------------------------
  procedure Remove(Remove_Arg : IN Se.S_Expr;
                   Fctbase    : IN OUT Fct.Factbase;
                   Rbase      : IN OUT Rul.Rulebase);

-------------------------------------------------------------------
  procedure Add_Justifier(Arg      : IN Se.S_Expr;
                          Fctbase : IN OUT Fct.Factbase);
  procedure Add_Justificand(Arg      : IN Se.S_Expr;
                            Fctbase : IN OUT Fct.Factbase);
  procedure Add_Constraint(Arg      : IN Se.S_Expr;
                           Fctbase : IN OUT Fct.Factbase;
                           Rbase   : IN OUT Rul.Rulebase);
  procedure Relax_Constraint(Arg      : IN Se.S_Expr;
                             Fctbase : IN OUT Fct.Factbase;
                             Rbase   : IN OUT Rul.Rulebase);
```

```
   procedure Set_Justificand(Arg      : IN Se.S_Expr;
                             Fctbase : IN OUT Fct.Factbase);
   procedure Set_Justifier(Arg      : IN Se.S_Expr;
                           Fctbase : IN OUT Fct.Factbase);
-------------------------------------------------------------------------
-- procedure forward_cycle
-- description : forward evaluation of rules starting with rule_arg
--               all rules activated as a result of rule_arg are fired
--               and the process recurses until no more rule is firable
-- exception raised : none
-------------------------------------------------------------------------
   procedure Forward_Cycle(Event, Lparm : IN Se.S_Expr;
                           Fctbase      : IN OUT Fct.Factbase;
                           Rbase        : IN OUT Rul.Rulebase);
   function Eval(Exp      : IN Se.S_Expr;
                 Fctbase : IN Fct.Factbase;
                 Rbase   : IN Rul.Rulebase) RETURN Se.S_Expr;
   procedure Exe_Routine(Rout     : IN Natural;
                         Larg     : IN Se.S_Expr;
                         Fctbase : IN OUT Fct.Factbase;
                         Rbase    : IN OUT Rul.Rulebase);
END Hiplan;
```

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de Messieurs

. SAGNES Georges , Professeur
. GERBER Roland , Professeur

Monsieur FONKOUA Alain Blaise

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " **Informatique** "

Fait à Grenoble, le 20 septembre 1989

Pour le Président de l'I.N.P.G.
et par délégation,
le Vice-Président
P. VENNEREAU