NUMÉRO D'ORDRE : 1524 ANNÉE : 1995

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

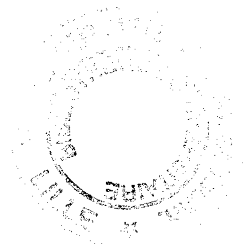
Akram Djellal BENALIA

HELPDraw : un environnement visuel pour la génération automatique de programmes à parallélisme de données

Thèse soutenue le 29 mai 1995, devant la commission d'examen :

Président :	Jean-Marc GEIB	Professeur	LIFL - USTL
Rapporteurs :	Guy-René PERRIN	Professeur	ICPS - ULPS
	François IRIGOIN	Maître de Recherche	CRI - Paris
Examinateurs :	Jean-Luc DEKEYSER	Professeur	LIFL - USTL
	Philippe MARQUET	Maître de conférences	LIFL - USTL
	Pierre MANNEBACK	Maître de conférences	Fac. Polyt. Mons- Belgique

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. (33) 20 43 47 24 Télécopie (33) 20 43 65 66



UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

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

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

PROFESSEUR EMERITE

M. A. LEBRUN

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

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

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

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

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

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

PROFESSEURS - 1ère CLASSE

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

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

PROFESSEURS - 2ème CLASSE

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

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

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

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

Remerciements

Je tiens à remercier

Jean-Marc GEIB d'avoir accepté la présidence de ce jury,

Guy-René PERRIN et François IRIGOIN d'avoir rapporté cette thèse,

Pierre MANNEBACK pour son enthousiasme et pour sa présence comme examinateur dans ce jury.

Les travaux présentés sont aussi ceux d'une équipe, je tiens particulièrement à remercier Jean-Luc DEKEYSER de m'avoir fait confiance en m'ouvrant la porte de son équipe. Sa rigueur et son esprit critique m'ont appris la remise en cause et le bénéfice qui en découle finalement. Je remercie Philippe MARQUET pour ses nombreuses suggestions et critiques. Les nombreuses discussions que nous avons eues ont grandement contribué à l'aboutissement de ce travail.

Je remercie Dominique et Cyril, mes deux compères. Cette expérience de tri-rédaction a été pour moi une motivation supplémentaire. Un merci tout particulier à Dominique qui m'a aidé dans la préparation de la soutenance, à Cyril et Boris pour avoir pris en charge la préparation du pot. L'ambiance agréable du bureau m'a permis d'évoluer dans un cadre agréable.

Je remercie les autres membres de l'équipe CSPI, thésards ou ex-thésards, et les membres du LIFL qui ont contribué au bon déroulement de mes travaux.

Enfin, je remercie ma famille qui n'a pas cessé de me soutenir même à distance tout le long de ma thèse.

À mes très chers parents

Introduction **1**

I Environnements graphiques de programmation parallèle **5**

1 Les modèles de programmation parallèle 6
1.1 Le parallélisme de contrôle 6
1.2 Le parallélisme de données 7
1.3 Le parallélisme de flux 8
2 Classification des environnements graphiques 8
2.1 Conception et implémentation 8
2.1.1 Parallélisme de tâches 10
2.1.2 Parallélisme de données 16
2.1.3 Parallélisme à flot de données 19
2.2 Déverminage de programmes 21
2.2.1 Parallélisme de tâches 22
2.2.2 Parallélismes de données 26
2.3 Transformation de codes (parallélisation et optimisation) 28
2.4 Analyse de performances 31
3 Conclusion 34

II Programmation visuelle et visualisation de programmes **37**

1 Définitions 38
2 Avantages de l'utilisation des graphiques 39
3 Langages de programmation visuelle 40
3.1 Techniques de spécification de programmes 42
3.1.1 Graphes flot de contrôle 42
3.1.2 Graphes flot de données 46
3.1.3 Langages complètement visuels 50
3.1.4 Hyperprogrammation 54
3.1.5 Programmation basée sur les exemples 59
3.1.6 Autres techniques de spécification 66
3.2 Langages de programmation visuelle dédiés au parallélisme 66
4 Systèmes de visualisation de programmes 76
4.1 Critères de classification 77
4.1.1 Portée de la visualisation 77
4.1.2 Niveau d'abstraction 79
4.1.3 Instrumentation 81
4.2 Systèmes de visualisation de programmes parallèles 83
4.2.1 Systèmes à instrumentation manuelle 85

4.2.2 Systèmes à instrumentation automatique 86
4.2.3 Systèmes à instrumentation semi-automatique 87
5 Problèmes généraux et perspectives 88
6 Conclusion 89

III L'environnement de programmation HELPDraw **91**
1 Le modèle de programmation data-parallèle géométrique HELP 93
1.1 Les hyper-espaces 93
1.2 Les objets data-parallèles 94
1.3 Niveau microscopique : les opérations calculatoires 94
1.3.1 Domaine de conformité 95
1.3.2 Domaine contraint : constructeur « on » 96
1.3.3 Domaine masqué : constructeur « where » 96
1.3.4 Association 97
1.3.5 Affectation injective 97
1.4 Niveau macroscopique : les opérations géométriques 97
1.5 Les triangles 101
1.6 Un exemple d'algorithme data-parallèle conçu selon le modèle HELP . 102
2 Développement des instructions de base 104
2.1 HELPDraw non contextuel 106
2.2 Définition d'hyper-espaces 106
2.3 Définition des objets data-parallèles 107
2.4 Réalisation des opérations géométriques 110
2.5 Application des opérations microscopiques 115
2.5.1 Opérations arithmétiques et logiques 115
2.5.2 Domaines contraints et masqués 116
2.6 Les règles de construction d'historiques 120
2.6.1 Les règles concernant les opérations microscopiques 120
2.6.2 Les règles concernant les opérations géométriques 125
2.7 Développement directe d'expressions data-parallèles 126
2.7.1 Construction d'une expression 126
2.7.2 Les différents composants d'une expression 128
2.7.3 Utilisation d'une expression 129
2.8 Manipulation des scalaires 130
2.9 Intégration du code dans un programme 131
3 Développement de blocs d'instructions 132
3.1 L'éditeur de blocs d'instructions 132
3.1.1 Le constructeur de domaine contraint par bloc « on » 132
3.1.2 La construction conditionnelle « If_Then_Else » 133
3.1.3 Le constructeur de domaine masqué « Where_Elsewhere » . . 135
3.1.4 La construction « While » 135
3.1.5 La construction « Repeat » 135
3.2 Génération de code pour un bloc d'instructions 136
4 Conclusion 137

IV Génération automatique de code C-HELP **139**
1 Description du langage data-parallèle C-HELP 140
1.1 Déclaration des objets du modèle 140
1.1.1 Les hyper-espaces 140
1.1.2 Les objets data-parallèles 141
1.2 Les opérations microscopiques 143
1.3 Les opérations macroscopiques 143
2 Génération automatique de code 146
2.1 Déclaration des objets du modèle 146
2.1.1 Les hyper-espaces 146
2.1.2 Les objets data-parallèles 147
2.2 Application des opérations micro et macroscopiques 148
2.2.1 Les opérations microscopiques 149
2.2.2 Les opérations macroscopiques 151
3 Exemple de génération de programme 157
3.1 Conception géométrique 158
3.2 L'algorithme géométrique 160
3.3 Le code C-HELP généré automatiquement 161
4 Conclusion 161

V Génération automatique de code HPF **163**
1 Description du langage HPF 164
1.1 Les tableaux 166
1.2 La directive « TEMPLATE » 166
1.2.1 Syntaxe 166
1.2.2 Exemples 167
1.3 Les directives d'alignement 167
1.3.1 Syntaxe 168
1.3.2 Exemples d'alignement par rapport à des templates 168
1.4 La directive « DYNAMIC » 169
1.5 La directive « PROCESSORS » 169
1.5.1 Syntaxe 169
1.5.2 Exemples 169
1.6 Les directives de distribution 170
1.7 L'instruction FORALL 172
2 Traduction de la sémantique du modèle géométrique HELP en HPF 173
2.1 La première alternative écartée 173
2.2 Le principe du passage du point à l'indice 174
2.2.1 Représentation d'un objet temporaire 174
2.2.2 Traduction d'une expression data-parallèle 176
2.3 Déclarations des éléments du modèle géométrique 177
2.3.1 Les objets rectangulaires 177
2.3.2 Les objets non rectangulaires 177
2.3.3 Les DPO dynamiques 178
2.4 Le niveau macroscopique 179
2.4.1 Les étapes du calcul de la fonction de description f 180

2.4.2 Les outils de maintien des informations nécessaires au calcul de f . . . 182
2.4.3 Calcul de f pour chaque opération macroscopique . . . 184
2.4.3.1 Déplacements . . . 184
2.4.3.2 Rotations . . . 185
2.4.3.3 Réplications . . . 188
2.4.3.4 Décalages . . . 190
2.4.3.5 Extractions . . . 191
2.4.3.6 Réductions . . . 194
2.4.4 Formule générale de la fonction de description f . . . 195
2.4.5 Exemple de génération automatique d'une expression data-parallèle . . . 197
2.4.6 Discussion . . . 202
2.5 Le niveau microscopique . . . 203
2.5.1 Application d'opérations géométriques à une sous-expression microscopique . . . 203
2.5.2 Les constructions data-parallèles . . . 204
2.5.3 Les règles du niveau microscopique . . . 207
3 Exemple de génération de programmes : Inversion de matrice . . . 209
4 Conclusion . . . 212

VI Illustration des différentes étapes de développement d'un programme sous HELPDraw **215**
1 Conception géométrique . . . 215
2 Réalisation sous HELPDraw . . . 218
3 Génération de codes . . . 224
3.1 Le code C-HELP . . . 224
3.2 Le code HPF . . . 224
4 Résumé . . . 225

Conclusion **227**
1 Résumé des travaux . . . 227
2 Perspectives . . . 228
2.1 Un langage de programmation visuelle data-parallèle . . . 228
2.1.1 Définition du squelette du programme . . . 230
2.1.1.1 La construction de séquence « Seq » . . . 231
2.1.1.2 Les autres constructions de contrôles . . . 231
2.1.1.3 La structure d'une « Fonction » . . . 231
2.1.2 Exemple de programme . . . 232
2.2 Analyse et visualisation géométrique d'un programme data-parallèle . . . 235

Bibliographie **237**

Table des figures **254**

Introduction

Devant les besoins de puissance de plus en plus élevés pour résoudre certains problèmes (mécanique des fluides, météorologie, etc.), le parallélisme apparaît comme solution réaliste et prometteuse. Il occupe de ce fait une place de plus en plus importante dans le domaine du développement actuel du calcul scientifique.

Les informaticiens sont aujourd'hui confrontés au problème de conception de langages et d'environnements permettant la programmation aisée et efficace des machines parallèles. Les concepteurs des langages sont influencés par deux critères qui sont généralement contradictoires : soit le langage est proche du programmeur, soit il est proche de la machine. Le premier type de langage s'oriente vers une abstraction de la machine (virtualisation) tout en observant une éventuelle perte d'efficacité (exemple de tels langages : Fortran). Le second type permet une plus grande efficacité au niveau de l'exploitation de la machine, mais nécessite d'une part une bonne connaissance de l'architecture (plus difficile à maîtriser), et diminue d'autre part la portabilité du programme (exemple de tels langages : MPL de MasPar).

Conscients de cet antagonisme, les informaticiens continuent toujours à chercher des moyens permettant de rendre les machines parallèles les plus accessibles possibles. Nous, nous abordons ce problème en prenant en considération le point suivant :

> Comment programmer dans ce modèle sans être contraint ni par la syntaxe d'un langage particulier ni par l'architecture de la machine cible ? Ceci bien évidemment en gardant le maximum de performances.

Pour y répondre, nous nous orientons vers l'étude d'une *couche visuelle* au-dessus des langages qui permettrait de traduire automatiquement l'algorithme data-parallèle définit par un support visuel en un code source data-parallèle. Comme ce n'est pas l'utilisateur qui va lui-même écrire le programme source, il ne fait qu'interagir avec la couche visuelle, peu importe alors si le langage cible est très proche de la machine ou offre une syntaxe complexe.

Cette couche visuelle se rapporte à un domaine relativement récent qui est : « *la programmation visuelle* ». Dans cette thèse nous montrerons la faisabilité d'une telle couche, à travers

la conception et la réalisation de notre environnement visuel HELPDraw. Ce dernier permet, à partir de simples interactions avec l'utilisateur, de produire des codes data-parallèles compilables : C-HELP (un langage data-parallèle mis en œuvre dans notre équipe) ou HPF (High Performance Fortran) le nouveau Fortran data-parallèle.

La communauté informatique déploie constamment des efforts dans le but d'offrir des environnements permettant d'assister le mieux possible les programmeurs. L'apparition des écrans graphiques a considérablement contribué à l'émergence d'environnements de plus en plus conviviaux pour les utilisateurs. Les environnements graphiques sont d'un apport encore plus important lorsqu'ils sont dédiés aux utilisateurs des machines parallèles. C'est dans ce cadre que nous avons consacré le premier chapitre à la description de plusieurs environnements graphiques. Nous les avons classés selon des critères précis. Ils sont dédiés au parallélisme de tâches, de données, ou à flot de données. Ils interviennent à différentes phases du cycle de développement d'un programme: au niveau de l'implémentation, du déverminage, de l'analyse de performances, et/ou de la transformation de codes (parallélisation et optimisation).

Le domaine des langages visuels est relativement récent, nous avons préféré définir, au second chapitre, la programmation visuelle, la visualisation de programmes, et les différents termes afférents. Nous montrerons ensuite l'intérêt de l'utilisation des graphiques. Puis nous décrirons en détails les différentes méthodes de programmation visuelle ; nous les appelons techniques de spécification. Nous les illustrerons par une étude de plusieurs langages de programmation visuelle pertinents, notamment ceux dédiés au parallélisme. Ces systèmes sont décrits selon une classification que nous proposons.

La visualisation de programmes est la partie duale de la programmation visuelle. Elle permet de rejouer un programme selon plusieurs formes graphiques. Le programme peut être montré à plusieurs niveaux. Nous expliquerons les différents moyens employés pour la mise en œuvre des systèmes de visualisation. Nous décrirons quelques systèmes en montrant comment ils entrent dans la classification que nous proposons pour les systèmes de visualisation.

Les deux premiers chapitres présentent sur deux aspects différents un grand nombre d'environnements graphiques dédiés au parallélisme. Ceci montre bien l'intérêt de ce genre d'outils dans la communauté scientifique. L'étude de ces chapitres, nous permettra en outre de mettre en évidence quelques points importants que nous retiendrons pour la mise en œuvre de HELPDraw.

Le troisième chapitre décrit l'environnement visuel HELPDraw. Ce dernier utilise le modèle géométrique HELP (Hyper-Espace et Langages Parallèles) comme support visuel pour la programmation data parallèle. À la suite d'une brève description de ce modèle, nous montrerons les fonctionnalités de chaque module de l'environnement HELPDraw. Ceci nous permettra

de voir en particulier comment l'utilisateur peut exprimer son algorithme, conçu selon le modèle HELP, par des manipulations directes et interactives de l'environnement graphique. HELPDraw transforme ces manipulations en un code source ; il produit actuellement des instructions ou des blocs d'instructions. Le programmeur peut choisir de générer un code C-HELP ou HPF.

Les deux chapitres suivants IV et V sont consacrés respectivement à la génération automatique de codes C-HELP et HPF. Ils expliquent les mécanismes mis en œuvre par HELPDraw pour la production automatique du code. En ce qui concerne C-HELP, le principe est assez simple car ce langage est l'implémentation directe du modèle géométrique HELP dans une syntaxe C.

L'approche HPF est plus complexe. Dans le chapitre V, nous exposerons différentes manières d'aborder la génération automatique. Dans la solution que nous avons retenue, HELPDraw doit en particulier assurer le passage de la notion de points et de référentiel (sous-jacente à notre modèle géométrique HELP) à la notion d'indices de HPF. Les mécanismes employés dans les deux chapitres IV et V seront illustrés à la fin de chaque chapitre par un exemple d'algorithme data-parallèle.

Pour mieux montrer le fonctionnement de HELPDraw, le dernier chapitre retrace, à travers le développement d'un exemple de programme, toutes les étapes que l'utilisateur de HELPDraw doit respecter : de la conception géométrique de l'algorithme jusqu'à la génération de code.

Nous résumerons enfin les travaux qui ont amené ce document à terme puis nous ouvririons quelques perspectives orientées soit vers la génération de programmes complets (l'environnement HELPDraw actuel ne génère que des instructions ou des blocs d'instructions), soit vers une partie complètement duale à HELPDraw : un système de visualisation d'algorithmes data-parallèles.

Chapitre I

Environnements graphiques de programmation parallèle

L'apparition des terminaux graphiques nous ont permis d'accepter les systèmes multi-fenêtres comme standard en matière d'interface. Ces systèmes font abstraction de beaucoup de problèmes aussi bien au niveau de l'écriture de programmes qu'au niveau de la compréhension de leur exécution. Avant la mise en œuvre de ce type de systèmes, les utilisateurs étaient contraints de manipuler des shells interactifs afin de développer leurs programmes.

La nécessité des systèmes multi-fenêtres est certes justifiée au niveau de la programmation séquentielle, mais elle l'est encore plus en ce qui concerne la programmation parallèle. Ceci est dû à plusieurs raisons, en particulier le besoin d'analyser des programmes parallèles qui sont rarement déterministes, d'écrire des programmes gérant beaucoup de ressources (de synchronisation, de communication, de traitement distribué, etc.), d'assurer la portabilité des programmes en adéquation avec la grande émergence d'architectures parallèles actuelles.

De par l'importance de ces systèmes, nous avons consacré ce premier chapitre à l'étude de plusieurs environnements graphiques de développement de programmes parallèles. Pour mieux montrer ce qui se fait autour de nos travaux et ce qui a été exploité pour les différents modèles de parallélisme, nous proposons une classification de ces environnements. Nous présentons d'abord les différents modèles de parallélisme. Nous indiquons ensuite les critères que nous avons choisis pour la classification. Nous décrirons enfin plusieurs systèmes et environnements représentatifs en montrant en particulier l'apport de chacun et son interactivité avec l'utilisateur.

1 Les modèles de programmation parallèle

Le parallélisme peut être exploité de manières différentes. Nous distinguons principalement trois types de parallélisme :

- le parallélisme de contrôle ;
- le parallélisme de données ;
- le parallélisme de flux.

1.1 Le parallélisme de contrôle

L'exploitation du parallélisme de contrôle provient du fait qu'une application peut être décomposée en plusieurs actions pouvant être exécutées de manière concurrente. Ces actions sont appelées aussi tâches, c'est pourquoi ce type de parallélisme est également nommé « parallélisme de tâches ».

La décomposition d'une application est liée à l'architecture ; plus la décomposition est fine, plus le nombre de ressources de calcul requises augmente. Lorsque ce nombre est limité, nous sommes amenés à gérer le placement et l'ordonnancement des actions.

Le gain idéal en temps d'exécution serait d'exécuter les actions indépendamment les unes des autres, chacune sur un processeur. Mais dans une application réelle les actions présentent presque toujours des dépendances (cf. figure I.1). Une dépendance intervient lorsqu'une action doit être terminée pour qu'une autre commence. Nous distinguons deux cas de dépendance [GRS91] :

1. Dépendance de contrôle de séquence : elle correspond au séquencement dans un algorithme classique. Elle nécessite souvent la synchronisation des actions.
2. Dépendance de contrôle de communication : elle se crée lorsqu'une action envoie des informations à une autre action.

Ces dépendances imposent des restrictions sévères sur l'association des ressources de calcul aux actions : il arrive que des actions soient en attente d'autres actions alors que certaines ressources restent inoccupées. L'exploitation du parallélisme de contrôle consiste à gérer les dépendances entre les actions d'une application pour obtenir une allocation de ressources de calcul aussi efficace que possible. Les langages à parallélisme de contrôle doivent permettre

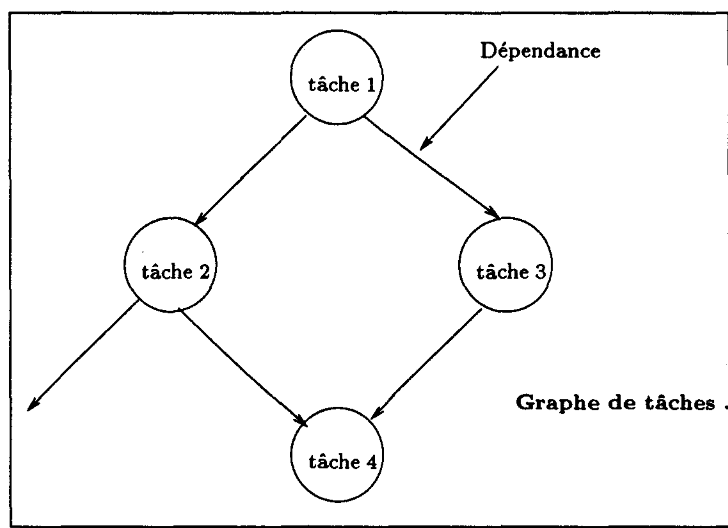


FIG. I.1 - *Dépendance de contrôle*

d'exprimer ces dépendances et leur gestion, en offrant des constructions et des outils de synchronisation, de communication, de placement, etc. L'utilisation de ces outils est souvent une tâche délicate, surtout pour de non-informaticiens, que ce soit au niveau de l'implémentation des programmes ou de leur analyse. C'est pour cette raison que les environnements graphiques doivent rendre transparente ou tout au moins plus confortable l'utilisation de ces outils.

1.2 Le parallélisme de données

Le parallélisme de données provient du fait que certaines applications agissent sur des structures de données homogènes (vecteurs, matrices, etc.) en répétant une même action sur chaque élément de la structure.

Les problèmes que nous pouvons rencontrer dans ce modèle de programmation concernent en général le partitionnement, la distribution et l'alignement des structures de données. Le découpage de ces structures et leur distribution sur l'ensemble des processeurs dépendent de l'application et de la topologie de la machine cible (SIMD, MIMD ou hétérogène). Pour réduire le nombre de communications, le programmeur doit savoir aligner les structures les unes par rapport aux autres.

La mise en œuvre d'environnements graphiques doit permettre de réduire ou faire abstraction de ces problèmes, que ce soit au niveau de la conception, de l'implémentation, ou du déverminage.

1.3 Le parallélisme de flux

Le parallélisme de flux vient du fait que certaines applications fonctionnent selon le mode du travail à la chaîne : on dispose d'un flux de données, généralement similaires, sur lesquelles nous devons effectuer une suite d'opérations en cascade [GRS91].

Ce modèle de parallélisme, appelé aussi « parallélisme à flot de données », est dual au parallélisme de contrôle. Dans le premier, le traitement est dirigé par la disponibilité des données au lieu de la disponibilité des instructions [ERL92]. Une autre dualité existe cette fois entre le parallélisme à flot de données et le parallélisme de données. La différence réside dans le placement spatio-temporel des données par rapport aux actions [GRS91].

Dans ce modèle, les environnements graphiques peuvent intervenir au niveau de l'édition et la visualisation des graphes flot de données, faisant d'une part abstraction de l'implémentation et permettant d'autre part l'analyse des programmes.

2 Classification des environnements graphiques

Ce chapitre examine une trentaine de systèmes que nous avons classifiés selon deux critères. Le premier précise la *phase de développement* où le système intervient : (1) Conception et implémentation de programmes, (2) Déverminage de programmes (3) Analyse de performances, et/ou (4) Transformation de codes (parallélisation et optimisation). Le deuxième critère spécifie le type de parallélisme auquel est dédié le système : (*i*) Parallélisme de tâches, (*ii*) Parallélisme de données, ou (*iii*) Parallélisme à flot de données. Il est clair que ces classes ne sont pas toujours disjointes. Ainsi, certains systèmes peuvent se trouver dans plusieurs classes, par exemple au niveau de la conception puis au niveau du déverminage. Le tableau I.1 résume les caractéristiques des systèmes étudiés.

2.1 Conception et implémentation

L'évolution des langages de programmation a été guidée en grande partie par la volonté de rendre la programmation plus facile et suscitant le moins d'erreurs possibles. La progression des langages machines à l'assembleur, puis aux langages de haut niveau et maintenant aux langages de programmation visuelle, montre le travail qui se fait pour essayer de retrouver des représentations de plus en plus naturelles des problèmes à résoudre. De ces représentations ou abstractions résulte une indépendance par rapport aux architectures cibles, ce qui évite une éventuelle gestion de ressources systèmes, offre une facilité de programmation, et augmente la productivité des programmeurs.

TAB. I.1 - *Classification des systèmes de développement de programmes parallèles*

Systèmes	Conception et implémentation	Déverminage	Analyse de performances	Transformation[a]
Parallélisme de tâches				
αTrellis		★		
ChaosMON			★	
G++	★			
HeNCE	★	★		
MIN-Graph			★	
Parade	★	★	★	
PF-View		★		
Poker	★			
PPSE	★			
Projections			★	
Schedule	★	★	★	
TIPS	★		★	
Visputer	★	★		
Parallélisme de données				
ALEX	★			
Faust		★	★	★
Forge 90				★
IVE		★		
LIVE		★		
MPPE		★	★	
Parafrase-2				★
ParaScope				★
PAT				★
PAWS			★	★
PRISM		★	★	
TOP/DOMDEC	★			
VISTA			★	
VISUAL.PEI	★			
VOSpeL				★
Flot de données				
GRAPE	★			
NL	★			
TDFL	★			

[a] Transformation = parallélisation et/ou optimisation de code

L'évolution d'architectures radicalement différentes pour le traitement parallèle a de nouveau incité le programmeur à gérer les ressources matérielles ; des ressources qui peuvent aller jusqu'à des milliers de processeurs avec des interactions complexes et imprévisibles. Ceci peut être vu comme un échec à offrir de réelles abstractions dans le domaine de la programmation parallèle, obligeant par conséquent les programmeurs à se préoccuper de détails d'implémentations non nécessaires. Le besoin d'abstractions adéquates devient ainsi plus important pour la programmation parallèle que ce qui l'est en ce qui concerne la programmation séquentielle.

Plusieurs environnements graphiques ont été mis en œuvre dans cet esprit pour assister le programmeur au niveau de la conception et de l'implémentation. Nous en retrouvons plusieurs autour du parallélisme de tâches, mais beaucoup moins au niveau du flot de données et très peu en ce qui concerne le parallélisme de données.

2.1.1 Parallélisme de tâches

Au niveau du parallélisme de tâches, nous avons choisi de présenter les outils tels que HeNCE qui assure l'exploitation des multiples ressources de calcul d'un réseau de machines hétérogènes; Schedule permet d'implémenter des programmes Fortran portables ; G++ permet de développer des applications temps-réel ; Parade et Visputer offrent la possibilité de concevoir et d'éditer graphiquement des programmes à passage de message, et de configurer des processus parallèles sur un réseau multi-processeurs ; d'autres sont semblables à Visputer et Parade, par exemple Poker, PPSE et TIPS.

HeNCE (Heterogenous Network Computing Environment) [BDG+91] est une couche au-dessus de PVM assistant les scientifiques dans le développement de programmes parallèles devant s'exécuter sur un réseau d'ordinateurs. L'utilisateur spécifie explicitement le parallélisme par l'intermédiaire d'une interface graphique permettant entre autres l'édition de graphes HeNCE et la configuration des machines virtuelles. Il construit des graphes exprimant les dépendances et le flot de contrôle (graphe flot de tâches). Chaque nœud représente une sous-routine (écrite en C ou en Fortran). HeNCE offre en particulier quatre types de constructions de contrôle de flot : les boucles, les fans, les pipes, et les conditions. Un avantage essentiel de HeNCE est la possibilité de spécifier, pour un nœud donné, plusieurs implémentations ; à chaque implémentation correspond une machine. HeNCE peut prendre en compte dynamiquement (au cours de l'exécution) la machine qui doit exécuter un nœud donné. Un nœud de la machine virtuelle (l'ensemble des machines définies par l'utilisateur) peut être une station de travail comme il peut être une CM-2 à 64k processeurs. Les concepteurs de HeNCE prétendent que cette combinaison de spécification de graphes et la définition de multiples implémentations favorisent le passage des approches SPMD (Single Program Multiple Data) courantes vers des systèmes de programmation massivement parallèles. HeNCE permet

également la visualisation de programmes, cet aspect sera décrit plus loin avec les systèmes de déverminage (sous-section 2.2).

Schedule Jack Dongarra et al. [DSB92] ont développé un environnement graphique appelé Schedule permettant d'assister le programmeur dans deux phases de développement. Le module « Build » assiste le programmeur dans l'implémentation de programmes, tandis que le module « Trace facility » permet l'analyse de programmes à travers une visualisation animée du flot d'exécution (ce module sera traité plus loin — sous-section 2.2.1).

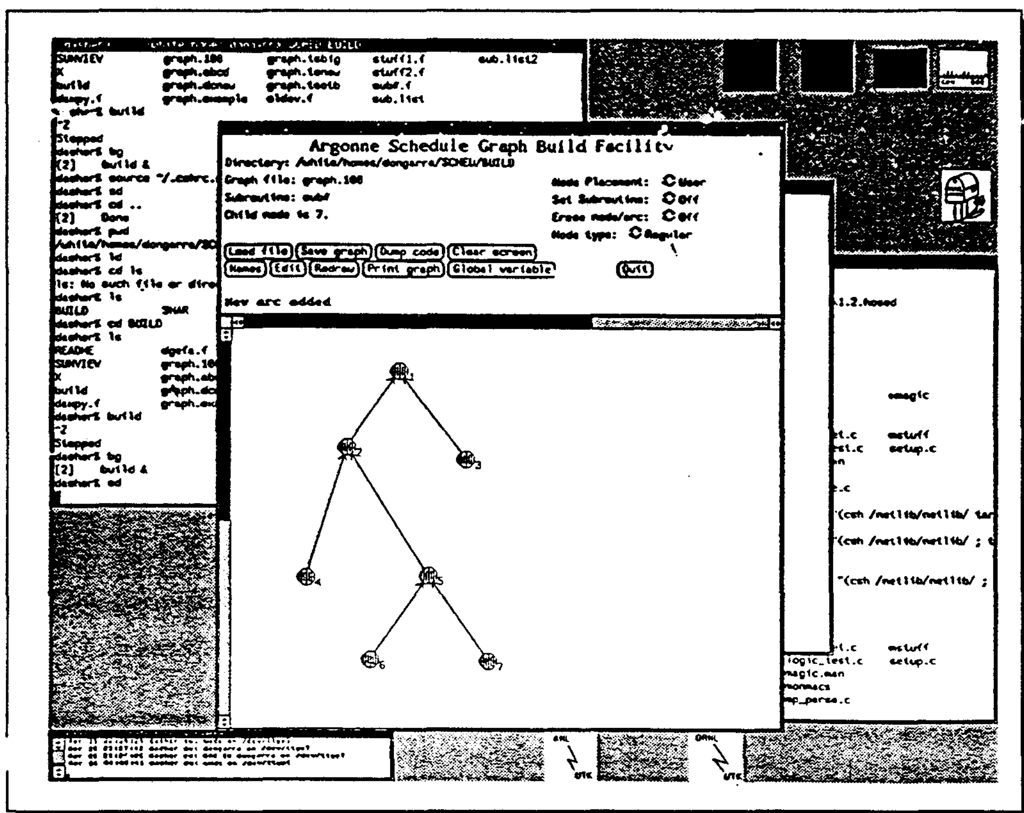


FIG. I.2 - *Schedule : un exemple de construction de programme sous Build*

Build génère des programmes Fortran portables sur plusieurs architectures : Vax 11/780, Alliant FX/8, Sequent Balance 21000, Encore Multimax, Cray-2, Cray X-MP, Cray Y-MP, IBM 3090, et Flex 32. L'utilisateur dessine interactivement le graphe de dépendances de son programme, puis spécifie pour chaque nœud les sous-routines correspondantes (écrites en Fortran séquentiel). Le système génère ensuite le code avec des appels à la bibliothèque Schedule. C'est justement cette bibliothèque qui assure la portabilité. La figure I.2 montre un exemple de construction de programme sous Build.

Parade L'environnement Parade (PARallel And Distributed Environment) offre des outils visuels agissant à plusieurs niveaux du développement de programmes à parallélisme de tâches : conception, placement de tâches, et analyse de programmes [BPP+93]. Il s'adresse aux architectures à mémoire distribuée. Un prototype tourne sur un réseau de Transputers ou de stations UNIX.

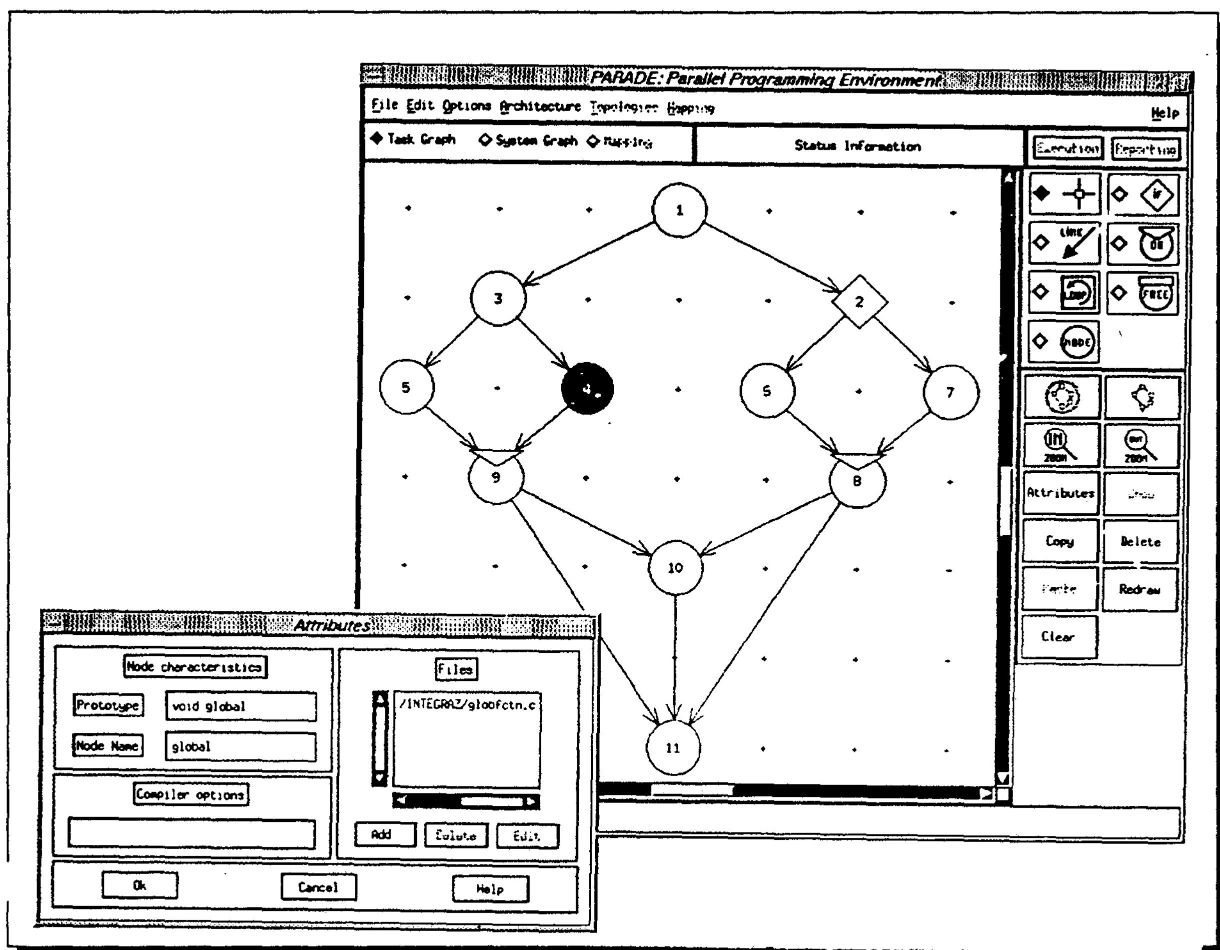


FIG. I.3 - *L'interface graphique de Parade : la fenêtre principale et une fenêtre de définition d'attributs pour le nœud 4*

L'utilisateur dessine son graphe de tâches dans un éditeur graphique selon un modèle inspiré de la programmation à flot de données. Deux types de nœuds peuvent se trouver dans un graphe : des nœuds procéduraux et des nœuds de contrôle. Les premiers correspondent au code séquentiel des tâches (des fonctions C ou des procédures Pascal). L'utilisateur précise pour tout nœud, s'il est en réception des données provenant (1) de tous les arcs en entrée (un nœud ET) ou (2) d'au moins un arc (un nœud OU). Dans un nœud de contrôle, il spécifie une condition qui sera évaluée lors de l'exécution du programme permettant ainsi d'orienter le flot de contrôle (branchement conditionnel). La figure I.3 montre l'interface graphique de Parade : dans cet exemple, il y a un seul nœud de contrôle (nœud 2) et dix procéduraux dont

deux nœuds OU' (8 et 9) et le reste de nœuds 'ET'.

Parade offre un autre éditeur graphique permettant de définir la topologie de la machine cible. L'utilisateur dessine la topologie ou en sélectionne une parmi les topologies standards pré-définies par le système, tel qu'un hypercube, tableau linéaire, anneau, etc. Il peut ensuite faire le placement des tâches de manière interactive ou utiliser le placement automatique offert par Parade. Ce dernier placement ne prend en compte que les caractéristiques statiques des graphes de tâches et de processeurs. À partir de ce placement, Parade génère automatiquement les instructions de synchronisation et de communication. Il construit un noyau directement au-dessus du système d'exploitation natif en utilisant des mécanismes de communications de bas niveau tels que les « sockets ».

G++ Le langage graphique G++ permet de spécifier des applications temps-réel parallèles [BLDA93]. À l'inverse de la plupart des autres environnements, G++ utilise une représentation graphique très proche des diagrammes d'états ou des Grafcets. Ceci est certainement influencé par la nature même des applications temps-réel où généralement un traitement se déclenche à l'arrivée d'un signal. En G++, il existe deux façons différentes pour organiser hiérarchiquement les états: une décomposition parallèle (état 'ET') et une décomposition séquentielle (état 'OU'). La décomposition parallèle correspond à la division d'un état en plusieurs composants agissant en même temps. Alors que la décomposition séquentielle divise l'état en sous-états s'exécutant l'un après l'autre. G++ utilise un éditeur graphique dirigé par la syntaxe « Gédraw ». La décomposition parallèle est exprimée en découpant verticalement le nœud. G++ offre également une syntaxe pour exprimer la synchronisation et la communication. La figure I.4 montre un exemple de programme où il y a deux composants parallèles.

Visputer Le langage Visputer, développé par Marwaha et Zhang [MZ94], offre la possibilité de concevoir et d'éditer des programmes Occam, et de configurer des processus Occam sur un réseau de Transputers. Le configurateur de Visputer fournit une interface graphique pour configurer n'importe quelle topologie de réseau de Transputers. Visputer utilise une notation graphique simple pour permettre de dessiner le diagramme d'un programme Occam. Un programme est représenté comme un graphe constitué de nœuds et d'arcs et est construit interactivement par le programmeur. Cette notation consiste en quatre objets visuels représentant les processus, les constructions (SEQ et PAR), les réplicateurs, et les liens. L'éditeur graphique est un outil de conception et de développement hiérarchique utilisant une représentation graphique bi-dimensionnelle ; un nœud de haut niveau peut être défini plus loin comme un sous-graphe.

Ayant construit un programme Occam et décidé comment le partitionner en un réseau

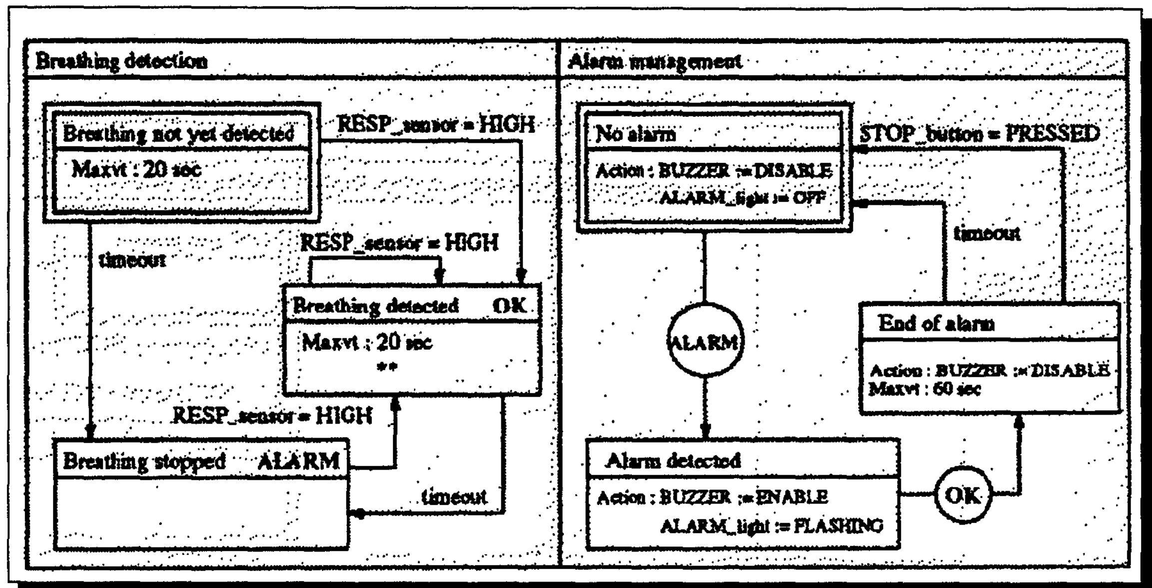


FIG. I.4 - *G++: un exemple de programme temps-réel*

de processus, l'utilisateur a besoin ensuite d'écrire un programme de configuration. Il y a trois étapes pour écrire ce programme: spécifier le réseau matériel, spécifier le réseau logiciel, puis placer le réseau logiciel sur le réseau matériel. Visputer supporte la configuration visuelle interactive et réalise la génération automatique du programme de configuration (textuel). Dès que l'utilisateur spécifie graphiquement les deux réseaux, le programme de configuration est automatiquement généré et est prêt pour la compilation. La figure I.5 montre un exemple de configuration créé en utilisant Visputer.

Autres Ils existe d'autres environnements semblables à Parade et Visputer. Le « vieux » Poker [Sny84], le premier environnement de programmation parallèle, assure le placement automatique des tâches sur la machine CHiP (Configurable Highly Parallel computer). PPSE (Parallel Programming Support Environment) réalise le placement à partir des deux graphes de tâches et de processeurs, puis génère un code C-Linda [LR90]. TIPS (Transputer-based Interactive Parallelizing System) dédié aux machines à base de Transputers. Le programmeur peut orienter le placement automatique en pondérant les arcs du graphe de tâches. TIPS offre également des outils d'analyse de performances [WCG+91].

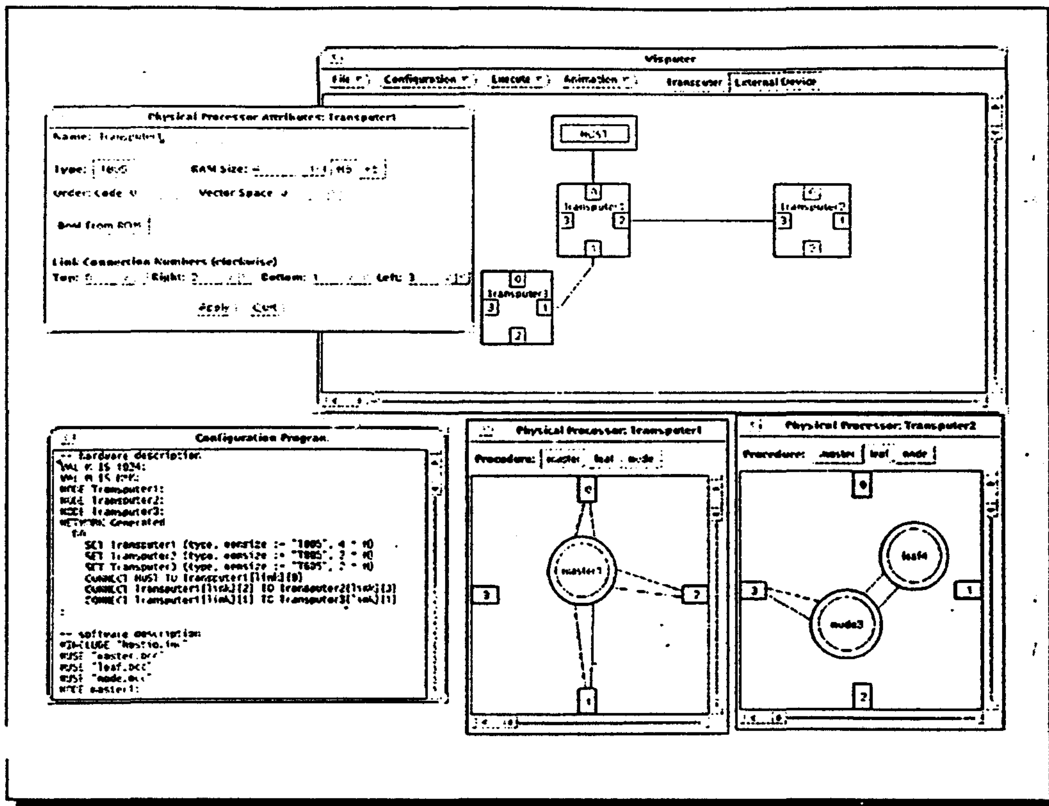
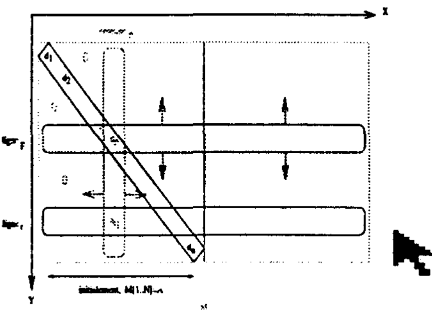


FIG. I.5 - *Une configuration créée en utilisant Visputer*



Vers un data-parallélisme visuel

Deux choses à retenir[1] :

1. La construction interactive du graphe de dépendances d'un programme est intéressante (*voir* Schedule ou Parade). On peut s'en inspirer dans deux situations :

   – pour la définition des différents modules d'un programme data-parallèle ;

   – également à un niveau plus bas, pour spécifier les différentes constructions de

1. Nous nous arrêtons à plusieurs endroits des deux premiers chapitres afin de mettre en évidence les points pouvant nous conduire à un parallélisme de données visuel. Ces points sont repérés par le symbole : « ***Vers un data-parallélisme visuel*** »

contrôle d'un programme.

2. Nous retenons aussi la spécification interactive de la topologie cible, ainsi que le placement des différents modules sur une machine distribuée (*voir* Parade ou Visputer). C'est encore plus intéressant lorsque il s'agit d'un machine hétérogène, vu qu'il peut y avoir beaucoup de paramètres à prendre en considération.

2.1.2 Parallélisme de données

Le parallélisme de tâches est un parallélisme à gros grain. Les environnements pouvant être réalisés autour de ce parallélisme, choisissent la manipulation de tâches comme niveau d'abstraction. Il est relativement facile d'avoir des représentations graphiques à ce niveau car, aussi bien les interactions même des tâches que les architectures cibles qui sont souvent des réseaux, peuvent être représentées par des graphes. Or en parallélisme de données qui est un parallélisme à grain fin, la programmation doit se concentrer sur la manipulation des données qui sont souvent d'un nombre important. Il est de ce fait plus difficile de mettre en œuvre des environnements graphiques permettant d'offrir des abstractions assistant le programmeur au niveau de l'implémentation de programmes. C'est pourquoi d'ailleurs (et malheureusement) le parallélisme de données est dans ce contexte peu exploré.

Nous citons ici TOP/DOMDEC développé à l'université du Colorado; il assure le partionnement et la distribution des structures de données. Il offre des outils de partitionnement, mais rien au niveau de l'implémentation de programmes. Le second système est ALEX spécialisé dans les manipulations de matrices. Et enfin VISUAL.PEI dédié à la résolution du système d'équations linéaires récurrentes

TOP/DOMDEC Les structures de données irrégulières sont utilisées dans plusieurs applications scientifiques, telles que les méthodes des éléments ou volumes finis. À cause de la masse importante des données, ces grilles sont généralement partitionnées puis traitées sur des machines parallèles à mémoire distribuée. Afin d'avoir un traitement efficace, le partitionnement doit maximiser l'équilibre de charge et minimiser les communications inter-processeurs. Charbel Farhat *et al.* ont développé TOP/DOMDEC (DOMain DEComposition) un environnement graphique et interactif pour le partitionnement des grilles de données et leur traitement en parallèle [FS93].

Il existe plusieurs algorithmes de partitionnement, mais il est très difficile de juger qu'un algorithme spécifique est le meilleur pour tel ou tel partitionnement. Ceci dépend du problème, de la grille de données elle même ainsi que de la machine cible. De ce fait, TOP/DOMDEC permet à l'utilisateur de choisir l'algorithme à appliquer. Il intègre huit algorithmes fréquemment utilisés dans le partionnement, tels que : l'algorithme glouton, l'algorithme de Cuthill-McKee,

ou l'algorithme de partionnement spectrale récursif. L'utilisateur peut inclure d'autres algorithmes de partitionnement. Il bénéficie des caractéristiques interactives de TOP/DOMDEC incluant l'évaluation de l'équilibrage de charge, les coûts de communication, et la génération des structures de données parallèles. Afin de tester plusieurs algorithmes de partitionnement, TOP/DOMDEC inclut aussi des outils de simulation permettant de les évaluer pour un problème spécifique utilisant une architecture particulière. Cette évaluation est faite selon des critères définis dans TOP/DOMDEC. Après cette évaluation, l'utilisateur peut appliquer d'autres algorithmes pour optimiser le partionnement.

TOP/DOMDEC est un produit commercialisé, il s'exécute sur les stations de travail Silicon Graphics et IBM RS6000 et il génère des partitionnements pour plusieurs architectures : CRAY Y-MP, KSR, IPSC-860, et CM-2.

ALEX Le système ALEX [KTC+90] intervient au niveau du développement des algorithmes numériques matriciels. Le programmeur manipule des objets graphiques qui représentent des matrices. ALEX est en fait plus qu'un simple environnement graphique assistant le programmeur dans le développement, il permet de libérer le programmeur de toute syntaxe textuelle. ALEX est un *langage de programmation visuelle*, c'est pourquoi nous reviendrons plus en détails à ce système dans le second chapitre. Nous le décrirons avec les autres langages de programmation visuelle (*cf.* chapitre II- 3.2).

Visual.Pei Sachant que la modélisation mathématique des problèmes peut se faire par l'intermédiaire d'équations linéaires récurrentes [Mon94], certains projets ont proposé des environnements qui font de cette modélisation un modèle de programmation.

La programmation consiste dans ce cadre à transcrire un problème sous formes d'équations mathématiques qui font apparaître une récurrence et de quelques équations permettant de mettre fin à la récurrence. On doit aboutir à une équation exprimée en fonction du temps. Cette équation donne l'état du système à l'instant $t+1$ en fonction de son état à l'instant t :

$$f(\ldots, t+1) \;=\; f(\ldots, t)$$

Le programme final consiste en la traduction de ces équations récurrentes ainsi que l'ensemble des transformations dans un langage cible.

Les environnements de programmation consistent à offrir des moyens permettant de guider le programmeur dans l'ensemble des transformations à entreprendre. Certains vont même plus loin, et offrent des environnement graphiques conviviaux, comme VISUAL.PEI [VP93, PVG94].

VISUAL.PEI est une couche graphique au-dessus de PEI (Parallel Equation Interpretor).

PEI offre un cadre formel pour la description des spécifications d'un problème. Il est construit autour des concepts du parallélisme de données :

- les données (ou variables) sont ici appelées champs de données, c'est l'équivalent des variables parallèles dans un langage à parallélisme de données ;
- les alignements correspondent à des changements de base en PEI (exemple : placer une matrice A sur une machine virtuelle cubique)
- les communications globales et les diffusions (broadcast) correspondent à des opérations géométriques (exemple : les réplications)
- les opérations de réduction qu'on trouve dans tous les langages data-parallèles ;
- les opérations calculatoires globales sont définies comme des opérations fonctionnelles appliquées aux champs de données (exemple : $A + B$).

Ces concepts doivent permettre par ailleurs une transcription des différentes manipulations dans un langage data-parallèle (exemple : HPF).

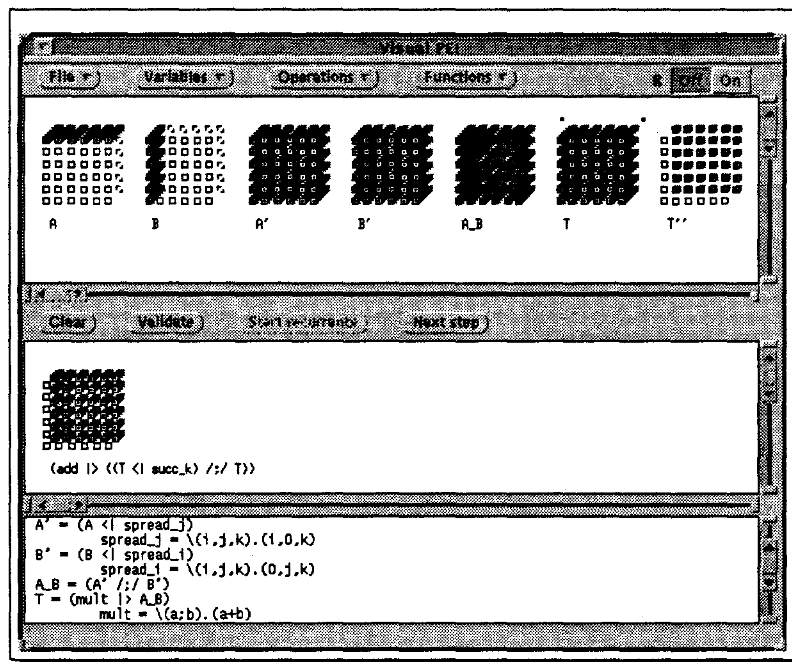


FIG. I.6 - *L'interface graphique* VISUAL.PEI *: exemple de multiplication de matrices*

L'interface graphique VISUAL.PEI permet l'application de ces opérations de manière graphique et interactive. La figure I.6 montre un exemple de multiplication de matrices ($A \times B$).

La fenêtre en haut visualise les champs de données définis dans le programme : les données initiales A et B puis les temporaires résultant de l'application des opérations effectuées. la fenêtre du milieu visualise l'opération courante. La dernière fenêtre donne les équations PEI générées automatiquement.

2.1.3 Parallélisme à flot de données

Dans le parallélisme à flot de données là aussi, on s'intéresse à un parallélisme à grain fin. Or à l'inverse du parallélisme de données, la nature même du modèle à flot de données suggère des représentations de graphes dirigés. Les environnements graphiques peuvent offrir ainsi des outils permettant en particulier l'édition de ces graphes.

Pour l'implémentation des programmes à flot de données, nous avons sélectionné TDFL qui permet de développer de nouveaux programmes ou d'adapter d'anciens programmes écrits de façon séquentielle ; GRAPE permet le développement d'algorithmes de traitement de signal ; et NL libère le programmeur de toute syntaxe textuelle.

TDFL (Task DataFlow Language) [SBKB90] est un langage graphique pour la programmation parallèle à flot de données. Il est dédié à l'écriture de nouveaux programmes et l'adaptation d'anciens programmes écrits dans des langages conventionnels. Il est indépendant de l'architecture et c'est le premier langage à flot de données supportant la modification dynamique des graphes de programmes.

Un programme TDFL est un graphe flot de données où chaque nœud est une fonction encapsulée. Les fonctions « nœuds » ne contiennent aucune opération de communication ou d'ordonnancement. En TDFL, la spécification de communication et de synchronisation est séparée de la spécification des unités de traitements. Le programme est écrit sur deux niveaux. Dans le premier niveau, le code séquentiel de chaque unité de traitement est spécifié dans des langages standards (en particulier C, Fortran, ou Pascal). Dans le deuxième niveau, plus haut que le précédent, les relations entre les unités de traitement sont spécifiées par des représentations graphiques. Ceci évite au programmeur d'inclure les primitives de communication et de synchronisation dans son code conventionnel. Il les traite séparément en utilisant des représentations graphiques compréhensibles. Ce second niveau inclut également certaines opérations fondamentales, telles que les itérations et la gestion du non-déterminisme.

TDFL a été implémenté sur plusieurs architectures, par exemple sur : Intel iPSC-1 ou un réseau de stations de travail IBM RT PC.

GRAPE À l'université de Louvain, Marc Engels et ses collaborateurs ont développé GRAPE (GRAphical Programming Environment), un environnement graphique pour le développement d'algorithmes de traitement de signal sur des machines multi-processeurs [Eng93]. Un algorithme est spécifié avec deux éditeurs : un éditeur graphique (à flot de données) et un autre textuel. La spécification consiste à dessiner un diagramme de blocs (cf. figure I.7). L'éditeur graphique fournit trois objets de base pour de tels diagrammes : sous-routines, fonctions, et connections. Les sous-routines représentent des blocs qui sont décrits à leur tour comme un diagramme de blocs complet. Elles supportent une conception hiérarchique descendante de la représentation graphique. La fonction est le niveau le plus bas dans la représentation graphique et est décrite à l'intérieur de manière textuelle. Une fois spécifié, le programme peut être compilé pour une machine mono-processeur DSP56001, pour deux ou quatre DSP56001 connectés en pipeline. GRAPE génère de l'assembleur. L'affectation des modules du programme aux processeurs se fait manuellement par l'utilisateur, par manipulation directe.

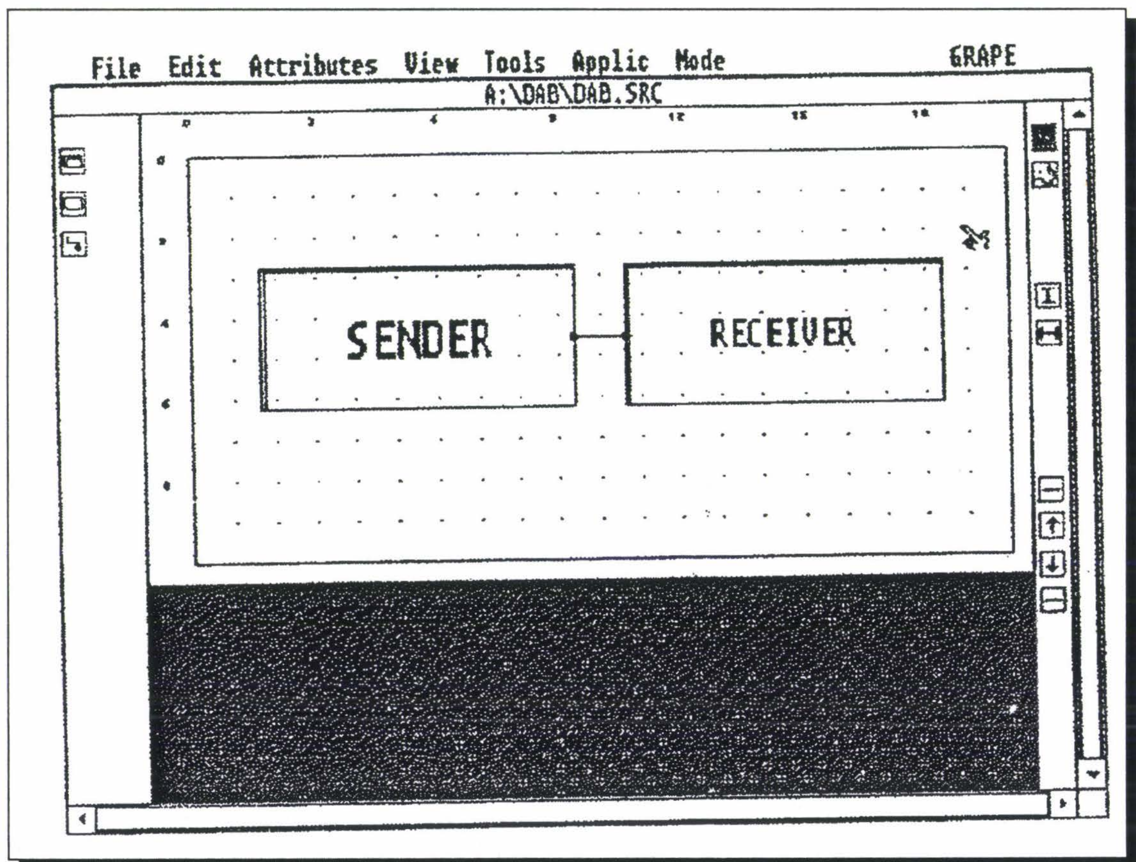


FIG. I.7 - *L'éditeur graphique de GRAPE*

NL Le langage NL, développé à l'université de Tasmania (Australie), est basé sur le traitement à flot de données [HM94]. Il est différent des environnements de programmation que nous avons cités, dans le sens où non seulement il donne des outils graphiques et interactifs

permettant d'assister le programmeur, mais mieux encore c'est un *langage de programmation visuelle*. Il libère le programmeur de toute syntaxe textuelle. Ainsi, nous avons préféré le décrire plus loin dans le second chapitre avec d'autres langages de programmation visuelle.

2.2 Déverminage de programmes

Le déverminage est une phase très importante dans le développement d'un programme. Le déverminage est défini comme la localisation, l'analyse puis la correction des erreurs. L'erreur est l'état où le programme n'effectue pas la fonction souhaitée. Comme elle est rarement située à l'endroit de sa manifestation, le programmeur doit acquérir des informations complémentaires à l'aide de traces ou de points d'arrêts dans des exécutions successives du programme [Roo94].

Le déverminage peut se faire pendant l'exécution, ou après l'exécution.

- Les « dévermineurs pendant l'exécution » permettent au programmeur d'interagir directement avec son programme en cours d'exécution et son programme source (références croisées). Le programmeur peut consulter le contenu des variables afin de contrôler leur validité. Il peut arrêter le programme, par exemple après l'exécution d'une fonction, pour contrôler aussi la validité de l'état de son application.
- Les « dévermineurs après exécution » permettent de consulter l'état d'un programme qui s'est arrêté anormalement. Le programmeur peut connaître l'endroit exact où l'exécution s'est arrêtée et visualiser l'état à cet instant. Ces dévermineurs se basent sur le principe de génération de traces pendant l'exécution de programmes.

Il faut noter cependant les problèmes particuliers à la programmation parallèle. Le premier concerne l'exécution non-déterministe des programmes parallèles, à l'inverse des programmes séquentiels où l'exécution est complètement déterministe (sauf utilisation de l'horloge, ou du générateur de nombres aléatoires). Ce problème provient du fait que l'ordre des événements peut être différent d'une exécution à l'autre à cause du non-déterminisme des langages parallèles, de la charge du réseau, etc. D'autres problèmes concernent les fonctionnalités des dévermineurs : comment visualiser des variables dans le cas d'une mémoire distribuée? comment arrêter simultanément ou presque toutes les activités d'un programme? l'exécution pas à pas d'un programme concerne t-elle une seule activité ou l'ensemble des activités? , etc.

Nous présentons dans cette section une dizaine de systèmes de déverminage dédiés aux parallélismes de tâches et de données. Nous n'avons malheureusement pas trouvé des dévermineurs pour le parallélisme à flot de données.

2.2.1 Parallélisme de tâches

Outre le déverminage classique montrant le code source et permettant par exemple la consultation des variables, les dévermineurs graphiques à parallélisme de tâches permettent également de visualiser l'animation des programmes à travers les communications et les interactions entre les tâches. Ils visualisent généralement un graphe de tâches, en montrant par exemple avec des couleurs différentes l'état des tâches (active, en attente de communication,...). Les communications peuvent être représentées par le coloriage des arcs. Ceci permet aux programmeurs de voir directement les problèmes tels que l'interblocage, la famine, etc.

Au niveau de ce type de parallélisme, nous retrouvons : αTrellis qui utilise les réseaux de Petri et la notion d'hypertexte pour naviguer dans un programme parallèle ; HeNCE que nous avons cité au niveau de l'implémentation de programmes ; PF-View développé par Utter et Pancake, il permet de déverminer des programmes Fortran représentés par une série d'icônes ; Trace facility un outil de Schedule, utilise les graphes de dépendances pour animer l'exécution du programme ; Visputer montre les communications en même temps sur le réseau matériel (réseau de Transputers) et sur la topologie logicielle.

αTrellis Stotts et Furuta de l'université de Maryland [SF90] utilisent les hypertextes pour naviguer dans un programme parallèle. Ils ont mis en œuvre αTrellis, un système hypertextuel dédié aux langages parallèles basés sur le modèle CSP (exemple : Occam). αTrellis se distingue des autres hypertextes par l'utilisation des réseaux de Petri qui ont l'avantage d'être des automates intrinsèquement parallèles. Un filtre convertit le programme source en documents hypertextes. Un document αTrellis peut être traité comme un programme parallèle abstrait montrant plusieurs fenêtres concurrentes et synchronisées. Celles-ci permettent de représenter le travail coopératif des processus. L'exploration hypertextuelle d'un programme aide l'utilisateur à observer les relations entre les composants d'un programme et à détecter les problèmes et les comportements inattendus dans le code. Le déplacement des jetons dans le graphe montre le transfert de contrôle entre les différentes parties du programme.

HeNCE L'environnement HeNCE [BDG+91], déjà évoqué au niveau de l'implémentation de programmes, peut visualiser et animer des vues de son exécution. Des statistiques variées sont enregistrées durant l'exécution pour donner la possibilité de les rejouer. Est offert aussi le mode trace, très utile pour analyser la performance du programme distribué (cf. figure I.8). Ces animations sont sensées montrer aussi le taux d'activité de chaque machine et de mettre en évidence les goulots d'étranglement et les charges non-équilibrées.

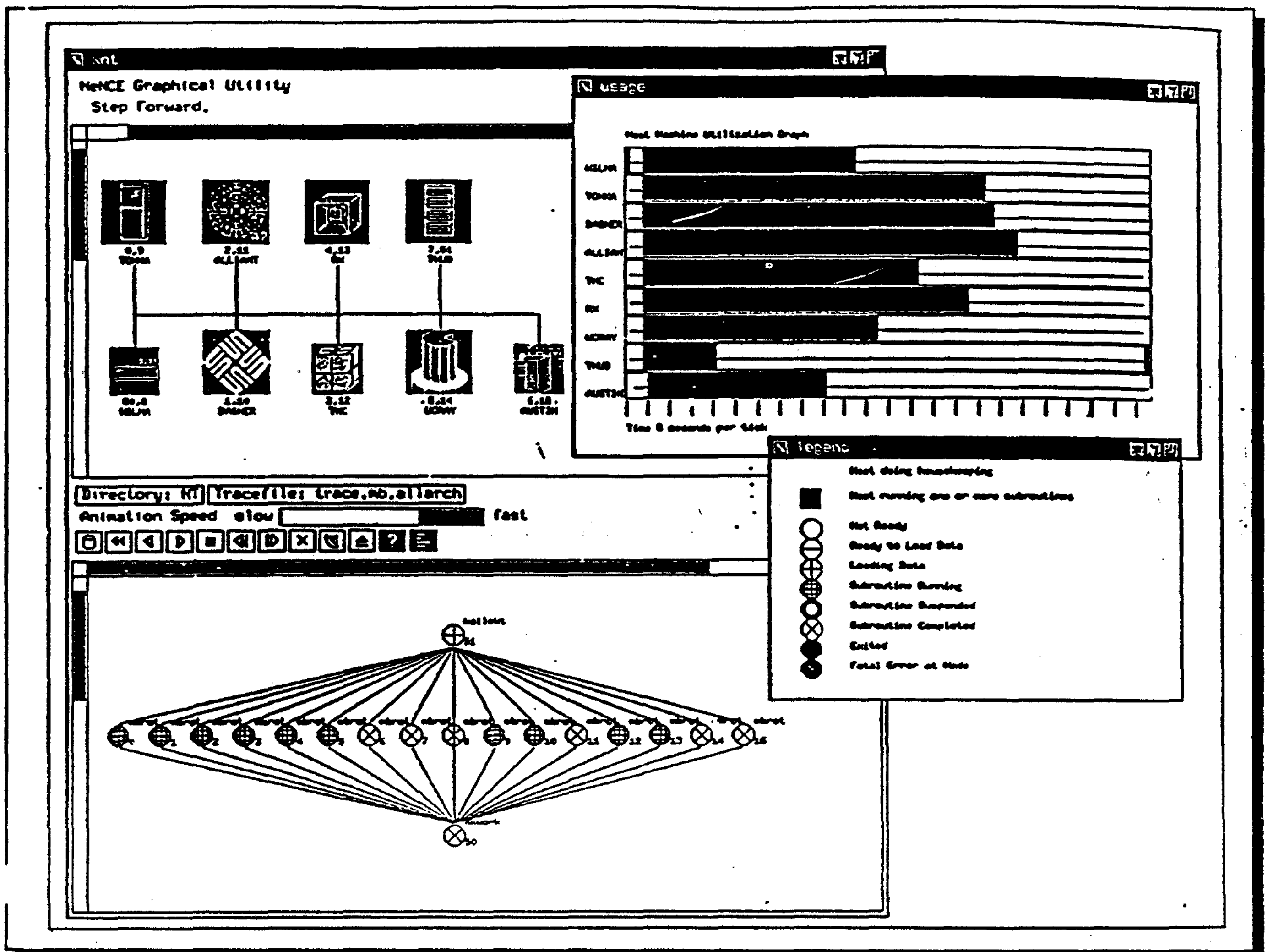


FIG. I.8 - *Le mode trace de l'interface graphique de HeNCE*

PF-View Sue Utter et Cherri Pancake [HP91] ont développé PF-View, un dévermineur parallèle graphique. Il se base sur une trace générée par un outil IBM PF-Trace (Parallel Fortran Trace facility). Comme les messages de cette trace sont difficiles à interpréter par un utilisateur, des techniques d'abstraction sont appliquées. Une représentation graphique de la structure du programme peut être obtenue en utilisant une corrélation, d'une part entre les événements de bas niveau enregistrés pendant l'exécution, et d'autre part les constructions de haut niveau du programme source. Cette représentation est mise en forme par diverses techniques de visualisation. Le programme est visualisé sous forme d'une série d'icônes (représentant les unités d'exécution) reliés entre eux pour représenter le flot d'exécution. Les icônes, distingués par des couleurs différentes, représentent par exemple les début et fin des tâches, des tâches en attente, les début et fin des boucles parallèles, les « Case » parallèles, etc. En cliquant sur un icône, l'utilisateur peut voir par exemple l'ensemble des processus participant à la construction, ou aussi quelle est la tâche qui se fait attendre. Cette visualisation

est synchronisée avec celle du code source en utilisant les références croisées.

Trace facility de Schedule Le module Trace facility de Schedule (évoqué plus haut), permet d'animer le comportement de l'exécution d'un programme [DSB92]. Pour pouvoir utiliser ce module, il suffit que l'utilisateur compile son programme avec la bibliothèque graphique de Schedule, au lieu de la bibliothèque régulière. L'exécution du programme permet de générer un fichier de trace. Dès que celui-ci est chargé, le graphe de dépendances est visualisé sur l'écran. Chaque nœud est représenté selon l'état dans lequel il se trouve à ce moment là. La figure I.9 montre un exemple de ré-exécution d'un programme.

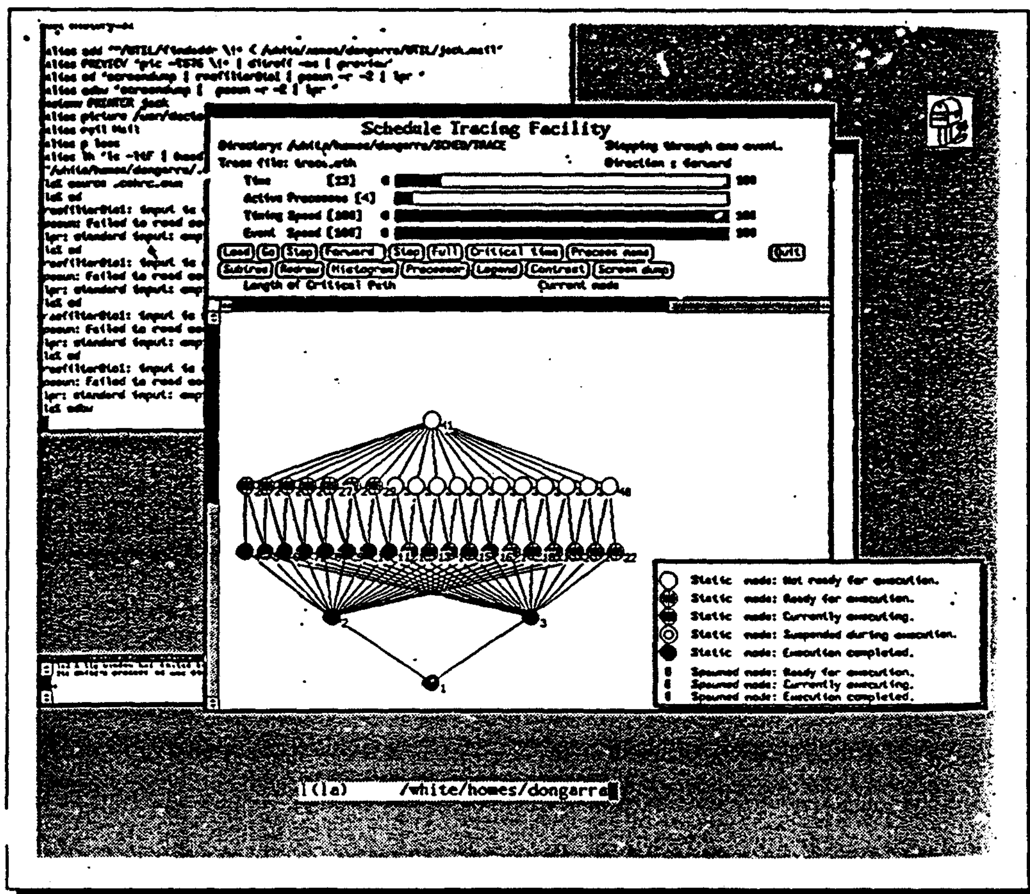


FIG. I.9 - *Un exemple de ré-exécution d'un programme Schedule sous Trace facility*

Cette animation aide à comprendre les problèmes de performances et à détecter éventuellement les goulots d'étranglement ainsi que les problèmes de synchronisation, puisque le graphe de dépendances possède de façon naturelle une telle interprétation graphique. D'autres moyens sont offerts aussi pour l'analyse de performances, retrouver par exemple le chemin qui a consommé le plus de temps ou évaluer le temps d'exécution de l'application entière.

Visputer En plus des facilités d'implémentation de programmes que nous avons déjà décrites, Visputer permet également d'animer l'exécution d'un programme [MZ94]. Une fois que le programmeur a décrit graphiquement son programme puis effectué le placement des tâches sur les Transputers, le pré-compilateur de Visputer insère un code de collection d'événements dans le programme Occam. Celui-ci est ensuite compilé par le compilateur Occam standard et exécuté sur le réseau de Transputers. Durant l'exécution, les événements sont collectés puis stockés dans un fichier d'événements.

La version actuelle de Visputer s'intéresse aux événements concernant les communications entre les processus s'exécutant simultanément. L'animateur extrait les informations du fichier d'événements, puis rejoue l'exécution en animant les communications sur un diagramme. Il montre les processus lorsqu'ils sont prêts à émettre, prêts à recevoir et lorsqu'ils communiquent. Ces événements sont d'un intérêt particulier, puisqu'ils peuvent indiquer les erreurs de communication tels que les inter-blocages, et donner à l'utilisateur une idée de l'attente d'un processus avant que la communication ne prenne effet.

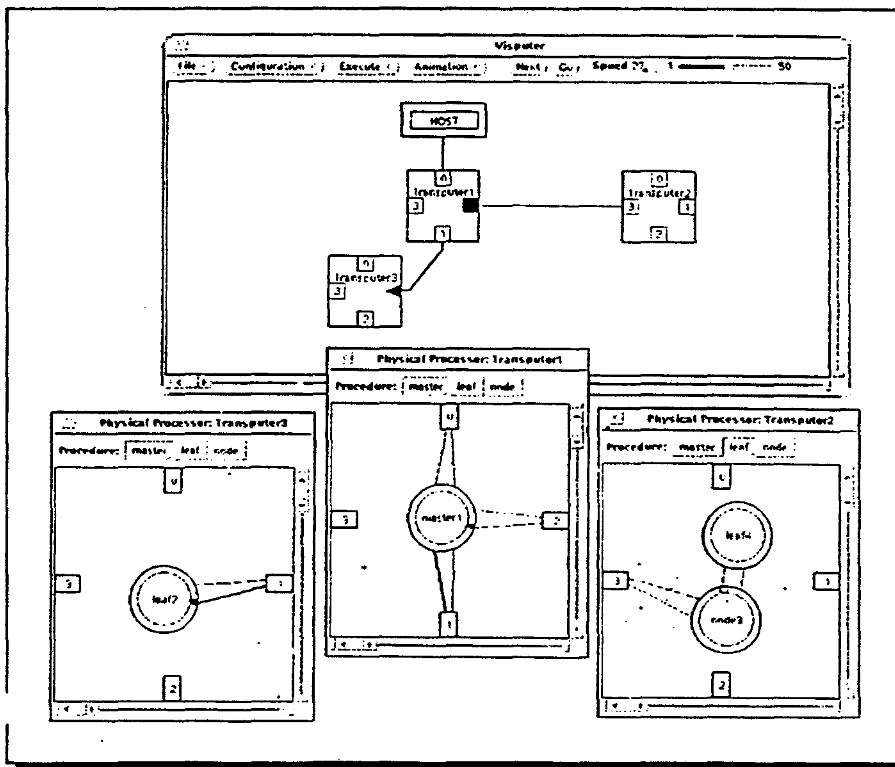


FIG. I.10 - *Animation d'un programme sous Visputer*

L'utilisateur observe les événements de communication sur le réseau logiciel et au même moment sur le réseau matériel. La figure I.10 montre un écran de Visputer lors d'une animation.

2.2.2 Parallélismes de données

Les dévermineurs à parallélisme de données s'intéressent en particulier à la visualisation des données ou éventuellement l'activité des processeurs pour les machines SIMD. Nous pouvons citer, à ce niveau, les dévermineurs fournis avec les machines, tels que MPPE pour la MasPar et PRISM pour la CM-5. Ils offrent des outils pour la visualisation classique du programme source, des points d'arrêt, l'inspection des expressions, etc.

MPPE Lors du déverminage, le programmeur peut contrôler et observer son programme s'exécuter sur la machine parallèle. Le dévermineur montre le code source, le programmeur peut y insérer des points d'arrêts, inspecter des variables locales ou globales, etc. La MasPar est une machine SIMD, lorsqu'on atteint un point d'arrêt, ce sont tous les processeurs élémentaires qui sont arrêtés en même temps.

MPPE offre également des outils de visualisation qui permettent au programmeur, même si le programme fonctionne bien, de voir comment la partie data-parallèle du programme utilise la grille des processeurs élémentaires. Le visualiseur de données est encore plus intéressant lorsque le programme est intrinsèquement visuel, par exemple un programme de traitement d'images.

Les deux outils de visualisation offerts par MPPE montrent une grille 2-D représentant le tableau des processeurs élémentaires de la MasPar [Dig92] :

- le visualiseur de données montre quelles sont les instances d'une variable parallèle qui ont des valeurs appartenant à un intervalle donné ;
- le visualiseur de machine montre l'activité des processeurs élémentaires à un instant donné.

MPPE permet également la visualisation de la grille des processeurs (en indiquant les processeurs actifs à un instant donné), et des variables parallèles (type Plural) en 2-D.

Prism Dans le traitement data-parallèle, il est souvent important d'obtenir une représentation visuelle des éléments définissant une variable parallèle ou entrant dans la construction d'une expression. Ces éléments, spécifiés de manière interactive, peuvent être visualisés sous Prism dans des formats textuels et graphiques variés [Thi92].

Les représentations incluent :

Texte les données sont visualisées par des chiffres et des caractères.

Niveaux de gris les valeurs sont visualisées par des niveaux de gris.

Table de couleurs à chaque valeur correspond une couleur (c'est l'utilisateur qui choisit la rangée des couleurs).

Seuillage à chaque valeur correspond un pixel noir ou blanc selon le seuillage spécifié.

Graphe les valeurs sont représentées dans un graphe où l'index de la donnée correspond à l'abscisse (d'un point du graphe) et la valeur correspond à l'ordonnée.

Surface visualisation en trois dimensions des contours d'une partie bi-dimensionnelle des données (*cf.* figure I.11).

Vecteur visualisation de données complexes par des vecteurs. Prism permet de visualiser des données ou des intervalles de données. Il offre aussi la possibilité d'examiner rapidement de grand tableaux en sélectionnant des sous-tableaux (formes géométriques).

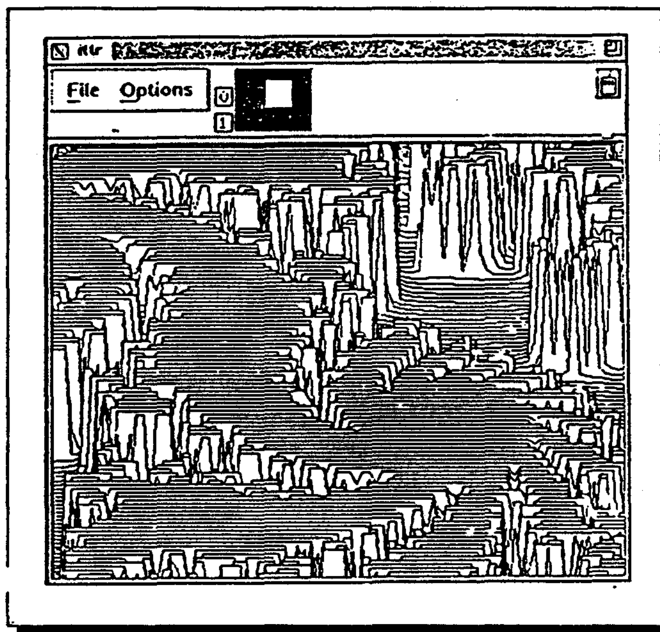


FIG. I.11 - *Prism: un exemple de visualisation de surface*

Un outil permettant de naviguer à travers les données visualisées est offert à l'utilisateur. Si un tableau parallèle est multi-dimensionnel, le visualiseur de données montre des coupes du tableau. L'outil de navigation permet de sélectionner les axes à visualiser et la position de la coupe.

Les programmes Fortran 77 peuvent être convertis en CM-Fortran en utilisant le convertisseur CMAX. Pour de tels programmes, Prism donne la possibilité de les déverminer et de les analyser en utilisant un code source Fortran 77 ou CM-Fortran.

Autres D'autres systèmes de déverminage de programmes data-parallèles seront traités dans le second chapitre avec les systèmes de visualisation de programmes parallèles. Nous verrons en particulier : IVE [FLK+91] dédié aux machines SIMD, en particulier la Connection Machine (*cf.* 4.2.1) ; et LIVE [LSCA93] qui est indépendant du langage de programmation(*cf.* 4.2.3).

2.3 Transformation de codes (parallélisation et optimisation)

Afin de tirer avantage des machines parallèles à moindre coût, les scientifiques préfèrent souvent transformer leurs énormes codes séquentiels plutôt que de réécrire leurs applications de manière parallèle. La transformation intervient sur deux niveaux : (1) optimiser des parties du code, par exemple transformer deux boucles successives en une seule boucle ; (2) ou paralléliser des codes. Selon l'architecture de la machine cible, la parallélisation peut porter sur l'identification (syntaxique) de tâches pouvant s'exécuter de manière concurrente puis assurer leur distribution sur des machines MIMD, ou l'identification dans une boucle d'instructions pouvant être réécrites comme des opérations vectorielles à exécuter sur des machines SIMD (appelée aussi *vectorisation*).

La transformation de code nécessite une analyse préalable du programme permettant d'en extraire plusieurs représentations intermédiaires : le graphe du flot d'exécution du programme, graphe de dépendances de données, graphe d'appels, et/ou graphe de tâches. Plusieurs paralléliseurs *automatiques* ont été développés, ils se basent dans leur analyse sur le code source et les représentations intermédiaires pour générer un programme parallèle. Malheureusement, les résultats ne sont pas toujours satisfaisants : la parallélisation est presque souvent incomplète bien que faisable manuellement. De ce fait, l'intervention manuelle devient nécessaire, quoique plusieurs problèmes peuvent surgir à ce moment là : décider quelles transformations utiliser, comment les ordonner, quelles sont les parties du code à transformer, etc. C'est là qu'interviennent les paralléliseurs *semi-automatiques* : ils interagissent avec l'utilisateur afin de choisir les bonnes décisions, puis traiter les parties non détectables de façon automatique.

Plusieurs environnements sont développés dans cet esprit, par exemple : Parafrase-2 un vectoriseur/paralléliseur[2] ; Faust qui supporte plusieurs langages vectoriels ; ou VOSpeL qui transforme des programmes Fortran en une forme vectorielle ou data-parallèle. Ces systèmes sont décrits dans cette section avec d'autres que nous citerons brièvement, tels que ParaScope, PAT, et Forge 90.

Parafrase-2 est classifié dans le tableau I.1 avec les systèmes à parallélisme de données bien qu'il appartienne également à la première classe. Les autres systèmes ne concernent cependant que le parallélisme de données.

2. Dans notre classification, les vectoriseurs sont considérés avec les systèmes à parallélisme de données.

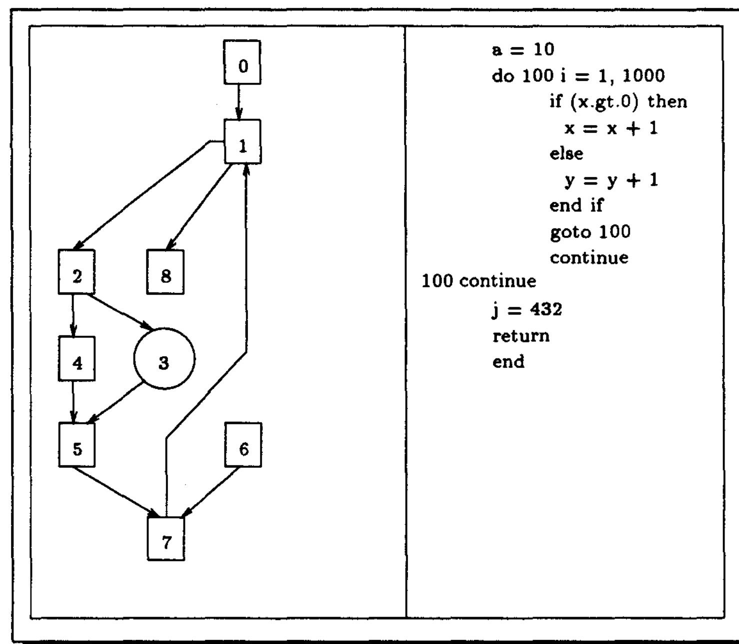


FIG. I.12 - *Une partie de l'interface graphique de Parafrase-2*

Parafrase-2 L'environnement Parafrase-2 [PGH+89] est un vectoriseur/paralléliseur de source à source. Il peut extraire du parallélisme à partir d'une boucle ou identifier des groupes d'instructions pouvant s'exécuter de façon concurrente. Il traite plusieurs langages, en particulier C, Fortran et Pascal. Un pré-processeur analyse le code séquentiel en entrée puis génère plusieurs représentations de structures de données: graphe de dépendances, graphe du flot du programme, graphe d'appels, et graphe de tâches. Ces représentations sont visualisées à travers une interface graphique. L'utilisateur peut par exemple localiser les dépendances puis demander des informations concernant la cause de ces dépendances. Il peut éventuellement supprimer des dépendances pour autoriser la parallélisation. La figure I.12 montre un programme simple avec le graphe de flot associé. Chaque nœud du graphe est essentiellement un bloc d'instructions d'affectation. L'utilisateur peut, à l'aide d'un menu, consulter le contenu d'un nœud donné. Le système mettra en évidence les lignes du code correspondant à ce nœud.

À partir de ces représentations, Parafrase-2 applique sa parallélisation puis regénère des représentations semblables mais concernant cette fois le code parallèle. Un post-processeur prend en entrée les nouvelles représentations qu'il transformera enfin en un code textuel correspondant au langage initial.

Faust Vincent Guarna *et al.* [GGJ+89] à l'université de l'Illinois ont développé Faust: un environnement graphique pour la programmation parallèle. Il est dédié aux applications scientifiques et tourne sur des stations de travail. Il intègre plusieurs outils interactifs permettant d'assister le programmeur à différents niveaux du développement (compilation, optimisation,

déverminage, et analyse de performances). Il offre en particulier *Sigma*, un outil de parallélisation. Ce dernier permet de mettre au point un code parallèle généré (parallélisé) automatiquement ou d'optimiser un algorithme parallèle. Il se base pour son traitement sur une base de données constituée et gérée par Faust. La base contient l'analyse des dépendances (intra et inter-procédurales), un graphe flot de contrôle ayant assez d'informations pour regénérer le fichier source (même les commentaires), et l'analyse du code objet généré par le compilateur. L'utilisateur peut de façon conviviale consulter la base de données et avoir toutes les informations qu'il souhaite (exemple : quelle routine utilise cette variable? cette routine peut-elle être vectorisée? il peut demander des statistiques, etc.). Sigma supporte plusieurs langages : Fortran vectoriel pour l'Alliant, Cedar Fortran pour la machine multi-processeurs Cedar, C parallèle orienté objet (similaire à C++ et au Cedar C parallèle). La représentation graphique de la forme interne du code permet de guider le système dans les transformations. L'utilisateur sélectionne la partie à modifier puis choisit interactivement (à travers des menus) la transformation nécessaire (exemple : vectorisation de boucle, parallélisation, des transformations spécifiques à la machine, etc.). Si la transformation choisie tente de modifier la sémantique originale du programme, l'utilisateur sera tout de suite prévenu par le système.

VOSpeL Chang et al. [DCS92] à l'université de Pittsburgh ont mis en œuvre VOSpeL (Visual Optimization Specification Language), un système permettant d'exprimer graphiquement et interactivement des transformations de programmes. Il parallélise des programmes Fortran en une forme vectorielle ou data-parallèle. La représentation intermédiaire du programme à paralléliser est une forme visuelle du graphe de dépendances (VPDG: Visual Program Graph Dependence) qui permet à la fois la transformation et la visualisation du code. Les graphes de dépendances de programme (PDG) montrent explicitement les dépendances de données et de contrôle dans le programme. Ils permettent à l'utilisateur de voir le programme à plusieurs niveaux d'abstraction et de retrouver directement les instructions pouvant s'exécuter en parallèle. VOSpeL offre en particulier la possibilité d'annuler (undo) une transformation. En effet, l'utilisateur peut appliquer une série de transformations et se rendre compte par la suite qu'une d'entre elles est inappropriée et empêche d'autres transformations. VOSpeL garde l'historique des transformations permettant à l'utilisateur de revenir en arrière. À la différence de beaucoup de systèmes de parallélisation, VOSpeL permet de montrer à la fois le code à transformer, sa forme intermédiaire et le code après transformation ; les trois représentations sont synchronisées et mettent en évidence la partie du code à transformer.

Autres Il existe d'autres systèmes graphiques de parallélisation, par exemple : ParaScope développé à l'université de Rice [KMT91]. Il est conçu en particulier pour restructurer des programmes Fortran séquentiels en leur forme parallèle. Les programmes peuvent être parallélisés de manière automatique ou manuelle interactive. Il génère du code pour les machines Cray et Sequent. Le système interactif PAT (Parallelization AssistanT) génère à partir de pro-

grammes Fortran 77 des programmes parallèles pour les machines KSR-1, IBM3090, Sequent, et Cray Y-MP [AMS89]. Selon une évaluation de la NASA [Che93], le système Forge 90[3] transforme des programmes Fortran 77 en Fortran 90 ou HPF. Il supporte les machines Cray Y-MP, Intel iPSC/860, Delta, Paragon, CM2, CM5, et des réseaux de stations utilisant PVM ou Express. Le rapport de la NASA fait remarquer cependant que le système Forge 90 n'arrive pas à paralléliser certaines boucles parallélisables par PAT.

2.4 Analyse de performances

L'analyse de performances doit permettre de déterminer les causes du comportement erroné ou inacceptable d'un programme. Les performances d'un programme exécuté sur une machine parallèle peuvent être affectées par plusieurs facteurs tels que l'algorithme, le langage de programmation, le compilateur, et le système d'exploitation. Les performances peuvent se référer à plusieurs métriques, par exemple la qualité de l'algorithme, vitesse théorique par rapport à la vitesse réalisée, utilisation des processeurs, charge du réseau, et performance du cache.

Un bon outil d'analyse de performance doit être capable de répondre de façon raisonnable aux questions suivantes [NCA93] :

- Comment s'exécute le programme et ses procédures?
- Comment peut-on expliquer les performances du programme?
- Comment peut-on augmenter ces performances?

Des outils plus ambitieux doivent être capables de répondre à des questions telle que : « Que se passe-t-il si la taille du problème ou le nombre de processeurs augmente? ». Lors de l'évaluation, on a souvent besoin d'exécuter les programmes sur plusieurs architectures. Les outils d'analyse doivent pouvoir s'adapter à plusieurs architectures différentes.

Plusieurs analyseurs ont été mis en œuvre, nous en décrirons quelques uns qui sont graphiques. PAWS et ChaosMON sont indépendants du langage de programmation et de l'architecture. Le premier travaille avec des formes intermédiaires et le second est générique et s'adapte à une application spécifique. VISTA utilise un même modèle d'analyse pour des programmes SPMD et data-parallèles. MIN-Graph est dédié aux architectures Multi-processeurs BBN. Projections, basé sur le langage Charm, offre une visualisation de performances qualifiée d'intelligente. Il montre des informations spécifiques au système et au programme. Nous verrons également la partie analyse de performance des deux systèmes MPPE et Prism que nous avons décrits au niveau du déverminage (*cf.* 2.2.2).

3. Forge 90 est commercialisé par Applied Parallel Research Inc.

PAWS, MPPE, Prism et VISTA sont des systèmes d'analyse de performances pour le parallélisme de données, alors que les autres systèmes: ChaosMON, MIN-Graph et Projections, concernent le parallélisme de tâches.

PAWS L'objectif du système PAWS (Parallel Assessment Window System) est de pouvoir analyser des applications parallèles indépendamment du langage et de l'architecture cible [PGA+91]. PAWS est un environnement graphique composé de quatre outils. Le premier transforme l'application écrite dans un langage de haut niveau (exemple: Ada) en un graphe flot de données (décrit en IF1). Ce graphe doit représenter les dépendances de données et exprimer les différents niveaux de parallélisme (selon la granularité). Cette forme intermédiaire du programme est indépendante de la machine; elle peut être compatible avec diverses architectures. Le premier outil est en fait un paralléliseur automatique permettant d'extraire du parallélisme de données. Le deuxième outil permet à l'utilisateur de manipuler une base de données contenant diverses descriptions de machines. L'utilisateur peut introduire, modifier, ou effacer les caractéristiques d'une architecture donnée. Cette approche permet aux utilisateurs d'évaluer des machines conceptuelles avant toute construction réelle. Le troisième outil s'occupe de l'analyse elle-même. Il traite les résultats émanant des deux premiers outils (représentation graphique du programme et caractéristiques de la machine), puis génère son analyse. L'analyse est visualisée à travers une interface graphique. Elle montre d'une part le parallélisme idéal dépendant seulement du graphe de l'application, et d'autre part le parallélisme prédit en fonction du placement de ce graphe sur la machine cible. PAWS supporte les architectures SIMD (exemple: CM-2) et MIMD (exemple: Encore Multimax).

ChaosMON Kilpatrick et Schwan [KS91] ont eux aussi proposé un système d'analyse de performances indépendant du langage et de l'architecture. Leur système graphique ChaosMON est générique. L'analyse peut être spécifique à une application donnée. À la différence des systèmes automatiques, qui réalisent une instrumentation et une visualisation automatique, ChaosMON est semi-automatique. Il offre à l'utilisateur des outils lui permettant de concevoir sa propre visualisation. L'utilisateur crée d'abord une description de haut niveau de son application puis spécifie des modèles de performances basés sur cette description. Celle-ci est stockée dans une base de données gérée par le système. Les modèles créés peuvent être réutilisés avec d'autres applications de même classe. Il suffit d'instancier d'une part les composants du modèle à ceux du programme (processus) et d'autre part les attributs des composants du modèle aux variables du programme. Le système ChaosMON est dédié aux programmes multi-tâches. Il a été expérimenté sur deux plate-formes: des réseaux de stations UNIX et des machines BBN Butterfly. Pour les stations UNIX, les applications sont écrites en C et les processus communiquent à travers des primitives de communication standards d'UNIX. Pour la deuxième plate-forme, les applications sont écrites en CHAOSarc, un langage expérimental basé sur les objets et dédié aux applications temps-réel.

VISTA Rover et Wright [RW93] ont conçu un même modèle d'analyse de performances pour des programmes SPMD et data-parallèles. Leur système graphique d'analyse de performances VISTA (Visualization and Instrumentation of Scalable mulTicomputer Applications) a été conçu en particulier pour les machines à mémoire distribuée nCUBE 2 (MIMD) et MasPar MP-1 (SIMD). Il prend en compte les deux parties de l'analyse : l'instrumentation du programme et la visualisation des performances. Les informations sont traitées de la même façon dans les deux modèles SPMD et data-parallèle et sont présentées à travers une hiérarchie de diverses vues. Ces vues se basent, comme dans beaucoup de systèmes de visualisation de performances, sur les techniques de visualisation de données et les graphiques multi-dimensionnels.

MIN-Graph Zhang et al. [ZNQ93] ont développé MIN-Graph (Multistage Interconnection Network), un autre système d'analyse de performances conçu pour une machine spécifique. Il est dédié aux architectures multi-processeurs BBN basées sur les réseaux multi-étages. MIN-Graph visualise en particulier les performances concernant les communications inter-processeurs, l'ordonnancement des processus, et les accès mémoires-distantes. Le modèle de programmation est le multi-tâches.

Projections La clarté des informations visuelles offertes à l'utilisateur dépend d'une part de l'information extraite du programme et d'autre part des techniques utilisées dans la visualisation. Malheureusement, les compilateurs ne sont pas toujours capables d'extraire toutes les informations dont on pourrait avoir besoin. Il existe cependant des langages parallèles capables de fournir des informations spécifiques à un programme. Ces informations décrivent les événements pouvant surgir dans le programme et précisent leur impact potentiel sur les performances. Sinha et Kalé [SK94] ont mis en œuvre Projections, un système d'analyse offrant une visualisation intelligente des performances. Il est construit autour du langage Charm qui est une extension du C pour le parallélisme de tâches. Charm est dédié aux systèmes à mémoire partagée et les systèmes à passage de message ; il est orienté objet et dirigé par les message. Projections offre des informations spécifiques au système et au programme, telles que longueur des files d'attente, création et traitement des messages, création de nouvelles tâches, etc. Selon les auteurs, Projections offre une compréhension plus raffinée que ce qui est offert par les outils traditionnels (ceci est dû particulièrement à l'utilisation de Charm).

Prism et MPPE Les deux systèmes incorporent des outils d'analyse de performances. MPPE [Dig92] offre une analyse plus riche avec le profiling, alors que PRISM [Thi92] ne donne que des statistiques et quelques recommandations pour assister l'utilisateur à isoler des goulots d'étranglement de performances [Che93].

Les données fournies par Prism incluent :

- le temps système ;
- le temps de traitement ;
- le temps passé dans le transfert des données entre le processeur de contrôle et les nœuds ;
- le temps passé dans le réseau pour les communications générales ;
- le temps passé dans les combinaisons spécifiques de communication, tel que le plus-proche-voisin sur une grille ;
- le temps passé dans les réductions et les opérations *prefix* (pour ces dernières opérations, voir par exemple CMF [Thi90] ou HPF [For93].

Les données résultant de l'analyse de performances sont généralement visualisées par des histogrammes.

Autres Hackstadt et al. [HMM94] de l'université de l'Oregon ont orienté leur travaux pour étudier de nouvelles techniques et méthodes de visualisation. Il traitent en particulier le problème de l'extensibilité des vues au cours de l'exécution. Les vues ne doivent pas changer de format, de taille, de signification, ou de clarté lorsque des processeurs sont rajoutés à l'exécution. Ces techniques ont été expérimentées sur des programmes écrits dans deux langages de programmation data-parallèle : Dataparallel C et pC++.

Divers outils et méthodes sont décrits dans le journal [JPD93] qui a consacré un numéro spécial intitulé « Tools and methods for visualization of parallel systems and computations ». Par exemple, Natarjan, Chiou et Ang [NCA93] décrivent leurs expériences concernant la collecte des informations (instrumentation), l'analyse, et la visualisation des performances pour la machine Monsoon. Celle-ci est une machine multi-processeurs à flot de données construite par Motorola en collaboration avec le MIT. Sarukkai et Gannon [SG93] proposent une nouvelle approche pour la conception d'outils d'analyse de performances. Ils étudient des méthodes permettant au programmeur de spécifier la visualisation qu'il souhaite avoir. Ils ont développé une interface graphique basée sur les tableurs permettant d'accéder au code source de l'application. Celle-ci est présentée à travers une base de données relationnelle.

3 Conclusion

Nous avons décrit plusieurs systèmes et environnements graphiques permettant d'assister le programmeur dans plusieurs phases du développement d'une application parallèle. Nous

avons classifié ces systèmes afin de mettre en évidence l'apport de ces environnements au niveau de chaque modèle de programmation parallèle *cf.* table I.1. Ceci nous a particulièrement montré la différence du nombre d'outils entre le parallélisme de tâches et le parallélisme de données qui est notre modèle d'intérêt. Le manque d'outils graphiques au niveau de la conception et de l'implémentation est encore plus important qu'au niveau des autres phases de développement. Les outils qui existent par ailleurs dans les autres modèles de programmation parallèles assistent le programmeur mais ne le libèrent pas de la syntaxe, de l'apprentissage du langage, ou de la connaissance de l'architecture cible. Or, ceci est possible comme nous allons le démontrer dans le chapitre suivant.

Chapitre II

Programmation visuelle et visualisation de programmes

Pour bénéficier des performances offertes par les machines les plus puissantes, les scientifiques sont de plus en plus amenés à écrire des programmes dans des langages très spécialisés. Ils doivent de plus les implémenter sur des architectures spécifiques, par exemple parallèles, afin de tirer le maximum de profit en terme de puissance de calcul de ces machines. Le problème qui se pose est qu'ils sont souvent obligés d'apprendre un langage, respecter rigoureusement une syntaxe, et/ou connaître l'architecture cible. Faut-il que les physiciens soient des informaticiens ou est-ce qu'il est possible de leur éviter cela? Une approche de ce problème serait d'étudier l'utilisation des graphiques comme langage de programmation. Ceci a été appelé « programmation visuelle » ou « programmation graphique ». Quelques systèmes de programmation visuelle ont montré avec succès que des scientifiques peuvent créer des programmes assez complexes après un petit apprentissage [Hal84].

La programmation visuelle est devenue ces dernières années un domaine important de la recherche informatique [McI92]. Les objectifs à court terme sont de permettre aux débutants de programmer de simples applications pour eux-mêmes, et de rendre les programmeurs professionnels plus productifs dans des domaines spécialisés. Un objectif à long terme est la création de langages de programmation visuelle plus puissants et plus expressifs que les langages textuels.

Une autre classe de systèmes essaie de rendre les programmes plus compréhensibles en utilisant des graphiques pour illustrer les programmes après qu'ils aient été créés. Ils sont appelés systèmes de « visualisation de programmes » et sont habituellement utilisés lors du déverminage ou pour enseigner aux étudiants comment programmer[1].

1. Des exemples d'universités utilisant la visualisation de programmes, pour enseigner la programmation sont : « George Mason University, USA », « Waikato University, New Zealand », « Auckland University, New

Ce chapitre tente de donner une définition plus formelle de ces termes et montre l'intérêt d'utiliser les graphiques en programmation. Ensuite, les approches diverses de la programmation visuelle et la visualisation de programmes sont illustrées à travers une étude de systèmes pertinents.

1 Définitions

Comme pour tout concept nouveau, de longues discussions ont eu lieu pour définir la programmation visuelle, la visualisation de programmes, et de manière globale les langages visuels. Selon Brad Myers [Mye90b], dans le passé plusieurs systèmes de visualisation de programmes ont été incorrectement qualifiés de programmation visuelle (comme dans [GI85]). Il existe, jusqu'à présent, des personnes qui tentent sciemment ou par ignorance de qualifier certains langages de visuels, alors qu'ils ne le sont pas (comme par exemple Visual C++ de MicroSoft).

Pour éviter toute équivoque, nous avons préféré définir ces termes : programmation visuelle, visualisation de programmes, et langages visuels. Nous considérons dans tout ce qui suit qu'un langage de programmation doit permettre de manipuler des variables, des conditions et des itérations, éventuellement de façon implicite.

Programmation visuelle

La « programmation visuelle » se réfère à tout système permettant à l'utilisateur de spécifier un programme de manière bi-(ou multi)dimensionnelle. Bien que ce soit une très large définition, les langages textuels conventionnels ne sont pas considérés bi-dimensionnels puisque les compilateurs ou les interpréteurs les traitent en longueur en tant que flots mono-dimensionnels d'instructions. La programmation visuelle n'inclut pas les systèmes qu'utilisent les langages de programmation conventionnels (linéaires) pour définir des images, tels que Sketchpad, CORE, PHIGS, Postscript, la boîte à outils (toolbox) XToolKit, X11, etc. Elle n'inclut pas non plus les outils de dessins comme MacDraw ou XFig, puisque ceux-là ne créent pas des programmes. [Mye90b]

Les langages de programmation visuelle devraient être eux-mêmes présentés visuellement. Les constructions de programmation et les règles pour combiner ces constructions devraient être aussi présentées visuellement [Cha87]. Enfin, le programme visuel est un ensemble d'expressions graphiques créées à partir de telles constructions. Le terme « bi-dimensionnel » est

Zealand », « Oregon State University, USA », « University of Washington », « Carnegie Mellon University », ...

employé car en effet ces expressions sont réparties spatialement en deux ou plusieurs dimensions (exemple : un organigramme ou un graphe flot de données).

Visualisation de programmes

La « visualisation de programmes » est construite sur un concept entièrement différent de celui de la programmation visuelle. En programmation visuelle, les graphiques sont utilisés pour créer le programme lui-même, mais dans la visualisation de programmes, le programme est spécifié de manière textuelle et les graphiques sont utilisés pour illustrer certains aspects du programme ou de son exécution.

Selon Myers, si un programme créé en utilisant une programmation visuelle est à visualiser ou à déverminer, évidement cela sera fait de manière graphique qui pourrait être considérée comme une forme de visualisation de programmes. Il est par contre plus précis d'utiliser le terme programmation visuelle pour des systèmes permettant à un programme d'être créé en utilisant des graphiques, et visualisation de programmes pour les systèmes utilisant les graphiques seulement pour illustrer des programmes après qu'ils aient été créés.

Langages visuels

Les « langages visuels » se réfèrent à tous les systèmes qui utilisent les graphiques, incluant les systèmes de programmation visuelle et de visualisation de programmes. Bien que tous ces termes soient en quelque sorte similaires et équivoques, il est important d'avoir différents noms pour les différents types de systèmes, ce sont les noms conventionnellement utilisés dans la littérature. [Mye90b]

Ainsi nous n'utiliserons, dans ce qui suit, le terme « langage visuel » que si les deux aspects « programmation visuelle » et « visualisation de programmes » sont concernés ensemble.

2 Avantages de l'utilisation des graphiques

Programmation visuelle et visualisation de programmes sont des idées très attirantes pour de nombreuses raisons. Le système visuel humain et le traitement de l'information visuelle sont clairement adaptés pour des données multi-dimensionnelles [Mye90b]. Les programmes informatiques par contre sont habituellement présentés dans une forme textuelle n'utilisant pas la capacité totale du cerveau. Des présentations bi-dimensionnelles de programmes, telles que des organigrammes ou encore l'indentation des programmes, ont été longtemps connues

comme étant utiles à la compréhension des programmes. De nombreux systèmes de visualisation de programmes ont démontré que des images 2-D des structures de données, comme celles que nous dessinons à la main, sont très utiles. Clariss [CC86] prétend que la programmation graphique utilise l'information dans un format très proche des représentations mentales des problèmes, et qu'elle permet aux données d'être traitées dans un format en adéquation avec la façon dont les objets sont manipulés dans le monde réel. Il semble clair qu'un style de programmation plus visuel pourrait être plus facile à comprendre pour les humains, particulièrement pour des débutants.

Une autre motivation pour utiliser des graphiques est l'abstraction proposée aux programmeurs : nous obtenons ainsi un langage de haut niveau permettant de faciliter le travail de développement. Ils permettent de rendre visibles certains aspects invisibles d'un programme mais conceptuellement importants [RW91]. Ceci est particulièrement vrai lors du déverminage où les graphiques peuvent être utilisés pour montrer beaucoup plus d'informations concernant l'état du programme (telles que structures de données et variables courantes) qu'il est possible de le faire avec des représentations purement textuelles. Aussi, certains types de programmes complexes, tels que ceux utilisant des processus concurrents ou concernant les systèmes temps-réels, sont difficiles à décrire avec des langages textuels, là encore des spécifications graphiques peuvent être plus appropriées.

La popularité des interfaces à « manipulation directe » [Shn83, ZF88], où des itèmes sur l'écran peuvent être pointés et manipulés via la souris, contribue aussi à l'intérêt des langages visuels. Étant donné que beaucoup de langages visuels utilisent des icônes et d'autres objets graphiques, leurs éditeurs ont habituellement une interface utilisateur à manipulation directe. L'utilisateur a l'impression de construire un programme de manière plus naturelle que la conception abstraite proposée par les langages classiques.

3 Langages de programmation visuelle

L'utilisation des langages de programmation visuelle est de plus en plus répandue, on doit définir des critères permettant de les distinguer les uns des autres. Brad A. Myers [Mye90b] classifie les langages de programmation selon trois critères orthogonaux. (1) *Programmation visuelle ou non* : ce critère permet de distinguer les langages de programmation visuelle des langages textuels conventionnels, tels que Pascal, Fortran, Lisp, etc. Les 37 exemples de langages de programmation visuelle que donne Myers pour illustrer cette classe constituent un point d'accès pour un utilisateur non averti : ils expliquent plus concrètement la définition de la programmation visuelle. (2) *Programmation basée sur les exemples ou non* : Brad Myers a tenu à distinguer les systèmes à travers cette technique de spécification de programmes, ses activités de recherche se basent particulièrement sur ce type de programmation. Les langages

de programmation basée sur les exemples donnent la possibilité à l'utilisateur de réaliser des actions sur des exemples concrets (souvent par manipulation directe), et construisent en même temps un programme abstrait [Mye92]. C'est une technique de spécification différente des autres techniques (programmation basée sur les organigrammes, les graphes flot de données, etc.), mais ne constitue qu'un petit sous ensemble de langages de programmation visuelle. Un critère basé sur l'ensemble des techniques de spécification permettrait de couvrir la totalité des langages de programmation visuelle. (3) Enfin le troisième critère distingue les langages compilés des langages interprétés.

David W. McIntyre s'est intéressé dans sa thèse [McI92] aux langages de programmation basée sur les icônes : « programmation icônique ». Dans un langage de programmation icônique, l'utilisateur compose des programmes principalement en définissant, sélectionnant et plaçant des icônes à l'intérieur d'un espace dédié à la programmation. Ces éléments sont manipulés par l'utilisateur de manière interactive et selon une certaine grammaire spatiale pour la construction de programmes [GR90]. David McIntyre partage les langages de programmation visuelle en deux parties : langages de programmation icônique ou non. Il regroupe dans la première partie les langages de programmation basée sur les graphes de contrôle et ceux basés sur les graphes flot de données. Il existe pourtant des langages basés sur d'autres techniques de spécification qui peuvent être qualifiés d'icôniques : nous en décrirons quelques uns plus loin. Seuls les langages basés sur les exemples ou les contraintes (exemple : Peridot), appartiennent à la deuxième partie « programmation non icônique » : il n'y a pas de grammaire spatiale. Ces langages sont très souvent interprétés.

Margaret Burnett et Marla Baker [BB94] proposent eux aussi une classification, que l'on peut considérer comme trop détaillée : il y a six classes avec une quarantaine de branches. Il y a malheureusement certains critères très équivoques d'autant plus que les auteurs laissent le choix d'interprétation à l'utilisateur ; ils ne les définissent pas. Nous retrouvons néanmoins des critères importants pour la description d'un langage de programmation visuelle, tels que « language purpose » et « procedural abstraction ». Le premier permet de préciser le *domaine d'utilisation* du langage (programmation générale, traitement d'images, construction d'interfaces utilisateurs, etc.). Le second critère *abstraction procédurale* signifie qu'une partie du programme peut être condensée ou encapsulée en un seul objet (élément de base de la représentation visuelle d'un programme), par exemple un nœud lorsqu'il s'agit de graphes. Cet objet encapsulé peut représenter une procédure. L'abstraction de procédure est importante pour deux raisons. Premièrement, elle permet en compactant le programme d'économiser l'espace de l'écran, qui constitue un des grands problèmes de la programmation visuelle [KvBS92, Hil92]. Deuxièmement, elle permet à l'utilisateur de voir le programme à un niveau plus abstrait sans être obligé de voir donc tous les détails.

Dans ce qui suit, nous décrirons les différentes techniques de spécification de programmes. Nous illustrons chaque technique avec un exemple de langage de programmation visuelle. Nous verrons en particulier, pour tout langage étudié,

- son domaine d'utilisation ;
- comment il utilise la technique de spécification ;
- s'il permet ou non l'abstraction procédurale ; et
- comment il définit les constructions essentielles d'un langage de programmation : conditions, boucles itératives, etc.

Vu l'intérêt que nous portons au parallélisme, nous avons préféré décrire les langages de programmation visuelle concernant le parallélisme dans une partie distincte (*cf.* 3.2). Certaines techniques de spécification sont dédiées au traitement parallèle, c'est pourquoi nous ne les trouverons que dans cette partie.

La table II.1 donne la classification des langages de programmation visuelle que nous allons décrire.

3.1 Techniques de spécification de programmes

Nous décrirons dans cette partie les différentes techniques de spécification utilisées dans la programmation visuelle. Les langages de programmation visuelle peuvent se baser sur les graphes de contrôle, sur les graphes flot de données, être complètement visuels, se baser sur l'hyperprogrammation, sur les exemples, sur les formes, ou sur la réalité virtuelle.

3.1.1 Graphes flot de contrôle

Une des premières représentations visuelles de programmes est l'organigramme[2]. Grail (fait par ARPA en 1969) pouvait générer des programmes directement à partir d'organigrammes, mais le contenu des boîtes était de simples instructions en langage machine. Depuis, plusieurs langages basés sur les organigrammes sont apparus. Par exemple FPL (First Programming Language) décrit comme étant particulièrement adapté à l'enseignement de la programmation aux débutants, étant donné qu'il élimine les erreurs syntaxiques [CTB86]. D'autres langages sont IBGE [TB86], un éditeur graphique basé sur les icônes développé pour Macintosh, et OPAL [Har88] qui permet aux médecins d'introduire dans un système expert les

2. L'organigramme est la représentation de flot de contrôle la plus populaire et probablement la plus connue pour assister les utilisateurs dans la programmation graphique [Rae85].

TAB. II.1 - *Classification de langages de programmation visuelle*

Langage	Domaine d'utilisation	Technique de spécification	Abstraction procédurale	Interprété/ Compilé
Langages non parallèles				
Cantata	Programm. générale	GFD [a]	oui	comp
HyperPascal	Programm. générale	Hyper prog. [b]	oui	comp
Peridot	Dévelop. d'interfaces	PPE [c]	oui	interp
SUNPICT	Programm. générale	GFC (organig.) [d]	non	comp
VIPR	Programm. générale	Compl. visuel [e]	non	comp
Langages parallèles				
ALEX	Alg. de manip. de mat.	PPE + matrices	oui	interp
Cube	Programm. logique	Réal. virt. [f]	oui	comp
Forms/3	Alg. de manip de mat.	PBF [g]	oui	comp
NL	Programm. générale	GFD	oui	comp
Pigsty	Multi-tâches	GFC	oui	interp

[a] Graphe flot de données
[b] Hyper programmation
[c] Programmation par les exemples (démonstrationnel)
[d] Graphe flot de contrôle (organigrammes)
[e] Complètement visuel
[f] Réalité virtuelle
[g] Programmation basée sur les formes

connaissances concernant les traitements de cancer. Ces langages sont des langages compilés. Il en existe d'autres qui sont interprétés, par exemple Pict [GT84] qui a la particularité d'utiliser des images couleurs (icônes) au lieu du texte à l'intérieur des boîtes de l'organigramme. Pict a été amélioré ensuite en un langage compilé SUNPICT que nous décrirons ci-après.

D'autres systèmes ont exploité des représentations de flot de contrôle différentes : diagrammes d'états, réseaux de Petri ou réseaux de Nassi-Shneiderman. Le langage VERDI [SRGB90] par exemple utilise les réseaux de Petri pour spécifier les systèmes distribués ; c'est un langage interprété. Le langage compilé MOPS-2 [AYWC86] utilise les réseaux de Petri colorés. Il permet de construire et de simuler de manière visuelle les systèmes parallèles. Pigsty [Pon86] par contre utilise les organigrammes de Nassi-Shneiderman. C'est un langage interprété ; il est orienté traitement multi-tâches. Nous reviendrons plus en détails à ce langage lors de la description des langages concernant le parallélisme (*cf.* 3.2)

Le langage SunPICT

SunPICT	
Parallélisme :	non
Domaine d'application :	usage général
Technique de spécification :	graphe flot de contrôle (organigramme)
Abstraction procédurale :	non
Interprété/Compilé :	compilé

L'environnement SunPICT [GKM90, GM89, HIY+87] supporte un langage de programmation icônique basé sur des représentations de graphes flot de contrôle (organigrammes). Il permet de développer des programmes généraux. Une des trois parties de l'interface est dédiée à l'édition graphique des programmes. C'est une grille de petits carreaux ; chacun peut contenir un icône ou un pipe pour contrôler le flot.

Les programmes sont construits en plaçant des icônes dans cet éditeur. Il y a quatre types d'icônes prédéfinis dans SunPICT. Les icônes *control/function* contrôlent le flot du programme et manipulent les entrées/sorties. Les icônes START et STOP délimitent le programme. Les icônes WHILE-START et WHILE-STOP délimitent les boucles tant_que. Il existe des icônes utilisés pour les entrées/sorties des variables, et des icônes pour le passage de paramètres aux sous-programmes. Les icônes *Math* réalisent des opérations mathématiques telles que addition et multiplication. Les icônes *conditional* divisent le flot de contrôle en deux chemins possibles : le chemin vert « Vrai » est suivi si l'expression booléenne à l'intérieur de l'icône s'évalue à vrai, alors que le chemin rouge est suivi si l'expression est fausse. Les icônes *pipe* rouge et vert sont utilisés pour indiquer la direction du flot de contrôle. Ceci est juste un exemplaire de l'utilisation des couleurs à travers SunPICT qui aident à mettre en valeur la signification des constructions graphiques.

Deux autres type d'icônes (non prédéfinis comme les quatre types précédents) peuvent être inclus dans un programme SunPICT. Les icônes *name* spécifiant les noms d'autres programmes peuvent être placés pour indiquer des appels de sous routines. Les appels récursifs sont indiqués en plaçant l'icône « nom du programme » dans son propre code SunPICT. Les paramètres sont passés en plaçant un bloc-paramètre tout de suite après l'icône de subroutine. Des icônes *user-defined* sont construits dans le sous-système de définition d'icônes. Ils peuvent être ensuite utilisés dans n'importe quel programme. Dans le sous-système de définition d'icônes, le programmeur dessine les graphiques du nouvel icône, place des emplacements de variables vides, définit la sémantique de l'icône en utilisant un langage textuel spécial, et enfin sauvegarde la nouvelle définition.

Le Icon Action Language (IAL) utilisé dans le sous-système de définition d'icônes est un langage Lisp-like créé pour simplifier la programmation des actions des icônes SunPICT. Tous les icônes, les prédéfinis inclus, incorporent un code IAL déterminant leur sémantique.

Lorsqu'un icône est exécuté, les emplacements des variables (formelles) dans le code IAL sont remplacés par les noms textuels des variables effectives puis le code est passé à un interpréteur.

Bien que les variables aient des noms textuels dans SUNPICT, elles sont décrites visuellement dans un icône par de petits carrés de couleurs qui indiquent leur type (entier, réel ou chaîne de caractères). Les couleurs représentant chaque type ont des valeurs par défaut, mais peuvent être changées à l'exécution par l'utilisateur.

Discussion

Les graphes flot de contrôle de manière générale souffrent du manque des représentations des structures de contrôle de haut niveau ou des structures de données. Les langages construits autour des graphes de contrôle n'offrent pas d'abstraction procédurale. Les concepteurs de langages tentent de plus en plus de se tourner vers de nouvelles représentations visuelles, et ceux qui continuent à utiliser les graphes de contrôle ne s'investissent pas dans la recherche de solutions au problème d'abstraction (comme dans VERDI [SRGB90]) bien qu'ils sachent que ce problème limite sensiblement la taille des programmes à développer. Ces problèmes nous les retrouvons en particulier dans le langage SUNPICT.

Un programme SUNPICT construit par l'utilisateur peut être long. Il peut facilement dépasser la taille de l'éditeur. Augmenter cette taille permet de pallier à ce problème, mais ce n'est certes pas convivial pour une telle interface. Les programmeurs préfèrent plutôt cacher certains détails non nécessaires à un moment donné de la programmation (par exemple : le sinon d'une expression booléenne), mais ce n'est pas possible dans cet éditeur, à moins qu'ils reconstruisent des sous-programmes. Or ceci poserait le même problème dans l'autre sens, c'est à dire vouloir revoir les détails d'un sous-programme.

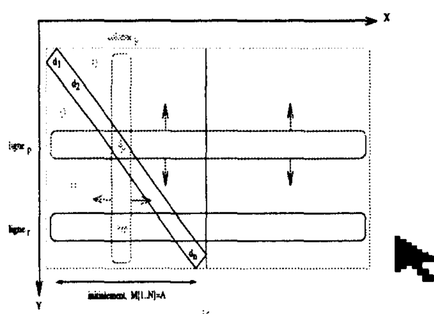
Pour remédier à tout cela, nous aurions préféré qu'il y ait des icônes « hiérarchiques », des icônes pouvant contenir d'autres icônes. Le programmeur pourrait à ce moment là encapsuler une partie du programme dans un seul icône. Au besoin, il ouvre cet icône pour voir son contenu.

Or, dans SUNPICT, tous les icônes ont le même niveau. Le programmeur peut facilement avoir un icône mathématique « Addition » juste à côté d'un icône conditionnel. Pourtant ce dernier est d'un niveau d'abstraction différent de celui du premier icône. Construire une expression moyennement longue, avec ce principe, ne ferait qu'encombrer le programme et surtout gêner la conception (c'est comparable à un programme écrit textuellement mais sans indentation). C'est un problème d'organisation et donc de convivialité. Nous aurions préféré que les expressions soient développées dans un autre éditeur. Au moment de l'utilisation d'une expression, le programmeur ramène juste un icône représentant cette expression et l'insère

dans le programme à l'emplacement souhaité.

SunPICT offre la possibilité de définir de nouveaux icônes-utilisateurs, mais il faut décrire la sémantique par l'intermédiaire du langage IAL. Or ceci renvoie le programmeur à travailler avec un langage textuel. Il aurait été préférable d'aller plus loin dans le développement afin d'offrir une interface visuelle pour la sémantique : le langage IAL sous forme visuelle.

SunPICT est un langage de programmation visuelle à usage général, mais nous ne savons pas comment sont gérés les types composés, par exemple les tableaux.



Vers un data-parallélisme visuel

Nous retenons[3] l'utilisation d'une part des icônes comme symbole graphique représentant un concept du langage (par exemple une construction de contrôle) et d'autre part les couleurs qui permettent de distinguer plusieurs alternatives au choix. Ceci n'est cependant pas d'un grand intérêt lorsque ce n'est pas accompagné d'une abstraction procédurale, c'est ce qui manque par exemple à l'environnement que nous avons étudié SunPICT.

3.1.2 Graphes flot de données

L'idée d'utiliser un graphe flot de données pour représenter un programme existe depuis longtemps [Den75]. Cependant, l'utilisation large du modèle de traitement à flot de données dans les langages de programmation visuelles est plus récente [AA82]. Le flot de données est une approche intrinsèquement visuelle, où un programme est décrit par un graphe orienté. Chaque nœud représente un opérateur (ou une fonction) et chaque arc orienté représente un chemin à travers lequel des données circulent [DK82]. Dès que toutes les données en entrée d'un nœud sont disponibles, celui-ci est exécuté. Le résultat est placé sur les arcs sortants du nœud pour être acheminé au prochain nœud.

L'article de Daniel Hils [Hil92] examine 15 langages de programmation visuelle utilisant le modèle à flot de données. Les langages sont : Hookup, Fabrik, InterCons, Hi-visual, Viva,

3. Les points que nous mettrons en évidence ici sont relatifs à des langages séquentiels, mais peuvent être réutilisés pour des langages ou environnements dédiés au parallélisme.

Cantata, VIPEX, LabView, ConMan, *vis*, Visual ToolSet, PROGRAPH, Show and Tell, ESTL, et DataVis. Les langages sont classifiés par leur domaine d'application :

Hookup concerne la musique; Fabrik et InterCONS sont dédiés à la construction d'interfaces utilisateur; HI-VISUAL, VIVA, Cantata sont destinés au traitement d'images; VIPEX et LabView traitent de la science; ConMan est conçu pour le graphisme; *vis*, Visual ToolSet, PROGRAPH, et Show and Tell sont des langages à programmation générale; et DataVis s'intéresse à la visualisation scientifique.

Nous avons choisi de décrire le langage Cantata développé par J.R. Rasure et C.S. Williams en 1990. Il permet en particulier le développement d'algorithmes de manipulation de matrices.

Le langage Cantata

Cantata	
Parallélisme :	non
Domaine d'application :	usage générale
Technique de spécification :	graphe flot de données
Abstraction procédurale :	oui
Interprété/Compilé :	compilé

Cantata est un langage de programmation visuelle, conçu à l'origine pour le traitement d'images [RASW90, WR90]. Il a été élargi ensuite pour être plus général [RW91]. Il a montré son utilité dans plusieurs domaines, en particulier en traitement de signal, comme langage d'interrogation de bases de données, pour le développement d'algorithmes de manipulation de matrices et pour les systèmes de contrôle [Hil92]. Il inclut plusieurs bibliothèques, par exemple 240 algorithmes pour les traitements de signal et d'images.

Cantata se base particulièrement sur le modèle de programmation à flot de données. Les interactions avec l'utilisateur sont contrôlées par éditeur graphique opérant à travers une hiérarchie d'espaces de travail pour construire un graphe flot de données. L'accès au langage visuel se fait par l'intermédiaire des objets graphiques de l'interface, appelés « glyphs » dans Cantata. la figure II.1 montre un espace de travail Cantata contenant un exemple d'application de traitement d'images.

À l'inverse des langages de programmation visuelle à grain fin comme SUNPICT où tous les détails sont obligatoirement spécifiés de manière visuelle, Cantata est un langage à gros grain. Les expressions mathématiques par exemple sont introduites textuellement dans un langage traditionnel C ou Fortran.

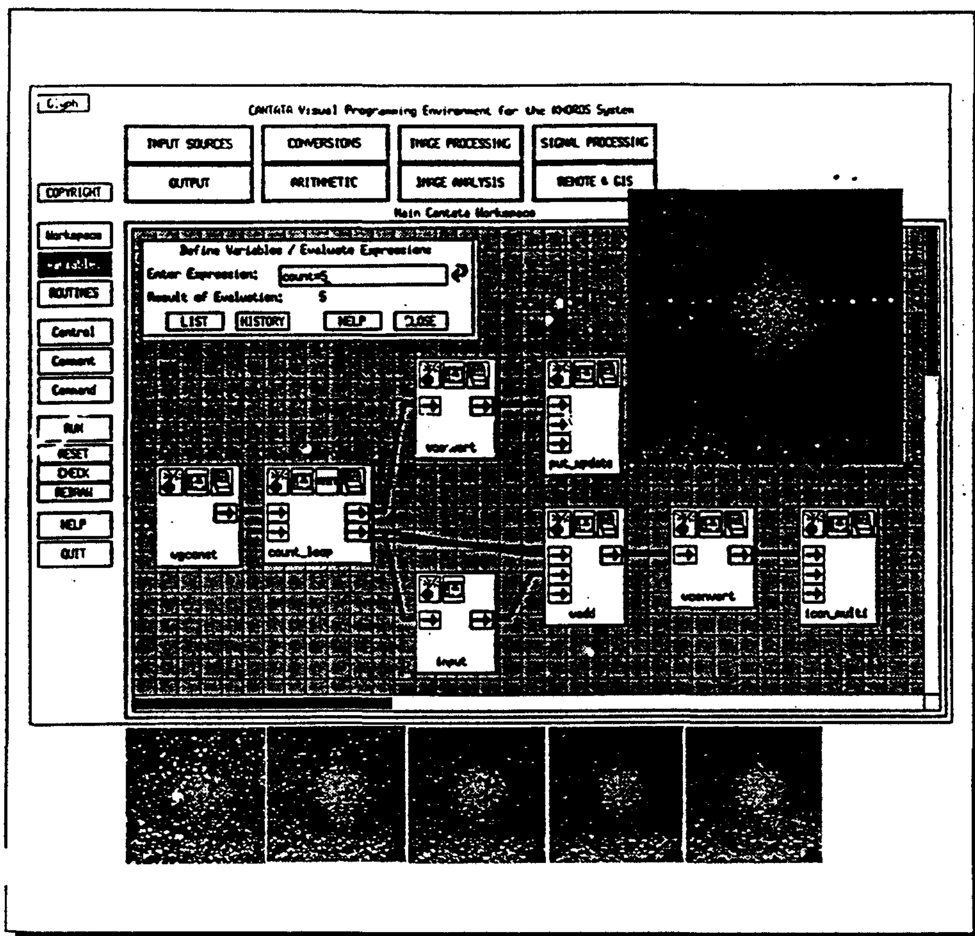


FIG. II.1 - *Cantata : un espace de travail contenant un exemple d'application de traitement d'images*

La syntaxe de Cantata se base sur les « glyphs », représentant les nœuds du graphe. Ce sont des icônes ayant chacun une sémantique. Les « glyphs » sont de plusieurs types, nous les partageons en deux classes : des glyphs simples (spécifiant des entrées/sorties, des constructions de contrôle, etc.) et des glyphs composés pouvant contenir un sous graphe (procédure ou fonction).

Deux formes d'itération sont possibles. Les deux sont des constructions de contrôle de flot : COUNT-LOOP (similaire à une boucle FOR, cf. figure II.1), et WHILE-LOOP. Chacune de ces boucles est représentée par un icône placé dans le graphe flot de données. La première entrée de l'icône représente l'initialisation de la boucle, la seconde représente le retour du flot de données pour la prochaine itération. Pour chaque itération, le flot de données est acheminé à travers une des deux sorties de l'icône. Une fois la condition de boucle satisfaite, le flot passe à travers la deuxième sortie de l'icône de boucle.

Une construction If_Then_Else contrôle le flot de données, plutôt que le flot de contrôle.

Une expression conditionnelle est évaluée pour diriger les données à travers un des deux chemins possibles. La figure II.2 donne un exemple de If_Then_Else. Cette figure montre aussi l'utilisation des espaces de travail hiérarchiques ; c'est l'abstraction de procédure permettant de mieux gérer la complexité visuelle. La figure contient une procédure utilisateur « MAR-Filter ». La procédure a été créée et sauvée, puis utilisée deux fois (avec des paramètres différents) dans le même espace de travail du programme. Elle apparaît dans les deux branches après l'icône If_Then_Else. Après le « Then », elle est encapsulée dans un glyph, alors que celle du « Else » est ouverte pour montrer l'intérieur du glyph et les différentes connections.

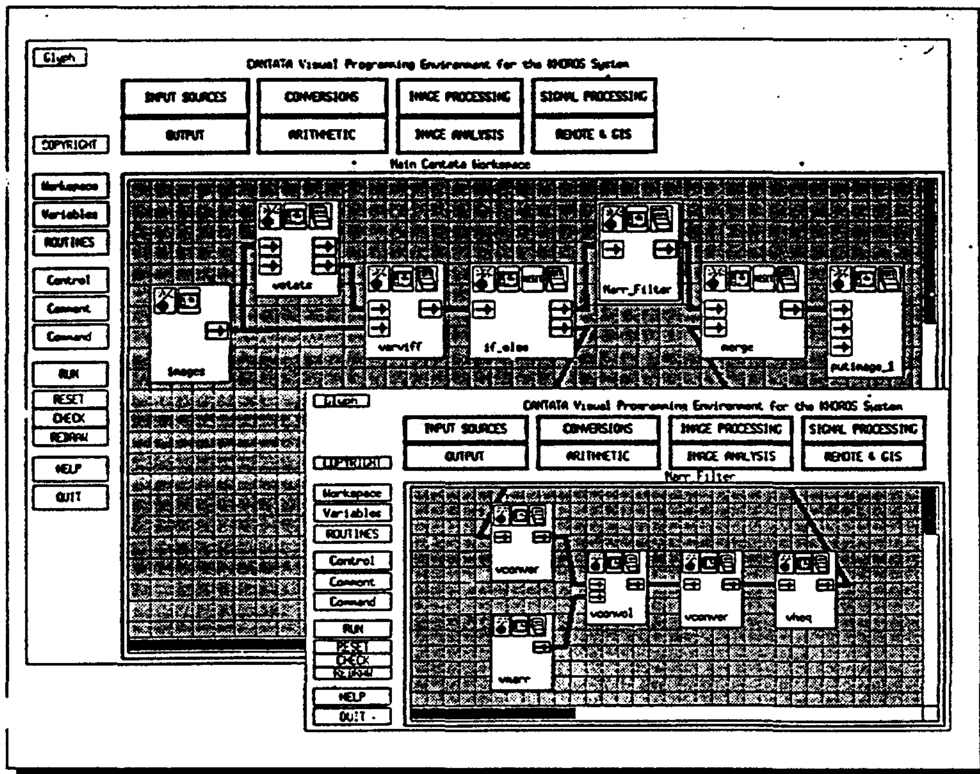


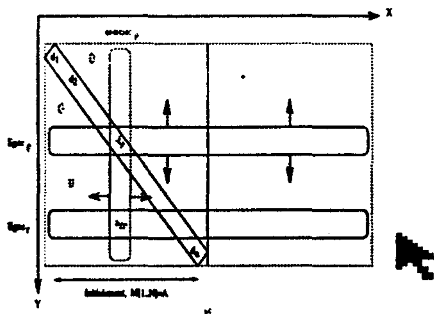
FIG. II.2 - *Cantata : un exemple de If_Then_Else et utilisation des espaces de travail hiérarchiques*

Discussion

Les graphes flot de données, comme les graphes de contrôle, supportent mal les structures de données non triviales ou les structures de contrôle de haut niveau, menant ainsi à des diagrammes confus [Rae85]. Certains travaux de recherche tentent de trouver les bonnes constructions de contrôle à mettre dans les langages basés sur les graphes flot de données [Hil92]. Ces travaux doivent tenir compte des domaines d'application et des utilisateurs finaux.

Les langages basés sur les graphes flot de données ne libèrent pas l'utilisateur de toute

syntaxe, il reste toujours des parties textuelles. En Cantata par exemple les expressions, les formules mathématiques et les algorithmes numériques sont introduits de manière textuelle utilisant les langages conventionnels C ou Fortran. C'est un aspect important dont l'intérêt a été souligné auparavant, mais nous aurions préféré que soit rajoutée l'option visuelle. Cantata aurait pu laisser la possibilité aux utilisateurs de développer des expressions dans un autre éditeur graphique d'expressions. L'utilisateur n'aurait eu à connaître aucune syntaxe textuelle. L'expression, une fois réalisée, sera insérée là où il faut dans le glyph correspondant.



Vers un data-parallélisme visuel

Nous retenons, à la suite de la description de Cantata, l'utilisation de la hiérarchie des espaces de travail permettant d'offrir une abstraction procédurale assez conviviale. Nous notons également l'utilisation des bibliothèques de fonctions ou d'algorithmes prédéfinis couvrant des domaines spécifiques tel que le traitement d'images. Ceci est très important car de toute évidence les utilisateurs préfèrent travailler dans un langage ou un environnement où ils font le moins possible de développement.

3.1.3 Langages complètement visuels

Cette technique de spécification a été proposée par W.C Citrin et al. [CDZ93]. Un langage complètement visuel ne signifie pas un langage de programmation qui ne contient pas de texte. Pour certains aspects du programme, il est clair que le texte peut être plus approprié. L'avantage d'un tel langage est que l'utilisateur n'a pas besoin de comprendre deux sémantiques différentes : la sémantique du modèle visuel et celle du modèle textuel. Il peut comprendre et écrire des programmes en connaissant seulement la sémantique visuelle.

En se basant sur cette propriété, W.C Citrin et al. [CDZ93] de l'université du Colorado ont développé le langage VIPR.

Le langage VIPR

VIPR	
Parallélisme :	non
Domaine d'application :	usage générale
Technique de spécification :	Complètement visuel
Abstraction procédurale :	non
Interprété/Compilé :	compilé

VIPR (Visual Imperative PRogramming language) est un langage complètement visuel. La particularité de VIPR est qu'il est un langage impératif, où on peut définir des variables, des structures de contrôle telles que If_then_else, While, et Goto. VIPR est dédié au développement de programmes généraux.

D'autres langages impératifs existent : PICT [GT84], C^2 [KG90], PECAN [Rei84], mais ils se basent sur des graphes de contrôle et n'ajoutent qu'une couche visuelle au-dessus d'un langage textuel existant, précisément au-dessus d'une sémantique textuelle.

En VIPR, tout est représenté par des cercles. La figure II.3 montre un exemple « Hello World ». Il y a des cercles pour l'affectation et pour l'instruction « printf ». Une exécution séquentielle est indiquée par des cercles imbriqués les uns dans les autres. Ainsi, puisque le cercle de l'instruction « printf » est à l'intérieur du cercle de l'affectation, la sémantique est telle que le « printf » s'exécute après l'affectation.

L'exemple montre que les conditions sont indiquées par l'inclusion de deux cercles à l'intérieur d'un même cercle. Chacun de ces cercles indique une branche possible et doit être gardé par un test. Celui-ci est indiqué textuellement à gauche du cercle. La première action d'une branche est écrite à droite du cercle, la suivante est représentée à l'intérieur du cercle par un autre cercle. Dans cet exemple, c'est un If_then_else qui est représenté, mais en réalité l'utilisateur peut inclure de la même façon plusieurs cercles pour représenter plusieurs alternatives : un « Switch ». L'action et le test sont tous les deux optionnels. Si le test est omis, il sera considéré à « vrai ». Si l'action est omise alors une instruction vide « null » est considérée.

VIPR offre aussi le branchement inconditionnel « Goto », cf. figure II.4. C'est une flèche qui va du cercle courant vers le cercle où doit se faire le branchement. Avec la flèche et le test conditionnel, l'utilisateur peut réaliser une boucle itérative ; la flèche pointera vers un cercle englobant. La figure II.5 montre un exemple d'appel de procédure et le retour « return » qui se fait par l'intermédiaire des flèches et des petits cercles au bout. L'utilisateur peut de cette façon réaliser aussi des appels récursifs.

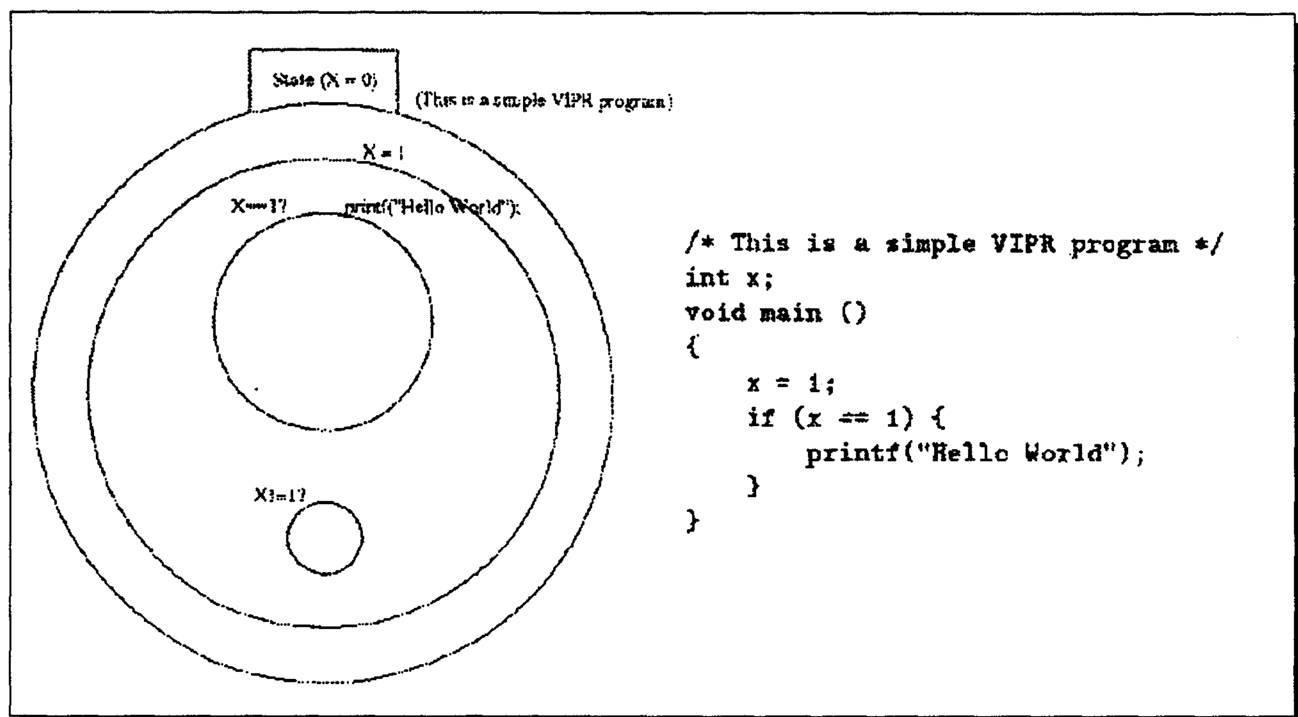


FIG. II.3 - *Un exemple de programme* VIPR « *Hello World* » *(à gauche) et son équivalent en C (à droite)*

Discussion

Cette technique de spécification propose deux choses : la première est l'idée d'avoir une sémantique visuelle originale indépendante de toute sémantique textuelle et la deuxième concerne les nouvelles formes de représentation visuelle. Or pour la première, rien ne justifie l'intérêt d'écarter les sémantiques textuelles ; la sémantique visuelle peut se baser sur une sémantique textuelle sans en faire référence explicitement. Le seul langage que nous connaissons autour de cette idée est VIPR, et il n'y a pas d'expérience pouvant justifier l'idée proposée par W.C. Citrin. Il est vrai par contre que l'utilisateur ne doit pas être obligé de connaître deux sémantiques différentes : textuelle et visuelle. Et c'est là l'intérêt de la deuxième proposition : trouver de nouvelles formes de représentation visuelle. En effet, les efforts doivent être dirigés dans ce sens et dans l'intention de libérer le plus possible l'utilisateur de toute syntaxe textuelle. Les langages à proposer doivent être en outre très proches du raisonnement humain, qu'ils prennent en compte l'abstraction des structures de données délaissée jusqu'ici [BA94], et surtout l'abstraction procédurale. Ceci concerne la technique de spécification elle-même, nous avons relevé également d'autres remarques sur le langage VIPR ; nous en présentons quelques unes.

L'interface graphique de VIPR n'est pas encore réalisée. La dernière version [CDZ94] est

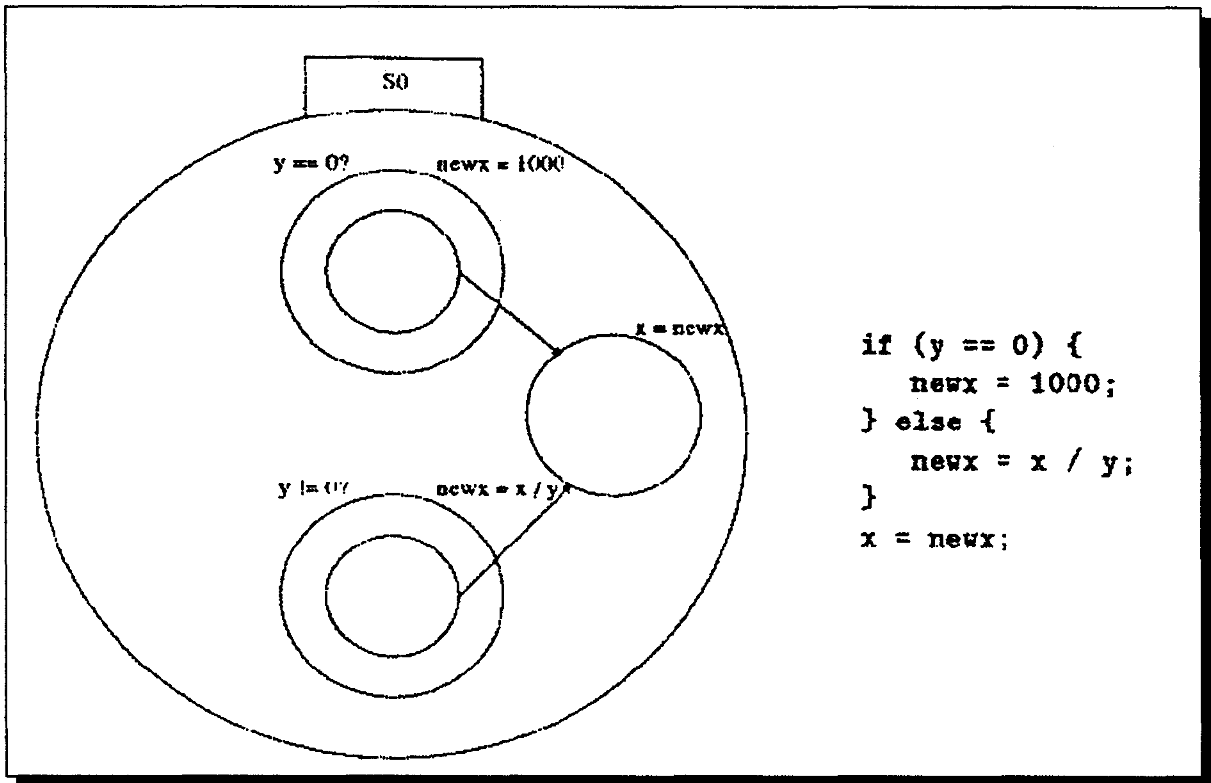


FIG. II.4 - VIPR : *représentation d'une instruction conditionnelle*

orientée objet, mais ne change pas la syntaxe visuelle que nous avons décrite. Il nous semble un peu difficile de réaliser graphiquement (par exemple sous X11) la manipulation des cercles, d'autant plus que souvent l'utilisateur conçoit ses programmes de manière descendante. Il ne connaît pas au préalable le nombre de cercles imbriqués. Si la taille des cercles englobants est modifiée à chaque fois que l'utilisateur veut insérer un cercle, cela gênera sans doute sa conception. Encore plus lorsqu'il doit faire des va-et-vient entre les différents niveaux de cercles pour y insérer des nouveaux. À noter malheureusement qu'il n'y a aucune abstraction procédurale.

Dans l'exemple de la figure II.4, le grand cercle inclut 3 autres cercles : deux correspondent au test, et le troisième représente une instruction en dehors du test. Il nous semble impossible de savoir dans certains cas que le troisième cercle représente une des alternatives d'un « switch » avec un test conditionnel à vrai (par exemple un switch avec trois cas et dans 2 cas il y a un « Goto » vers le troisième).

A chaque fois que les flèches ou les arcs font partie de la représentation visuelle, le phénomène « spaghetti » réapparaît. En VIPR, il suffit d'avoir des « If » ou « While » imbriqués, pour le constater.

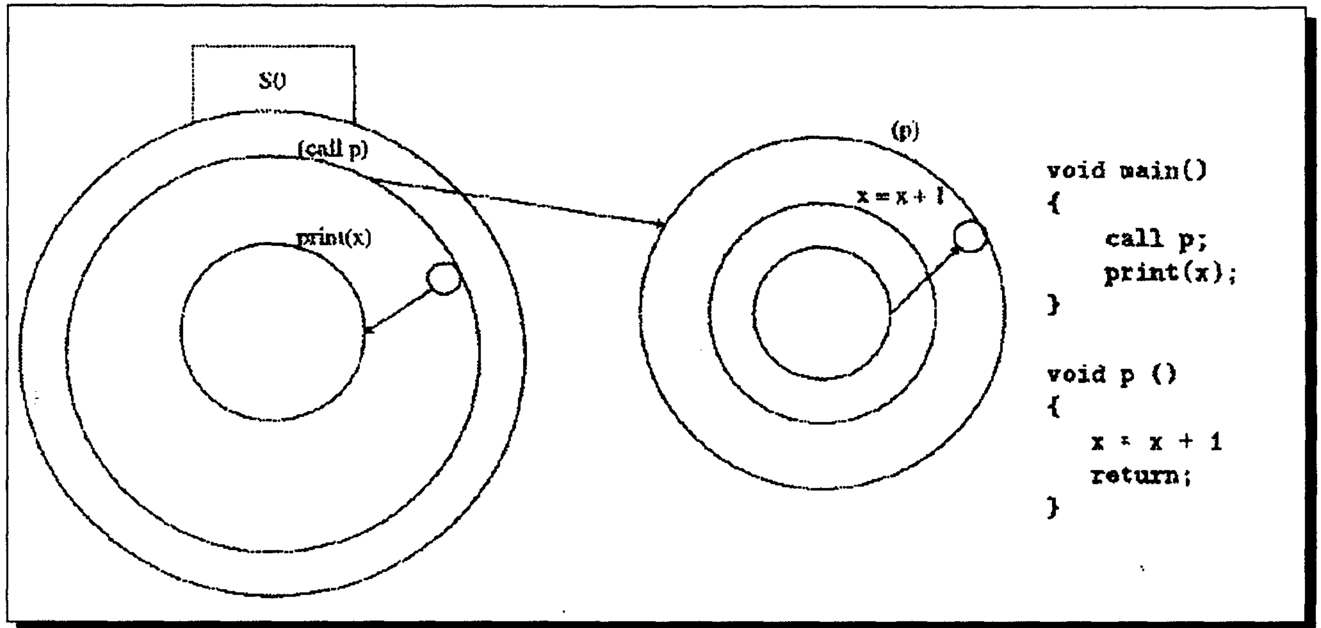
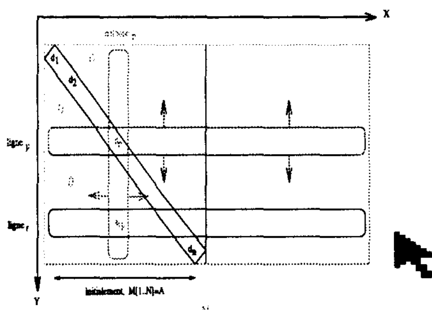


FIG. II.5 - VIPR : *Appel de procédure et retour*



Vers un data-parallélisme visuel

Nous retenons ici le principe essentiel d'un langage complètement visuel, c'est à dire de concevoir un langage de programmation visuelle dont la sémantique est le plus indépendant possible de toute sémantique textuelle.

3.1.4 Hyperprogrammation

L'hyperprogrammation est un terme nouveau introduit par P.J. Lyons [LSA93]. Il considère qu'un langage d'hyperprogrammation est un langage permettant d'exprimer un programme à travers différentes représentations visuelles, où chaque vue correspond à une partie ou un niveau d'abstraction du programme. Une vue pourrait représenter par exemple le graphe du programme. Les nœuds correspondent aux noms des fonctions ou procédures du programme. Un autre type de vue représenterait sous un autre aspect la définition même

d'une fonction.

L'hyperprogrammation introduit également les hyperliens pour permettre une navigation facile (dans tous les sens) entre les différentes vues. Ces hyperliens, créés automatiquement lors du développement d'un programme, sont similaires à ceux des hypertextes. Lors de la définition d'une procédure, l'utilisateur pourrait placer un icône correspondant à l'appel d'une fonction préalablement définie. En cliquant deux fois sur l'icône, les hyperliens l'amènent directement à la vue contenant la définition de la fonction à appeler. Un autre type d'hyperlien concernerait par exemple les identificateurs. Si la déclaration et l'utilisation d'un identificateur appartiennent à deux types de vue différents, les hyperliens permettront, lors de l'utilisation d'un identificateur, de retrouver directement la vue où il a été déclaré.

Le langage HyperPascal a été développé par P.J. Lyons et al. [LSA93] pour établir la faisabilité des langages visuels basés sur l'hyperprogrammation.

Le langage HyperPascal

HyperPascal	
Parallélisme :	non
Domaine d'application :	usage général
Technique de spécification :	Hyper programmation
Abstraction procédurale :	oui
Interprété/Compilé :	compilé

La fonctionnalité de HyperPascal est entièrement basée sur le langage Pascal. HyperPascal utilise les types booléen, réel, entier, et chaîne de caractères (string), des sous-programmes procéduraux ou fonctionnels, des enregistrements (record), des paramètres en entrée/sortie (var), etc. Dans l'environnement de programmation, à chaque type de donnée est associée une couleur pour mieux le distinguer.

La représentation visuelle d'un programme HyperPascal est divisée en trois types de vues. Les instructions et les structures de contrôle sont représentées dans des vues contenant des arbres d'actions (Action Tree Vues) ; chaque vue équivaut à une procédure ou fonction Pascal. Chaque arbre d'actions est (sous forme d'icône) un nœud dans la seconde vue : arbre de portée (Scope Tree Vues) qui a la responsabilité de contenir les déclarations de tous les identificateurs utilisés dans le programme. Les fenêtres de formes (Forms Windows) constituent le troisième type de vue. Elles offrent un environnement d'édition WYSIWYG [4] permettant

4. WYSIWYG : « What You See Is What You Get » est un acronyme utilisé surtout pour les traitements de texte et les générateurs d'interfaces pour dire que l'utilisateur aura en sortie « exactement » ce qu'il est entrain de voir lors de l'édition.

au programmeur de spécifier l'apparence des entrées/sorties du programme.

La figure II.6 montre un icône d'un sous-programme, c'est un nœud dans l'arbre de portée. Chaque icône de ce type contient 4 barres, encapsulées à gauche de la figure et étendues à droite. Ces barres représentent les paramètres d'entrée/sortie (à droite et à gauche), les noms des sous-programmes appelants (en haut) et appelés (en bas), et les variables locales (en bas aussi).

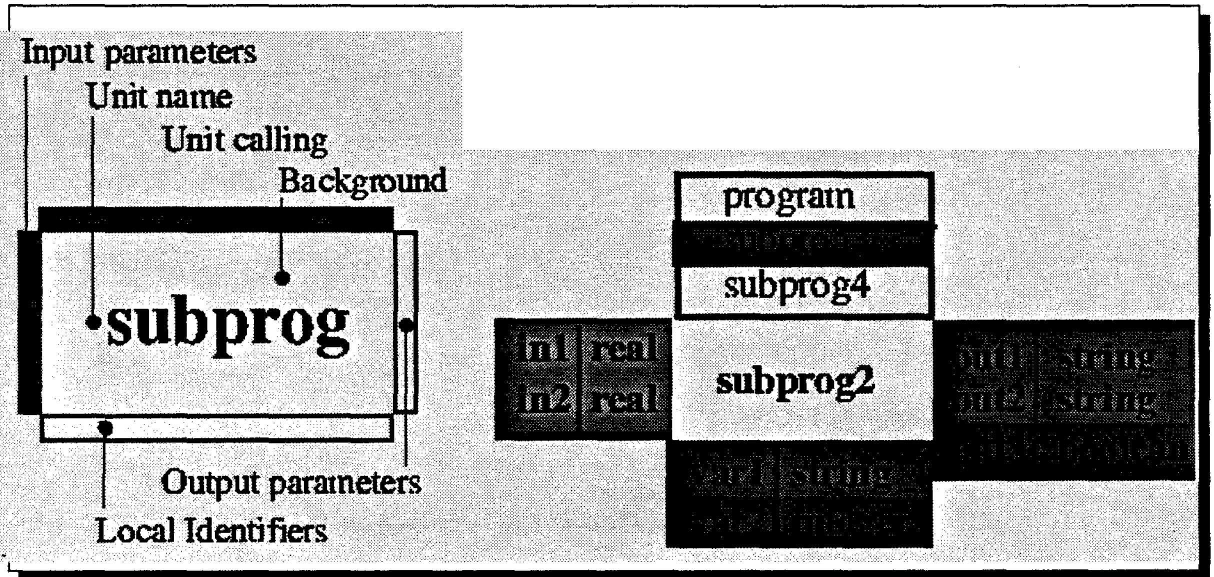


FIG. II.6 - *HyperPascal: un icône de sous-programme*

La figure II.7 montre un exemple d'arbre de portée. C'est un arbre de définition : le sous-programme 4 par exemple, tel que représenté dans la figure, est défini localement dans le sous-programme 2. La couleur de l'icône du sous-programme 3 montre qu'il s'agit d'une fonction retournant une valeur (dans ce cas un réel).

La figure II.8 montre un exemple d'arbre d'actions : un simple algorithme pour déterminer le maximum d'un ensemble d'entiers positifs saisis au fur et à mesure. L'item « a » est l'icône du programme avec les barres qui représentent le paramètre de sortie et les variables locales. L'itème « b » est un commentaire (un rectangle qui n'est pas une feuille est un commentaire). L'itème « c » est un icône de choix (case of) ; le texte qu'il contient est un commentaire, et la condition se trouve dans le sous-icône circulaire (ici étendu pour montrer la condition). L'itème « d1 » est un icône de boucle contenant lui aussi un commentaire. Les boîtes de la flèche gauche (pointant vers le bas) peuvent incorporer un code à exécuter avant d'entrer dans la boucle. La flèche adjacente (pointant vers le haut) a une fonction similaire ; elle peut contenir un code à exécuter à la sortie de la boucle. Le sous-icône triangulaire (pointant à droite) correspond à la condition de continuité, étendue dans l'itème « d2 ». Si la condition est évaluée à vrai, les instructions (en-dessous) s'exécuteront, sinon (rencontre d'un entier nul)

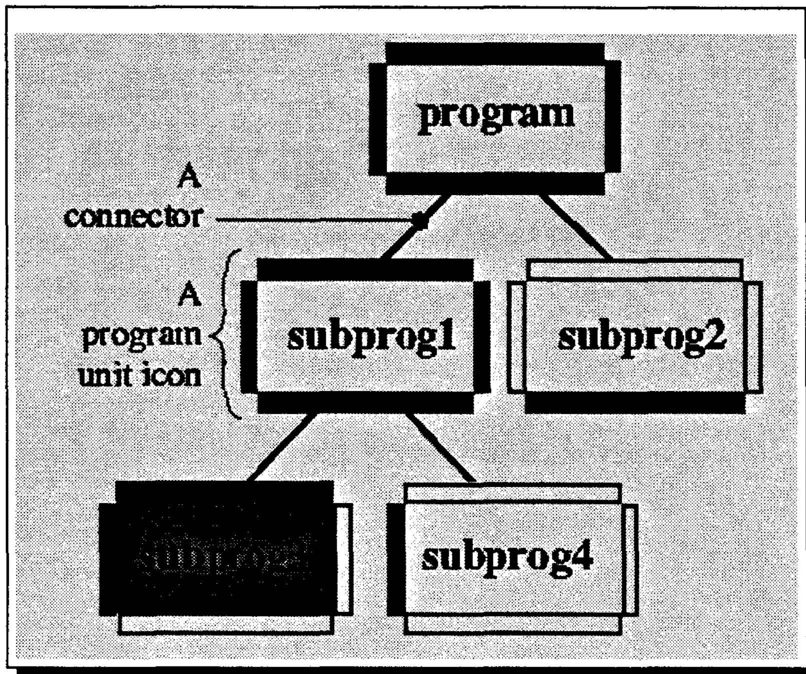


FIG. II.7 - *HyperPascal: quatre icônes de sous-programmes assemblés dans un arbre de portée*

la boucle s'arrêtera.

Ces vues sont reliées entre elles par des hyperliens. Ouvrir un identificateur (en double-cliquant sur son icône) mène l'utilisateur à sa déclaration dans l'arbre de portée. Ceci est possible parce que les arbres d'actions, bien que représentés dans des fenêtres indépendantes, sont conceptuellement des entrées dans l'arbre de portée. L'utilisateur trouve un autre intérêt dans les hyperliens qui concerne cette fois-ci les dépendances entre les sous-programmes. L'environnement de programmation maintient, au niveau de l'arbre de portée et de l'arbre d'actions de chaque sous-programme, la liste des autres sous-programmes appelant ce sous-programme ou appelés par ce sous-programme. Enfin, il y a des hyperliens entre les instructions d'entrée/sortie dans l'arbre d'actions et les fenêtres de formes associées. Ils permettent au programmeur de se déplacer de l'arbre d'actions à la fenêtre des formes associée afin de formater l'affichage de la variable, et de revenir ensuite à l'arbre d'actions.

Discussion

L'hyperprogrammation introduit deux techniques importantes et utiles : les vues et les hyperliens. Les différents types de vues permettent de faire abstraction de plusieurs niveaux d'un programme et d'économiser par conséquent l'espace de travail qui est un obstacle dans la programmation visuelle. En HyperPascal, les concepteurs ont choisi trois types de vues. Dans la première « arbre de portée », nous aurions préféré que soit rajoutée l'option branche

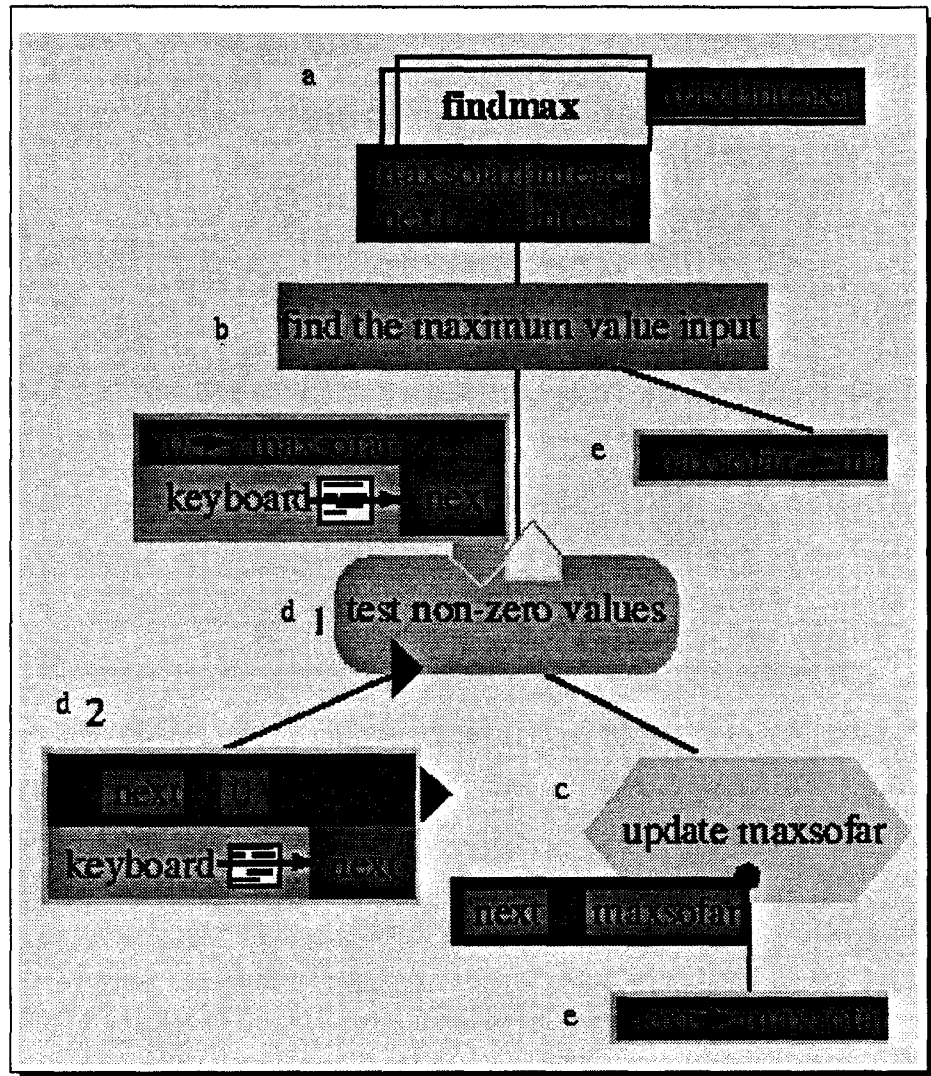


FIG. II.8 - *HyperPascal: exemple de représentation d'un arbre d'actions*

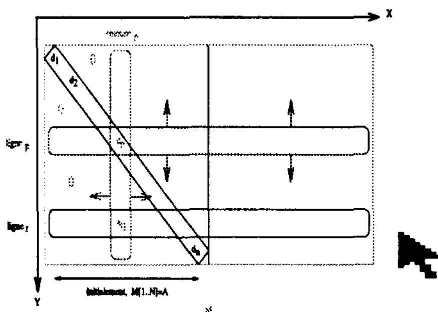
(par rapport à tout l'arbre). En effet l'expérience avec le gestionnaire de fichiers « filemgr » de OpenWin[5], nous a montré que les utilisateurs choisissent très rarement l'option arbre pour voir leurs répertoires et fichiers. Ils visualisent plutôt la branche qui lie le répertoire courant vers la racine. Lors de la définition par exemple du sous-programme 3 de la figure II.7, l'utilisateur n'a pas besoin de voir les détails des autres branches sous-programme 2 et sous-programme 4. En ce qui concerne l'arbre d'actions, nous trouvons que la structuration visuelle manque beaucoup d'ergonomie et de convivialité que ce soit au niveau global d'un sous-programme ou au niveau des constructions de contrôle telles que la boucle. HyperPascal offre dans le troisième type de vue la possibilité de formater les entrées/sorties. C'est un moyen visuel qui n'est malheureusement pas pris en compte par beaucoup de langages de

5. OpenWin est un gestionnaire de fenêtres « Window manager » utilisé par les stations de travail SUN. OpenWin est une marque déposée de OpenWin.

programmation visuelle, alors qu'il est très utile aux programmeurs.

La deuxième technique importante introduite par l'hyperprogrammation sont est la programmation par les hyperliens. Ils sont très utiles et offrent une grande convivialité à la programmation visuelle. Cet aspect a été suffisamment démontré avec les hypertextes.

Enfin il reste à valider l'hyperprogrammation avec d'autres langages d'autant plus que l'objectif est de définir un langage de programmation visuelle basé sur cette technique de spécification mais indépendant d'une sémantique textuelle particulière.



Vers un data-parallélisme visuel

Nous retenons d'une part la représentation graphique des différents niveaux d'un programme et d'autre part l'intérêt essentiel des hyperliens qui permettent une navigation très conviviale entre les différents niveaux d'un programme.

3.1.5 Programmation basée sur les exemples

De nombreux systèmes de programmation visuelle utilisent aussi la « programmation basée sur les exemples » — Example-Based Programming. Ces systèmes[6] donnent la possibilité à l'utilisateur de réaliser des actions sur des exemples d'objets concrets (souvent par manipulation directe), et construisent en même temps un programme abstrait [Mye92]. Il y a deux types de Programmation Basée sur les Exemples (PBE) : « Programmation Par les Exemples » et « Programmation Avec les Exemples » [Mye90b]. La première se réfère aux systèmes qui essayent de deviner ou *inférer* le programme à partir d'exemples d'entrées/sorties ou de traces d'échantillon d'exécution. Elle s'appelait aussi « programmation automatique » et a été généralement un domaine de la recherche en Intelligence Artificielle ; B. Myers qualifie ce type de systèmes de « Systèmes Intelligents » [Mye92]. Les systèmes de programmation avec les

6. Ces systèmes appartiennent aussi à une classe plus grande « les interfaces démonstrationnelles », où l'on trouve des systèmes de programmation ou autre (tels que des outils de dessins). Le terme démonstrationnel est utilisé car l'utilisateur démontre le résultat désiré à partir d'exemples. [Mye92]

exemples quant à eux exigent du programmeur de spécifier tout ce qui concerne le programme (aucune inférence n'intervient), mais le programmeur peut développer le programme sur un exemple spécifique. Le système exécute normalement les commandes utilisateur, mais se souvient d'elles pour une réutilisation postérieure. Halbert [Hal84] caractérise la programmation avec les exemples comme « *Do What I Did* », et la programmation par les exemples comme « *Do What I Mean* ».

La programmation basée sur les exemples est une grande classe, dans ce rapport nous considérons seulement le sous ensemble de langages qualifiés de programmation visuelle, c'est à dire là où les exemples sont introduits de manière graphique.

Des tentatives basées sur cette technique de spécification « programmation basée sur les exemples » ont eu lieu au début pour la mise en œuvre de langages de programmation visuelle compilés, par exemple le Traces que M.A. Bauer en 1978 décrit dans sa thèse [Bau78]. Malheureusement, ces systèmes avaient tendance à créer des programmes incorrects, et il était difficile de vérifier ce qu'avait fait le système sans étudier le code généré [Mye90b].

Il y a eu par contre de nombreux langages interprétés. B. Myers recense, dans ses deux papiers [Mye90b, Mye92], une douzaine de ces systèmes. Un des premiers est Pygmalion en 1977 [Mye90b] ; il utilise dans sa méthode de programmation les icônes[7] et il ne fait intervenir aucune inférence (programmation avec exemples).

Certains systèmes, en plus de la « programmation basée sur les exemples », utilise en même temps une autre technique de spécification. Fabrik [IWC+88] par exemple utilise les graphes flot de données. Il permet à l'utilisateur de connecter entre elles des primitives de bas niveau comme des opérateurs arithmétiques, ou des éléments (Widget) de niveau plus haut pour constituer des interfaces (exemple : ascenseurs, boutons, etc.). Fabrik se base lui aussi sur la programmation avec exemples. Le système ALEX [KTC+90] utilise les matrices. Il permet de spécifier des algorithmes de manipulation de matrices par l'intermédiaire d'exemples. L'utilisateur pointe sur un élément, une ligne, ou une colonne typique d'une matrice représentée graphiquement, puis spécifie comment se fait le traitement. Le système infère et généralise alors ce traitement pour opérer sur toute la matrice. Cette opération est qualifiée de traitement parallèle, nous reviendrons plus en détails à ce système lors de la description des

7. La première utilisation des icônes dans les interfaces est également attribuée à Pygmalion.

systèmes dédiés au parallélisme.

Système	Programmation basée sur les exemples		Autre technique
	Avec les exemples	Par les exemples (intelligent)	
Fabrik	⋆		graphes flot de données
ALEX		⋆	Matrices
Peridot		⋆	

Nous avons choisi, pour illustrer la technique de spécification « programmation basée sur les exemples », le système Peridot mis en œuvre par B.A. Myers [Mye90a]. Il utilise la « programmation par les exemples ou démonstrationnelle ». La description d'un tel système est plus riche que celle d'un système utilisant la « programmation avec les exemples ».

Le langage Peridot

Peridot	
Parallélisme :	non
Domaine d'application :	développement d'interfaces
Technique de spécification :	programmation par les exemples
Abstraction procédurale :	oui
Interprété/Compilé :	interprété

Le système Peridot (Programming by Example for Real-time Interface Design Obviating Typing) est dédié au développement des interfaces utilisateurs, en particulier la construction des éléments d'une interface (Widgets). Le concepteur dessine des images pour montrer à quoi l'interface devrait ressembler, puis il utilise la souris et d'autres périphériques d'entrées pour *démontrer* comment l'interface devrait fonctionner. Peridot généralise à partir de ces exemples d'images et d'actions pour créer des procédures paramétrées, comme celles que nous trouvons dans les boîtes à outils telles que XToolkit ou Motif.

En plus de la programmation par les exemples, Peridot utilise une autre technique : *la programmation par contrainte*. Les contraintes sont les relations maintenues automatiquement par le système. Dans Peridot, elles sont utilisées de deux manières différentes. Les *contraintes graphiques* lient divers objets graphiques entre eux de sorte que lorsqu'un objet change, les autres seront automatiquement modifiés (dépendance entre objets). Les *contraintes de données* sont utilisées pour assurer que des objets graphiques soient modifiés à chaque fois que des variables spéciales, appelées valeurs actives, sont modifiées (objets dépendants de certaines variables actives).

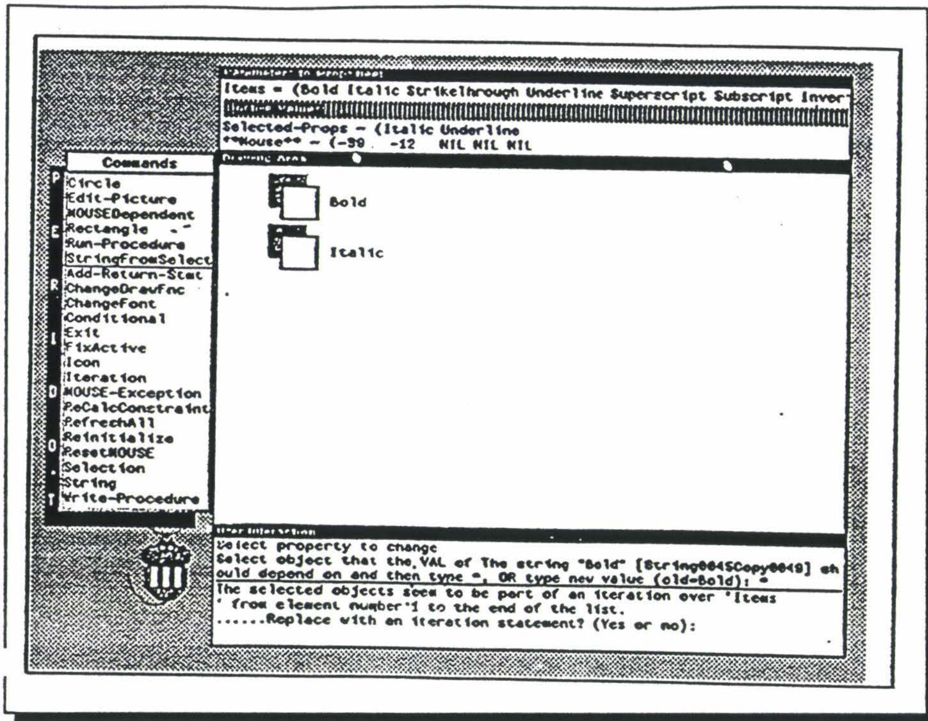


FIG. II.9 - *L'interface Peridot lors du développement d'un exemple de procédure*

La figure II.9 montre une configuration typique lors du développement d'une procédure sous Peridot : création d'un ensemble de boutons de contrôle. Avant cet écran, l'utilisateur a déjà introduit :

- le nom de la procédure : « PropSheet » ;
- les paramètres de la procédure, ici un seul : « items » ;
- des valeurs-exemples pour le paramètre, ici une liste : « (Bold Italic StrikeThrough Underline Superscript Subscript Inverted) » ;
- les valeurs-exemples pour la valeur active, ici une liste : « (Bold Underline) ».

Toutes ces données sont montrées dans les deux fenêtres en haut de la figure. Le menu à gauche offre les outils de dessin et les différentes fonctions que peut appliquer l'utilisateur. Au fur et à mesure que l'utilisateur dessine les graphiques (dans la fenêtre du milieu), Peridot établit à la suite d'inférences les relations entre les différents objets. C'est ainsi qu'il crée par exemple une contrainte graphique « la même taille » entre le carré noir du fond et le carré blanc au-dessus. Une contrainte de données pourrait être par exemple dans un autre contexte, la position d'un ascenseur par rapport à la taille d'un fichier visualisé. Sachant que les inférences peuvent être incorrectes, le système demande à chaque fois, dans la fenêtre

en bas de la figure, la confirmation de l'utilisateur. A la suite des deux éléments dessinés et ayant établi les relations entre ces éléments et la liste du paramètre « items », Peridot crée automatiquement le reste des itèmes, cf. figure II.10. Le système infère à partir de deux exemples le besoin d'une boucle.

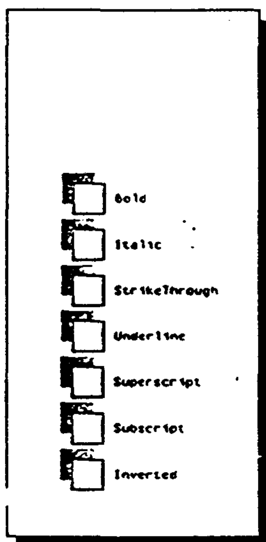


FIG. II.10 - *Peridot: un exemple d'inférence puis génération d'une liste d'itèmes*

Peridot permet de spécifier aussi des conditions par l'intermédiaire des « objets conditionnels ». Ces conditions sont différentes des constructions IF dans les langages conventionnels, car elles n'affectent pas le flot de contrôle. Elles sont appelées conditions parce qu'elles rendent l'apparence d'un objet dépendante d'une variable de contrôle. Les conditions sont créées de façon postfixée; l'utilisateur dessine en premier les objets graphiques (exemple: une marque de contrôle) à utiliser comme retour « feedback » lorsque la condition est vraie, puis spécifie de quoi ces objets dépendent: d'une valeur active ou d'un paramètre de la procédure.

Dans la figure II.11(a), l'utilisateur place une marque de contrôle centrée sur un des boutons. Cette marque est utilisée pour montrer quels sont les itèmes sélectionnés. Afin de *démontrer* que la sélection se fait par la souris, l'utilisateur utilise l'icône de la « souris simulée » (figure II.11(b)). Il place le nez de la souris sur la marque en pressant le bouton milieu de la souris. Comme il n'y a qu'une seule valeur active « Selected-Props » (cf. figure II.9), Peridot infère que la marque dépend de cette valeur. Mais puisque la valeur-exemple de la valeur active est une liste, le système infère que plusieurs itèmes sont concernés et qu'une marque doit apparaître pour chacun dans la liste. Une fois que l'utilisateur confirme ces inférences, Peridot visualise deux marques pour « Italic » et « Underline ». Enfin, le système demande à l'utilisateur si le bouton milieu de la souris devrait alterner « toggle », activer, ou désactiver l'objet sélectionné, et l'utilisateur tape t pour « toggle ».

Peridot génère au fur et à mesure le code LISP correspondant à la procédure « PropSheet ».

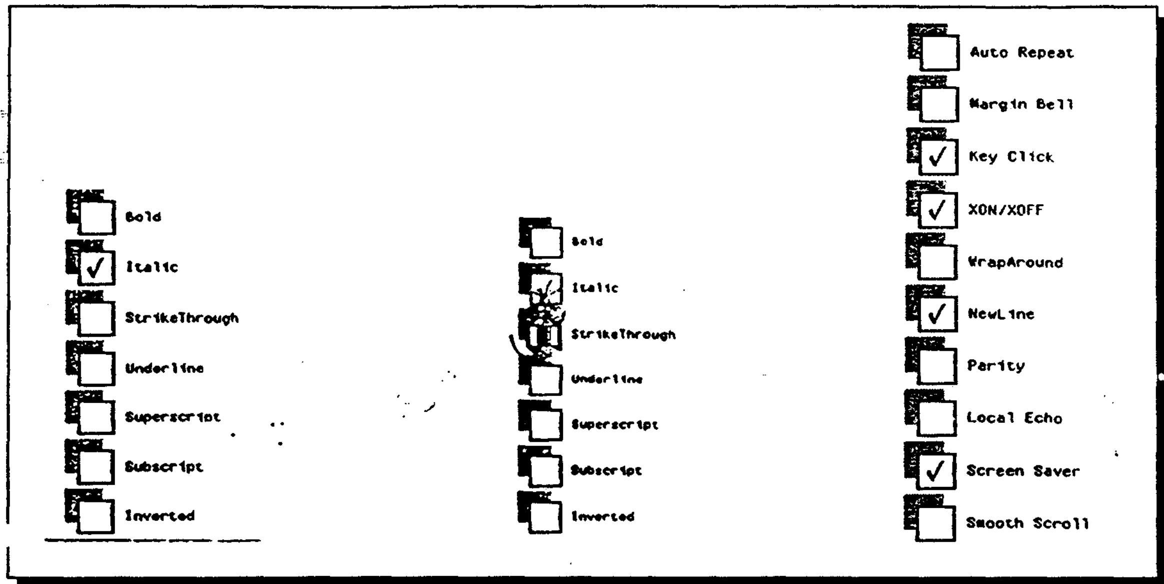


FIG. II.11 - *Peridot : l'aspect démonstrationnel*

Celle-ci peut être utilisée dans un programme en dehors de Peridot. Elle est paramétrée et peut donc être appelée avec une autre liste d'itèmes différents (figure II.11(c)).

Discussion

Les exemples permettent à ce type de langages d'utiliser des techniques de manipulation directe afin que le processus de conception soit concret plutôt qu'abstrait. Ce style est motivant dès lors que les gens font moins d'erreurs lorsqu'ils travaillent avec des exemples spécifiques plutôt que des idées abstraites. Il existe néanmoins quelques inconvénients que nous pouvons retrouver dans les systèmes de programmation par les exemples de manière générale :

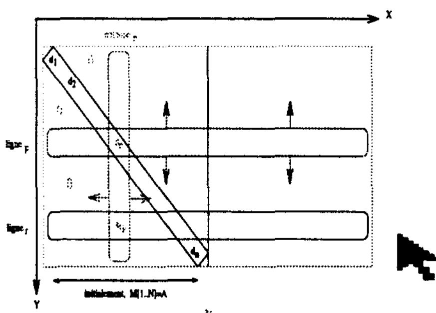
- Ces systèmes doivent parfois deviner (inférer), ce qui les amène forcément à faire des erreurs. Cela pourrait rendre inconfortable l'utilisateur qui voit surgir occasionnellement des erreurs. D'autant plus que, les systèmes ne sont pas toujours capables d'offrir aux utilisateurs la possibilité de revenir en arrière en utilisant le « undo » étant donné qu'il n'est pas simple à implémenter.
- La programmation par les exemples est plus complexe d'utilisation dans des cas où l'utilisateur connaît exactement ce qu'il souhaite et où un menu ou un icône permettent la sélection directe. Si l'utilisateur, dans l'exemple que nous avons cité plus haut, devait *démontrer* que le bouton milieu de la souris alterne l'état de l'objet sélectionné, il l'aurait fait en deux étapes : une fois lorsque l'objet est activé pour *démontrer* que le bouton milieu de la souris le désactive, et une seconde fois pour *démontrer* que le

bouton l'active. Autrement le cas d'alternance ne peut pas être distingué du cas toujours activer ou toujours désactiver. *Démontrer* cela est probablement plus coûteux en temps que de choisir dans un menu (comme le fait Peridot) parmi « alterner », « activer », et « désactiver ». Les concepteurs devraient utiliser cette technique de spécification seulement lorsque c'est approprié.

- Nous rappelons enfin l'échec de cette technique de spécification pour la mise en œuvre de langages compilés.

Un langage de programmation doit pouvoir manipuler des variables, des conditions et des itérations, éventuellement de façon implicite. Le langage de programmation visuelle Peridot offre ces possibilités, mais dans un contexte très limité. Une boucle itérative sert à répéter un certain nombre de fois, un ensemble d'actions utilisateurs. C'est un enregistrement de commandes réalisé à partir de deux itérations-exemples à rejouer plusieurs fois pour créer par exemple d'autres éléments. L'utilisateur doit être attentif en créant les deux itérations. Il doit réaliser les deux exemples de façon identique, autrement le système ne pourra jamais deviner qu'il s'agit d'une répétition. De plus une itération ne peut inclure que ce qui est *démontrable* interactivement dans l'éditeur graphique. Les conditions dans Peridot sont différentes de celles des langages conventionnels, nous ne pourrons pas spécifier par exemple un « If_Then_Else » ou un « Case ».

Il a été possible d'implémenter la base de règles de Peridot, car le domaine couvert par ce système est assez limité (construction d'interfaces). On est capable d'évaluer après chaque action-utilisateur les différentes possibilités d'actions pouvant se présenter ensuite. Le nombre de possibilités est raisonnable, il ne l'aurait pas été dans n'importe quel domaine, encore moins dans un langage à programmation générale. D'ailleurs, les inférences ont eu plus de succès lorsqu'elles s'appliquent à des programmes dans des domaines très limités [Mye92].



Vers un data-parallélisme visuel

Nous notons l'intérêt de la programmation basée sur les exemples. C'est l'unique moyen que nous connaissons qui permette aux programmeurs de développer un algorithme tout en observant en même temps le résultat visuel de leurs opérations. Ceci permet d'une part un contrôle direct de l'algorithme et d'autre part l'apprentissage du modèle de conception (par

exemple le modèle à parallélisme de données) et éventuellement le langage cible en regardant au fur et à mesure le code automatiquement produit.

3.1.6 Autres techniques de spécification

D'autres techniques de spécification sont exploitées pour la représentation des langages de programmation visuelle ; notamment des représentations basées sur les *formes*, *la réalité virtuelle*, ou *les feuilles de calcul* (SpreadSheet) appelées aussi tableurs.

Les *tableurs* tels que Lotus 1-2-3 ont été conçus pour aider les non programmeurs à faire la comptabilité. Selon B.A. Myers, ils ont eu beaucoup de succès car ils sont interprétés : l'utilisateur a immédiatement le résultat de son action ; ils offrent des agrégats et des opérations de haut niveau ; ils évitent la notion de variable (toutes les données sont visibles).

Les *formes* correspondent à des feuilles où peuvent être collés des objets, des cellules ou des compositions de cellules. Ils sont en quelque sorte similaires aux *feuilles de calcul.* En plus de la différence de l'aspect visuel, dans les *feuilles de calcul* ce sont les données qu'on met en évidence alors que dans les *formes* on retrouve aussi les constructions de contrôle sous forme visuelle.

La programmation basée sur la réalité virtuelle est très récente, nous en verrons un exemple plus loin. Comme chacun peut le deviner, le programmeur muni de gants et de lunettes entre dans un monde virtuel (un éditeur 3-D) et construit son programme.

3.2 Langages de programmation visuelle dédiés au parallélisme

Plusieurs langages ou environnements de programmation visuelle ont été mis en œuvre autour de la programmation séquentielle. Or, il est de plus en plus nécessaire d'avoir des langages et des environnements de programmation visuelle autour du parallélisme, d'autant plus que les programmes parallèles sont généralement plus difficiles à mettre en œuvre.

Nous décrirons, dans cette partie, quelques langages de programmation visuelle dédiés au parallélisme ou l'utilisant implicitement. Le langage Pigsty, développé à l'université d'Edinburgh, est basé sur les organigrammes de Nassi-Shneiderman (graphe de contrôle). Le langage NL, développé à l'université de Tasmania (Australie), est basé sur le traitement à flot de données. ALEX et Forms/3 permettent l'expression des algorithmes matriciels. Le premier langage se base sur les exemples et le deuxième sur les formes. CUBE est un langage de programmation logique basé sur la réalité virtuelle.

Le langage Pigsty

Pigsty	
Parallélisme :	oui
Domaine d'application :	multi-tâches
Technique de spécification :	graphe flot de contrôle (organigrammes de Nassi-Shneiderman)
Abstraction procédurale :	oui
Interprété/Compilé :	interprété

Le langage Pigsty est basé sur le langage Pascal et le modèle CSP (Communicating Sequential Processes) [Pon86]. Un programme Pigsty est une combinaison de texte et de graphiques. Les graphiques sont utilisés à deux niveaux. Ils représentent d'une part la structure globale du système composé de plusieurs processus en interaction, et d'autre part le flux de contrôle (l'ensemble des constructions de contrôle séquentiels et parallèles). Un programme Pigsty contient un ou plusieurs processus séquentiels qui peuvent communiquer les uns avec les autres. Un processus est représenté par un icône-boîte dans la fenêtre correspondant à la structure du système. Le codage correspondant à un processus est représenté par un graphe de contrôle, précisément un organigramme de Nassi-Shneiderman.

En effet, le programmeur a devant lui une représentation visuelle de son programme, mais malheureusement elle n'est pas faite de façon directe en utilisant par exemple un éditeur visuel. Le programmeur crée progressivement les représentations en introduisant textuellement des commandes que l'environnement va interpréter.

Pigsty permet également l'abstraction procédurale. Le programmeur a la possibilité de regrouper un sous-système de processus (un ensemble d'icônes-boîtes) en une seule boîte tout en retrouvant les ports d'entrée/sortie globaux du système dans la nouvelle boîte. Évidemment, il peut aussi étendre une boîte encapsulée pour retrouver le sous-système à l'intérieur.

La figure II.12 montre un exemple de programme Pigsty, c'est une solution au problème des cinq philosophes.

À signaler enfin que l'environnement I-PIGS qui gère la programmation en Pigsty permet aussi la visualisation d'un programme Pigsty. Il montre la partie du code en exécution, l'historique du contenu des variables, ainsi que l'animation des communications.

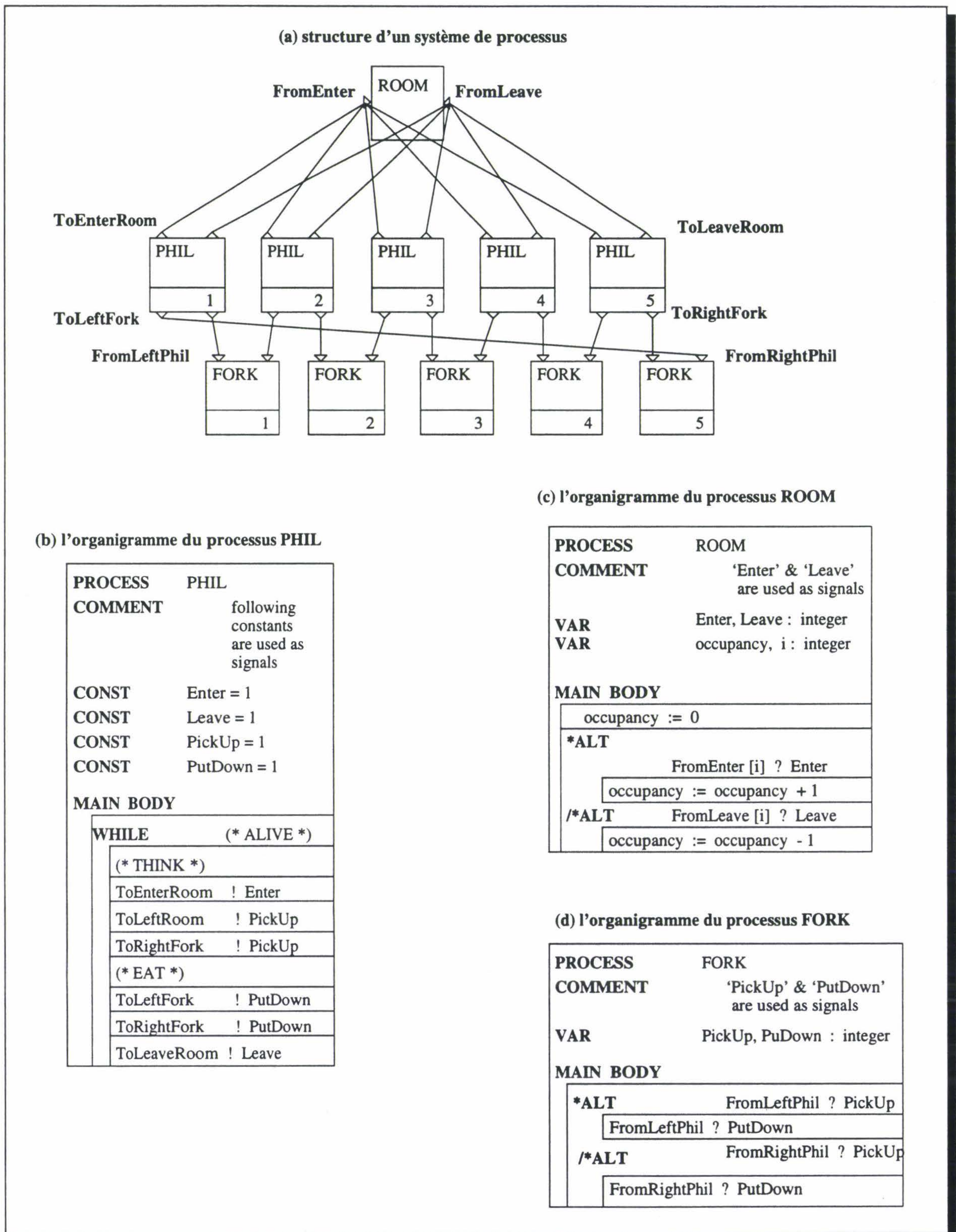


FIG. II.12 - *Pigsty: Un exemple de programme (Problème des cinq philosophes)*

Le langage NL

NL	
Parallélisme :	oui
Domaine d'application :	programmation générale
Technique de spécification :	graphe flot de données
Abstraction procédurale :	oui
Interprété/Compilé :	compilé

Un programme NL [HM94] est un *graphe flot de données.* Les données transitent entre les nœuds via les arcs. Les nœuds représentent les opérateurs qui transforment les données. NL est un langage fortement typé: il offre les types simples (integer, real, character, boolean), les types composés (record, array, stream) ainsi que les types fonctions. NL offre aussi, pour faciliter la tâche du programmeur, des constructeurs de plus haut niveau: des nœuds pour conditions (if), pour boucle d'itérations (while, for) et des nœuds de séquentialité. Un nœud peut lui-même être un graphe flot de données, ce qui permet au programmeur de travailler à n'importe quel niveau d'abstraction du programme (abstraction procédurale).

Nous avons choisi un petit exemple pour illustrer la programmation sous NL. Cet exemple développe le programme (graphe flot de données) correspondant aux calculs des racines carrées de l'équation quadratique Ax^2+Bx+C = 0.

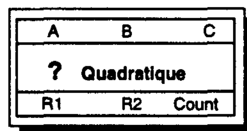


FIG. II.13 - *NL : le nœud If « Quadratique »*

La figure II.13 représente le plus haut niveau d'abstraction: le nœud principal (la représentation visuelle d'un nœud If). A, B et C représentent les entrées d'un nœud ; R1, R2 et Count les sorties.

Le programmeur peut travailler à plusieurs niveaux d'abstractions la figure II.14(a) montre l'expansion du premier nœud. Ce dernier représente 4 nœuds emboîtés (figure II.14(a)) : deux nœuds de tests et deux autres nœuds blocs. Cette figure montre justement comment est représenté le constructeur if_then_else. Chaque nœud contient un graphe flot de données. Les nœuds (1) et (3) de la figure II.14(a) représentent les tests A=0 et A<>0. Selon la valeur de A, un seul des nœuds (2) et (4) est exécuté. Si le nœud (2) est exécuté R1 aura la valeur (-C/B).

La figure II.14(b) représente l'expansion du nœud If « Racines » ; c'est le même principe dans un niveau d'abstraction plus bas que celui du nœud (4) de la figure II.14(a).

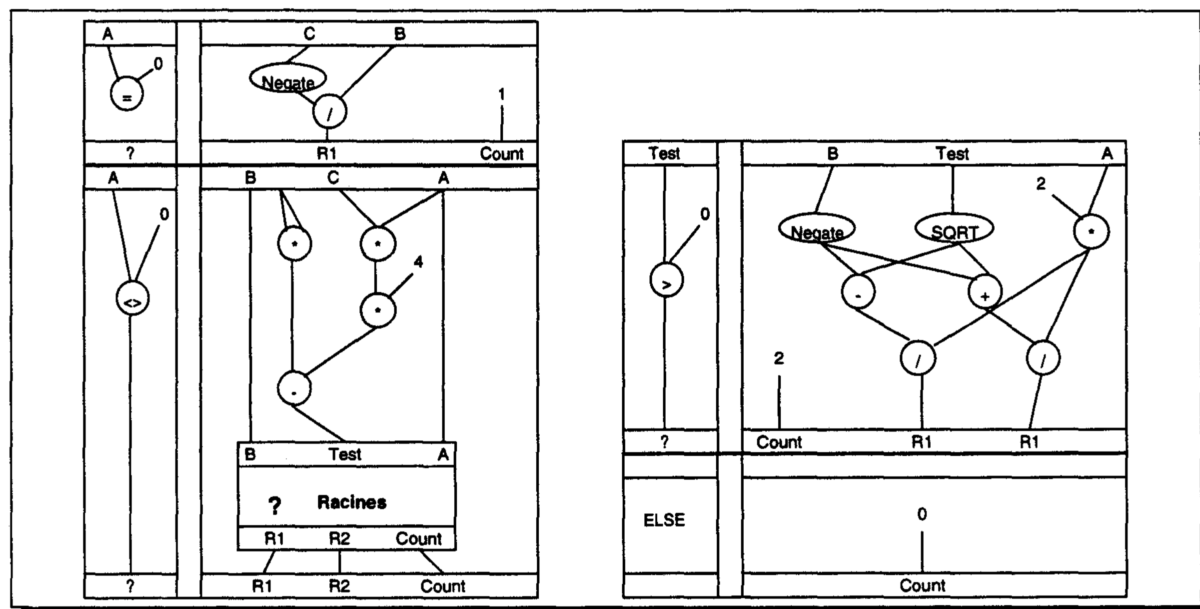
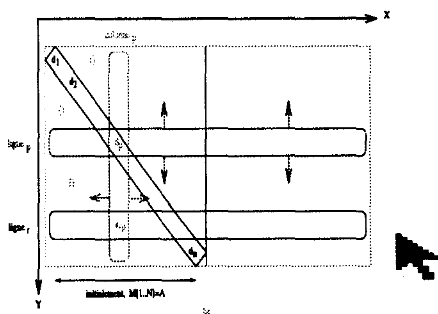


FIG. II.14 - *NL :(à gauche) Le nœud If « Quadratique » après expansion. (à droite) Le nœud If « Racines » après expansion*

Le programmeur peut définir ses propres nœuds qu'il rajoutera à la librairie. Il peut définir par exemple des expressions telles que « x^2 », « >0? », etc., ce qui permettra de réduire nettement la complexité des programmes.

En NL, le traitement parallèle est transparent au programmeur. Celui-ci conçoit son programme de manière séquentielle classique sans préciser explicitement un quelconque traitement parallèle. Le parallélisme n'est pas dans la conception, il est dans l'exécution par exemple lors de l'exécution d'une boucle.



Vers un data-parallélisme visuel

Nous retenons deux choses à travers NL :

1. La manière avec laquelle il assure l'abstraction procédurale. Le programme est un ensemble de boîtes les unes dans les autres.

2. Les paramètres d'entrées/sorties spécifiés par de petites cellules en entrée et en sortie des boîtes.

Il faut cependant faire attention au passage de paramètres réalisé à travers des arcs. On peut facilement engendrer le phénomène « spaghetti » (sans philosophes !). Il faut de ce fait réfléchir à d'autres moyens permettant d'assurer le passage de paramètres de façon plus lisible et plus conviviale.

Le langage ALEX

ALEX	
Parallélisme :	oui
Domaine d'application :	algorithmes matriciels
Technique de spécification :	programmation par les exemples + matrices
Abstraction procédurale :	oui
Interprété/Compilé :	interprété

ALEX (ALEXical programming language) [KTC+90] utilise les représentations visuelles de matrices pour définir des algorithmes numériques matriciels. Les seuls objets pouvant être manipulés sont les scalaires et les tableaux. C'est un langage de programmation visuelle *basé sur les exemples*, et il pourrait être adapté au parallélisme. Lorsqu'un élément, une ligne ou une colonne « typique » est sélectionné, alors toute opération qui lui est appliquée sera automatiquement et invisiblement répliquée à travers tout le tableau. Ceci correspond à la forme fonctionnelle « apply to all » dans la programmation fonctionnelle ; ça peut correspondre aussi à l'instruction FORALL dans le parallélisme de données. ALEX est un langage fonctionnel. En plus des représentations graphiques de matrices, un programme ALEX contient aussi des icônes « fonctions » : des représentations graphiques de fonctions ou d'appels de fonctions. L'utilisateur symbolise les dépendances de données et en particulier le passage de paramètres par des appariements de couleurs (au lieu des arcs). ALEX permet l'abstraction fonctionnelle (contre l'abstraction procédurale) : l'utilisateur peut encapsuler des parties de programme dans de nouvelles fonctions.

Le langage Forms/3

Forms/3	
Parallélisme :	oui
Domaine d'application :	algorithmes matriciels
Technique de spécification :	programmation basée sur les formes
Abstraction procédurale :	oui
Interprété/Compilé :	compilé

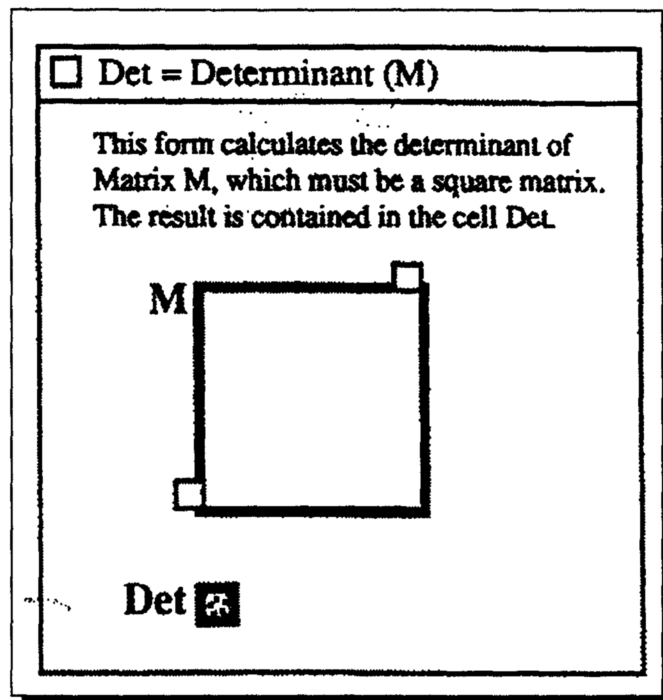


FIG. II.15 - *Forms/3: un exemple de forme avec des cellules et une matrice*

Forms/3 [GA92] (successeur de Forms/2 [AB90]) est un langage de programmation visuelle *basé sur les formes.* Il est dédié à la manipulation directe des matrices et il s'adresse principalement aux débutants en programmation. Le parallélisme ici est lié au traitement des matrices. La programmation en Forms/3 consiste à arranger et définir des formes. Une *forme* correspond à un morceau de papier sur lequel peuvent être collés des objets, qui sont soit des cellules soit des compositions de cellules telles que les matrices (cf. figure II.15). Une cellule consiste en une formule ou une équation permettant de déterminer sa valeur. Une formule peut être une constante ou des références à d'autres cellules composées avec des opérateurs (exemple : +, −, if, then, else, etc.) et/ou des appels de fonctions définies dans d'autres formes. La figure II.16 en montre un exemple.

Afin de définir la valeur de ses éléments, une matrice peut être partitionnée en une ou plusieurs zones rectangulaires appelées régions, ayant chacune une formule associée. Forms/3 utilise les couleurs pour identifier les différents ensembles de références, évitant ainsi les représentations textuelles qui auraient requis l'utilisation des indices. Le langage évite les itérations explicites en offrant des opérations de matrices et utilisant des formules génériques.

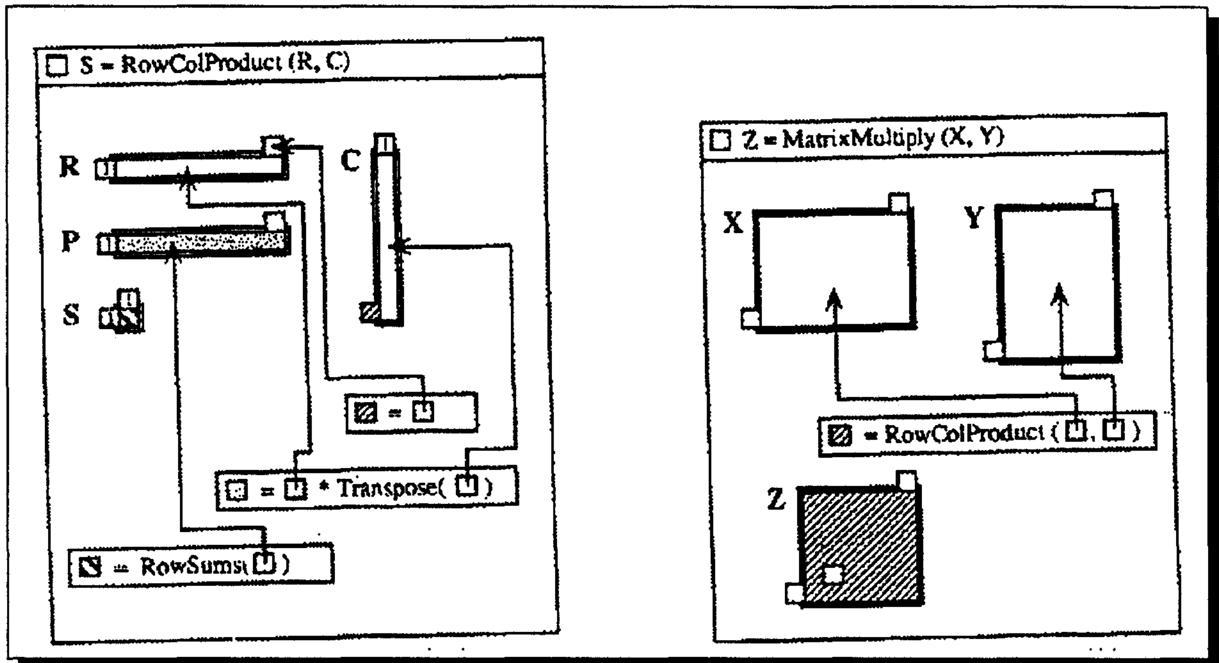


FIG. II.16 - *Forms/3: définition d'une forme pour le produit ligne-colonne et d'une autre forme pour la multiplication de matrices*

Le langage CUBE

CUBE	
Parallélisme :	oui
Domaine d'application :	programmation logique
Technique de spécification :	réalité virtuelle
Abstraction procédurale :	oui
Interprété/Compilé :	compilé

CUBE [NK91] est un langage dédié à la programmation visuelle. Il définit une syntaxe tridimensionnelle, qui devrait permettre d'éditer des programmes CUBE dans un environnement *basé sur la réalité virtuelle.* CUBE suit un modèle de programmation logique se basant sur la logique de Horn, c'est une logique d'ordre supérieur dans le sens où les prédicats sont traités comme des valeurs de première classe.

Un programme CUBE est représenté par un grand cube contenant des boîtes empilées verticalement, appelées des plans (*planes*). En CUBE, l'extension verticale indique des alternatives (chaque plan est une alternative) : dans le contexte des types c'est la somme des types, et dans le contexte de la logique c'est la disjonction. L'extension horizontale indique une combinaison : produit des types ou conjonction. Les éléments de base de la syntaxe CUBE sont : les cubes, les plans, les tubes (*pipes*), et les icônes.

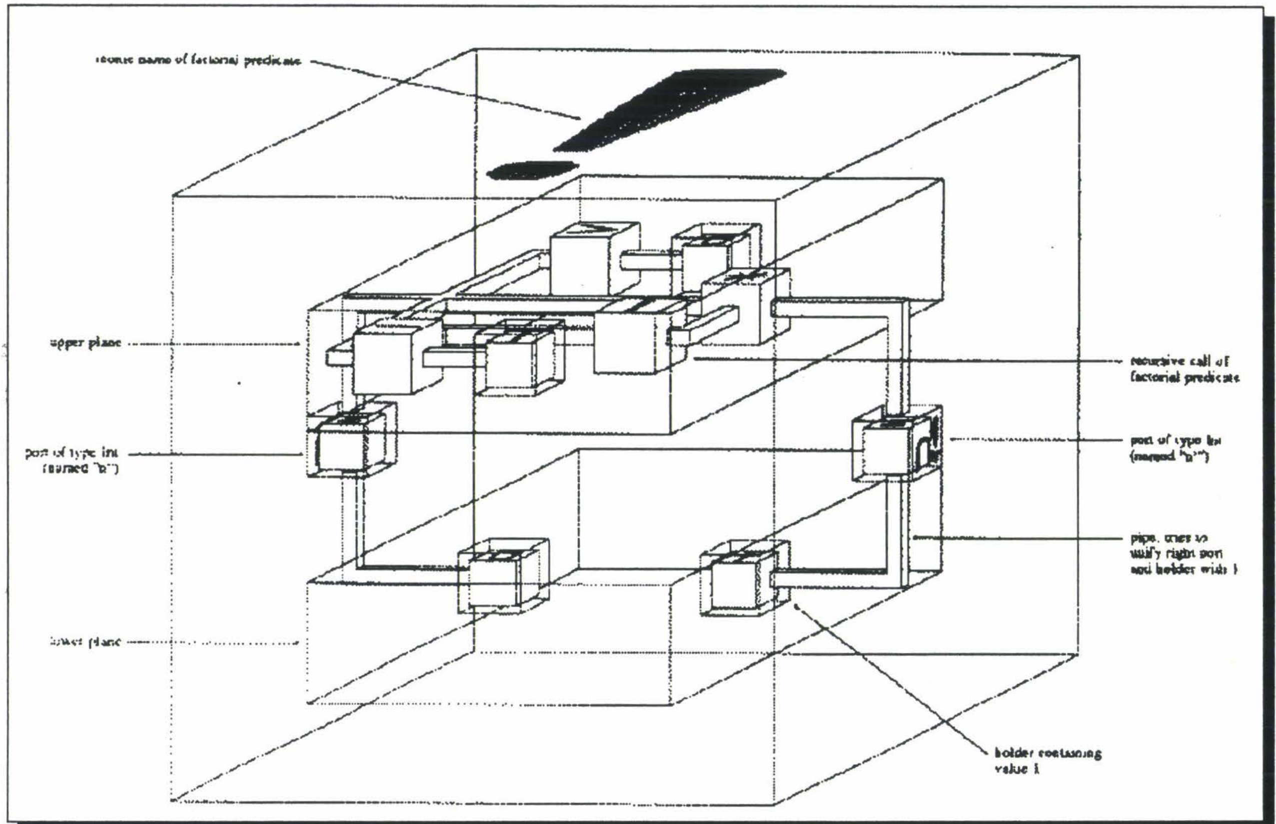


FIG. II.17 - CUBE : *définition du prédicat Factoriel*

La figure II.17 montre un exemple de programme CUBE : la définition récursive du prédicat Factoriel

$$\begin{gathered} fact \;=\; \lambda\{in : Int, out : Int\}.(in = 0 \;\wedge\; out = 1) \vee \\ (greater\{arg1 = in, arg2 = 0\} \;\wedge\; minus\{arg1 = in, arg2 = 1, res = y1\} \;\wedge \\ fact\{in = y1, out = y2\} \;\wedge\; times\{arg1 = in, arg2 = y2, res = out\}) \end{gathered}$$

Ce prédicat est représenté par un cube à deux ports, un en entrée et l'autre en sortie. Les valeurs entrent dans le cube à travers le port gauche pour être distribuées dans les deux plans internes. Les plans sont évalués indépendamment de manière concurrente.

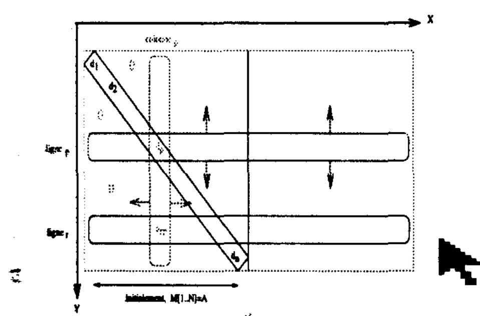
Si la valeur en entrée n'est pas nulle, et ne s'unifie donc pas avec la valeur 0 du cube en bas à gauche, le plan du bas échoue. Autrement, l'unification réussit et puisque il n'y a pas d'autres conditions à satisfaire, la valeur 1 du cube en bas à droite va être acheminée vers le port de sortie, et sera donc retournée comme résultat.

la valeur i en entrée sera distribuée aussi au plan du haut. Elle va être acheminée vers le prédicat *greater* « > » qui reçoit aussi, comme deuxième argument, la valeur 0 du cube à sa droite. Le prédicat réussira si i>0, sinon il échouera et causera par conséquent l'échec de

tout le plan du haut. i est acheminée aussi vers le prédicat *minus*, ensemble avec la valeur 1 du cube à droite du *minus*. Le résultat de la soustraction est envoyé dans une occurrence récursive du cube factoriel. Le résultat de ce dernier est acheminé à son tour vers un cube *multiplication* qui contient aussi la valeur i et qui va renvoyer le résultat au port à droite.

Comme la sémantique de la logique de Horn est intrinsèquement parallèle, telle est aussi la sémantique de CUBE. Une implémentation parallèle du langage était prévue, mais nous n'en savons pas plus jusqu'à présent.

Discussion Le principe du langage CUBE est sans doute un support pour la programmation visuelle ; il offre une nouvelle technique de spécification. CUBE est dédié à la programmation logique parallèle, mais il est certain qu'il ne sera utilisé que par des privilégiés pouvant se permettre des gants et des lunettes pour entrer dans la réalité virtuelle !



Vers un data-parallélisme visuel

À l'étude des trois derniers langages ALEX, Forms/3 et CUBE, nous retenons les points suivants :

1. La représentation et la manipulation des matrices en tant qu'objets graphiques (*cf.* ALEX et Forms/3). On peut s'en inspirer pour étendre cette représentation à toute structure homogène de données.

2. L'utilisation des couleurs pour le passage de paramètres dans ALEX, ce qui permet d'éviter les arcs.

3. La représentation graphique des formules dans Forms/3, ce qui permet au programmeur de voir concrètement ce qu'il est entrain de faire. À noter cependant que ça pourrait être encombrant à cause de l'utilisation des arcs, c'est pourquoi il faut réfléchir encore à une solution meilleure, par exemple substitution des arcs par un appariement de couleurs.

4. Enfin laissons nous rêver un peu en nous inspirant du langage CUBE. Pensons à un

environnement ou un langage de programmation visuelle à parallélisme de données où le programmeur peut se balader dans un espace virtuel tout en manipulant ses données.

4 Systèmes de visualisation de programmes

La « visualisation de programmes » est construite sur un concept entièrement différent de celui de la programmation visuelle. En programmation visuelle, les graphiques sont utilisés pour créer le programme lui-même, mais dans la visualisation de programmes, le programme est spécifié de manière textuelle et les graphiques sont utilisés pour illustrer certains aspects du programme ou de son exécution.

Nous examinons dans cette partie plusieurs classifications de systèmes de visualisations de programmes. Nous proposons ensuite notre classification en détaillant les critères que nous avons retenus. Nous décrirons enfin quelques systèmes de visualisation de programmes parallèles.

La classification la plus connue est probablement celle de Brad Myers [Mye86], révisée deux fois [Mye88, Mye90b]. Dans le dernier papier, il examine 19 systèmes de visualisation de programmes en les classifiant selon deux axes : (1) selon qu'ils illustrent le code, les données ou l'algorithme du programme, et (2) selon que la visualisation est statique ou dynamique. Cette classification constitue un point d'accès pour un débutant, mais il y a évidement d'autres critères pour distinguer encore mieux les systèmes de visualisation.

Roman et Cox [RC93] classifient les systèmes selon cinq critères, le premier est similaire au premier critère de B. Myers. Le deuxième critère décrit le *niveau d'abstraction* de l'information visualisée (représentation graphique directe, représentation structurelle, ou représentation synthétisée). Ce critère a été considéré aussi par Stasko et Patterson [SP92]. Le troisième critère de Roman et Cox concerne la *méthode de spécification*, appelée aussi dans d'autres classifications (exemple : [KS93]) « *instrumentation* ». Il précise les mécanismes utilisés pour construire la visualisation sachant que certains systèmes requièrent la modification du code et d'autres non. Ils sous-divisent ce critère en : prédéfinition, annotation, déclaration et/ou manipulation. Blaine Price [Pri90] regarde seulement si l'instrumentation se fait *manuellement* ou *automatiquement* (systèmes intelligents). Price et al. dans [PBS93] vont plus loin en détails lorsque c'est manuel ; ils regardent si elle se fait par un langage, une bibliothèque de routines à insérer, etc. Les deux derniers critères de Roman et Cox sont plutôt d'ordre ergonomique : « l'interface et son contrôle par l'utilisateur » et enfin la « présentation » qui montre de quelle

manière les visualisations véhiculent l'information.

4.1 Critères de classification

Dans notre classification, nous considérons les trois critères suivants: (1) « Portée de la visualisation » qui doit englober les deux critères de Myers, (2) « Niveau d'abstraction », et (3) « Instrumentation ».

4.1.1 Portée de la visualisation

La visualisation de programmes peut montrer graphiquement un ou plusieurs aspects d'un programme. Formellement, un programme peut être caractérisé par son code, ses données, et le comportement de l'exécution (l'algorithme).

Visualisation de codes Les systèmes de visualisation de codes illustrent le texte du programme en le convertissant en une forme graphique: organigramme, graphe de contrôle, graphe flot de données, diagramme de Nassi-Shneiderman, etc. La représentation peut être statique ou dynamique.

La figure II.18(A) [8], par exemple, montre un bout de code d'un programme de tri sous forme d'un diagramme de Nassi-Shneiderman [RC93]. Cette représentation statique est utile pour montrer la structure du code, mais n'apporte aucune information concernant le traitement.

Une visualisation dynamique par contre montre une animation du programme en exécution. Dans un graphe de contrôle par exemple, le nœud correspondant à la partie en cours d'exécution est coloré différemment des autres nœuds. Une forme de visualisation de codes dynamique souvent rencontrée est la représentation du texte du programme en mettant en évidence l'instruction courante. La figure II.18(B) en est un exemple. L'utilisateur peut suivre le déroulement du programme, en particulier au niveau des boucles imbriquées.

Visualisation de données Les systèmes de visualisation de données montrent des images des données actuelles du programme. La visualisation de données est souvent dynamique, mais peut être statique en montrant les structures de données (tableau, enregistrement, pile,...).

8. Les dessins utilisées dans cette sous section sont similaires aux vues générées par le système Pavane développé par Roman et Cox à l'université de Washington [RC93], et les images sont des copies d'écrans du même système.

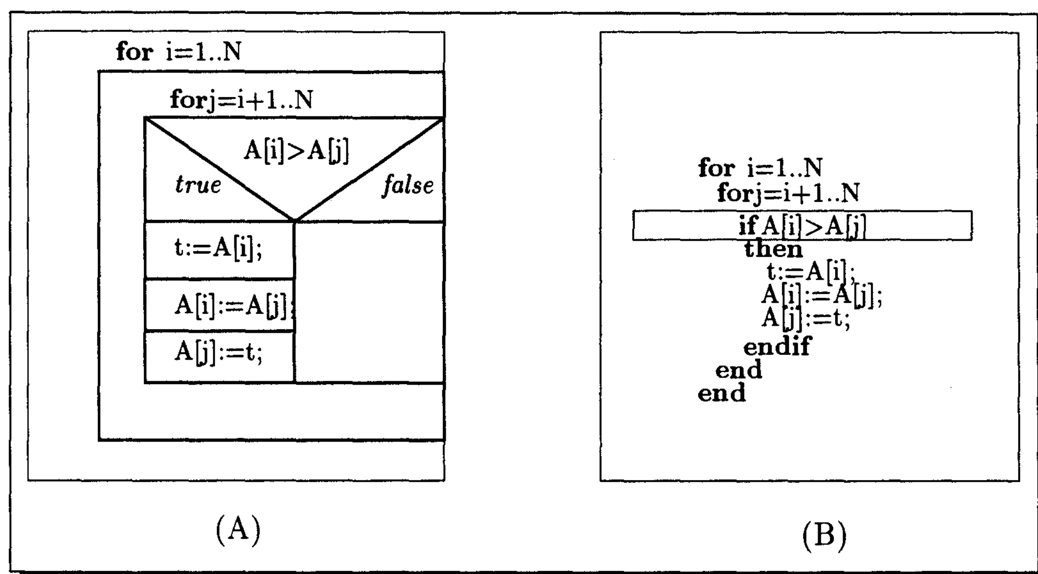


FIG. II.18 - *Un exemple de visualisation de codes: (A) statique, (B) dynamique*

Le but de cette dernière visualisation est de faciliter l'analyse du programme en évitant aux programmeurs de dessiner les structures à la main.

La figure II.19 visualise les données. Le tableau A est dessiné comme une rangée de boîtes dont la hauteur est proportionnelle à la valeur de l'élément stocké dans le tableau. Les variables i et j, utilisées comme indices du tableau, pointent avec des flèches les éléments correspondants. La visualisation dynamique de données peut représenter aussi des structures de données à partir du type, par exemple une liste chaînée. Les vues sont modifiées et mises à jour à chaque fois que la liste chaînée change (exemple: insertion d'un nouvel élément). L'utilisateur peut voir également le contenu d'un élément en cliquant dessus.

Visualisation d'algorithmes Les systèmes illustrant l'algorithme utilisent des graphiques pour montrer de manière abstraite le fonctionnement du programme. C'est différent de la visualisation de codes ou de données. Dans la visualisation d'algorithmes, les images peuvent ne pas correspondre directement aux données définies dans le programme, et les changements dans les images peuvent ne pas correspondre à des parties spécifiques du code. Par exemple, une animation d'algorithme d'une procédure de tri pourrait montrer les données comme des lignes de différentes hauteurs, et l'*interchangement* entre deux items pourrait être montré comme une animation de lignes se *déplaçant.* L'opération « interchangement » peut cependant ne pas être explicitement dans le code. La visualisation d'algorithmes est plutôt dynamique car elle montre le programme lors de son exécution.

La figure II.20 visualise l'algorithme de tri à travers la série d'interchangements effectués

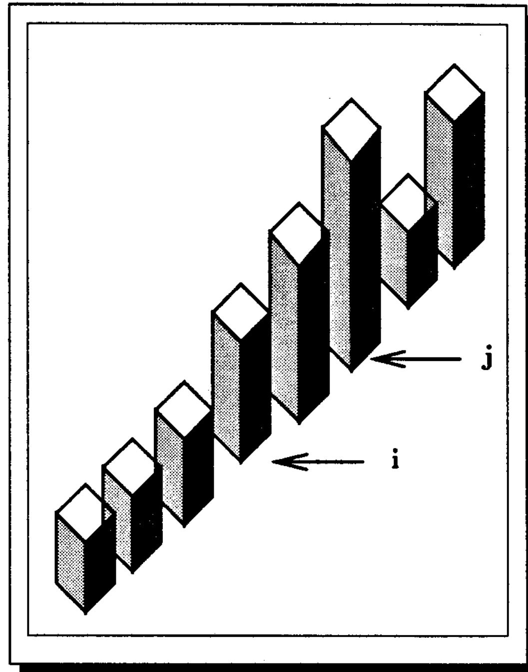


FIG. II.19 - *Un exemple de visualisation de données*

au cours de l'exécution, ainsi que les contenus initial et courant du tableau.

4.1.2 Niveau d'abstraction

Ce critère précise le type de l'information véhiculée par la visualisation. Il s'agit de montrer le niveau d'abstraction des concepts présentés graphiquement par le système de visualisation. Roman et Cox [RC93], qui utilisent aussi ce critère, distinguent trois niveaux d'abstraction que nous reprenons : représentation directe, représentation structurelle, et représentation

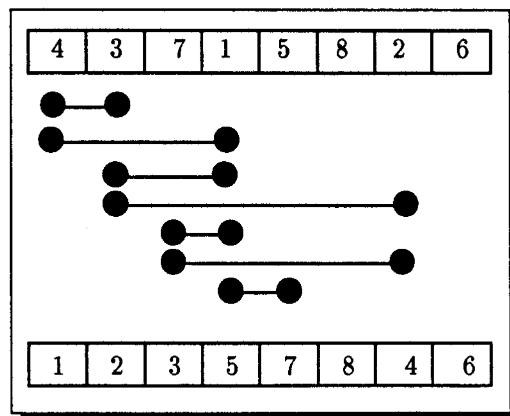


FIG. II.20 - *Un exemple de visualisation d'algorithmes*

synthétisée. Les limites entre les niveaux sont imprécises, et dans la pratique un système de visualisation est susceptible de supporter plusieurs niveaux d'abstraction, séparément et en combinaison.

Représentation directe C'est le niveau le plus bas d'une représentation graphique. Les systèmes de visualisation l'utilisant représentent le programme ou certains de ses aspects directement dans une image. Vu la grande limitation des mécanismes d'abstraction employés, l'utilisateur peut souvent et aisément reconstruire l'information originale à partir de la représentation graphique.

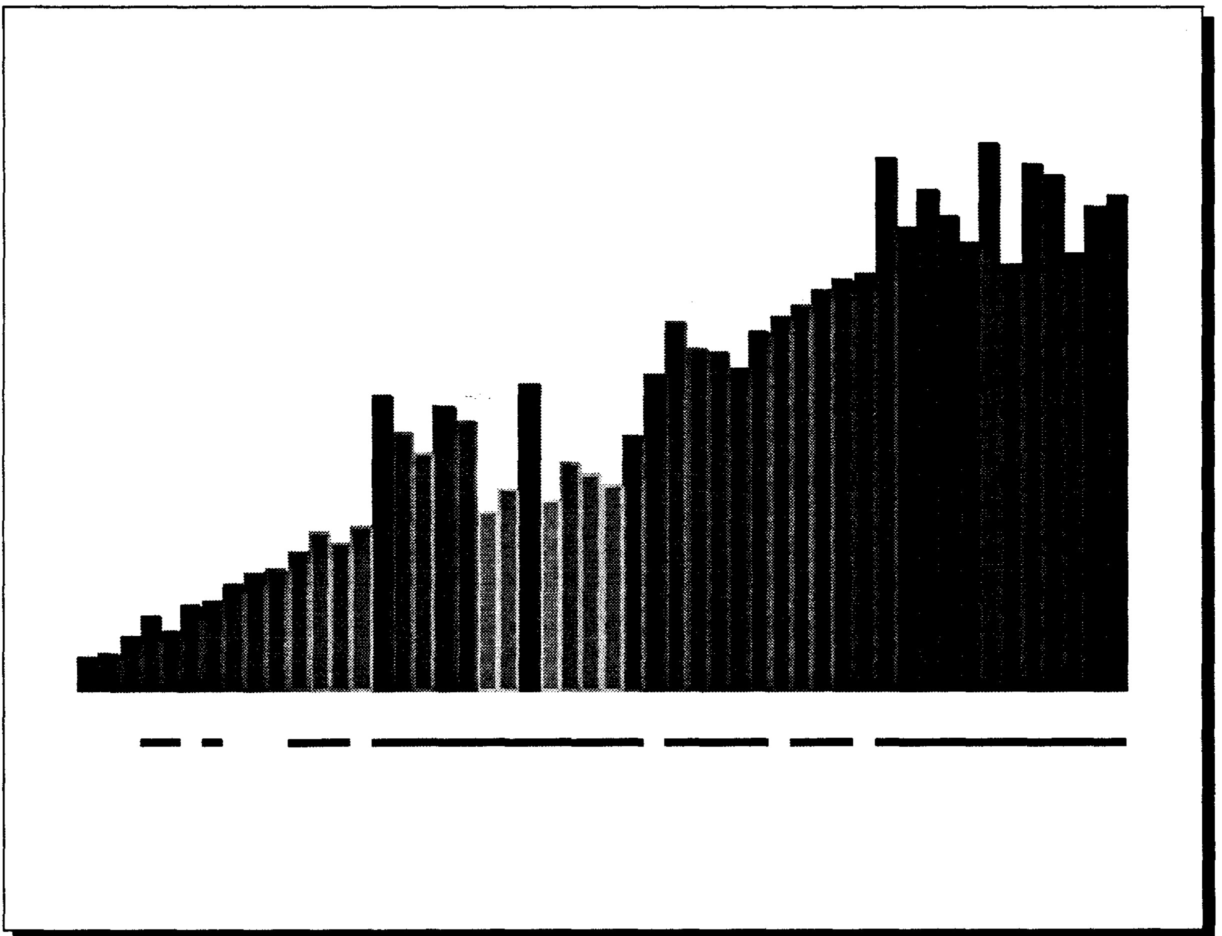


FIG. II.21 - *Représentation directe*

La figure II.21, générée par le système Pavane [RC93], est une représentation directe de l'algorithme du tri rapide « quicksort » [Sed91]. Elle montre les éléments du tableau à trier sous forme d'une rangée de rectangles dont la hauteur et la couleur sont déterminées par la valeur de l'élément. La ligne horizontale au-dessous de la rangée indique les régions qui restent à trier. Cette visualisation montre l'état du traitement mais n'aide pas spécialement

à la compréhension de l'algorithme.

Représentation structurelle Nous pouvons obtenir des représentations graphiques plus abstraites en focalisant l'attention sur des aspects particuliers du programme ou de son exécution. Une manière de le faire est d'annuler ou d'encapsuler des informations non pertinentes et de représenter directement le reste des informations. Une autre façon est de mettre en évidence les portions importantes de l'information dans l'image finale, en jouant par exemple sur les positions et les couleurs.

La figure II.22 montre un exemple de représentation structurelle. Les éléments triés sont visuellement séparés des autres éléments ; le travail accompli est placé au-dessus de ce qui reste à faire. Cette visualisation aide à mieux comprendre l'algorithme en montrant les pivots, les partitionnements et les éléments définitivement triés.

Représentation synthétisée Les représentations synthétisées sont distinctes des représentations structurelles dans le sens où l'information d'intérêt ne se trouve pas directement dans le programme mais dérivée à partir des données du programme. Certaines informations peuvent être logiquement présentes dans le programme, mais pas représentées de manière explicite.

Dans la figure II.23, l'exemple du tri rapide est repris avec un niveau d'abstraction plus haut. Une des manières d'expliquer le partitionnement et le tri est d'utiliser un arbre binaire, comme le fait d'ailleurs R. Sedgewick dans [Sed91]. Ici, l'algorithme est vu comme une structure d'arbre où chaque nœud correspond à un élément trié ou une région à trier. Lorsque celle-ci est traitée, elle produit un sous-arbre dont la racine est le nouvel élément trié et les fils les nouvelles régions. Cette structure d'arbre ne figure pas avec les structures de données du programme, elle doit être synthétisée par la visualisation.

4.1.3 Instrumentation

Dans le premier critère nous avons vu les différents types de visualisation. Le second critère nous a montré qu'il est possible d'augmenter le niveau d'abstraction de ce qui peut être visualisé. Chacun de nous peut imaginer plusieurs façons d'expliquer un programme et donc plusieurs manières de le visualiser afin qu'il soit aussi compréhensible que l'on souhaite. Nous pouvons cependant nous demander comment il serait possible de réaliser ces visualisations. Autrement dit comment se fait l'instrumentation ? Ce troisième critère « instrumentation » distinguera deux manières différentes de commander les vues souhaitées.

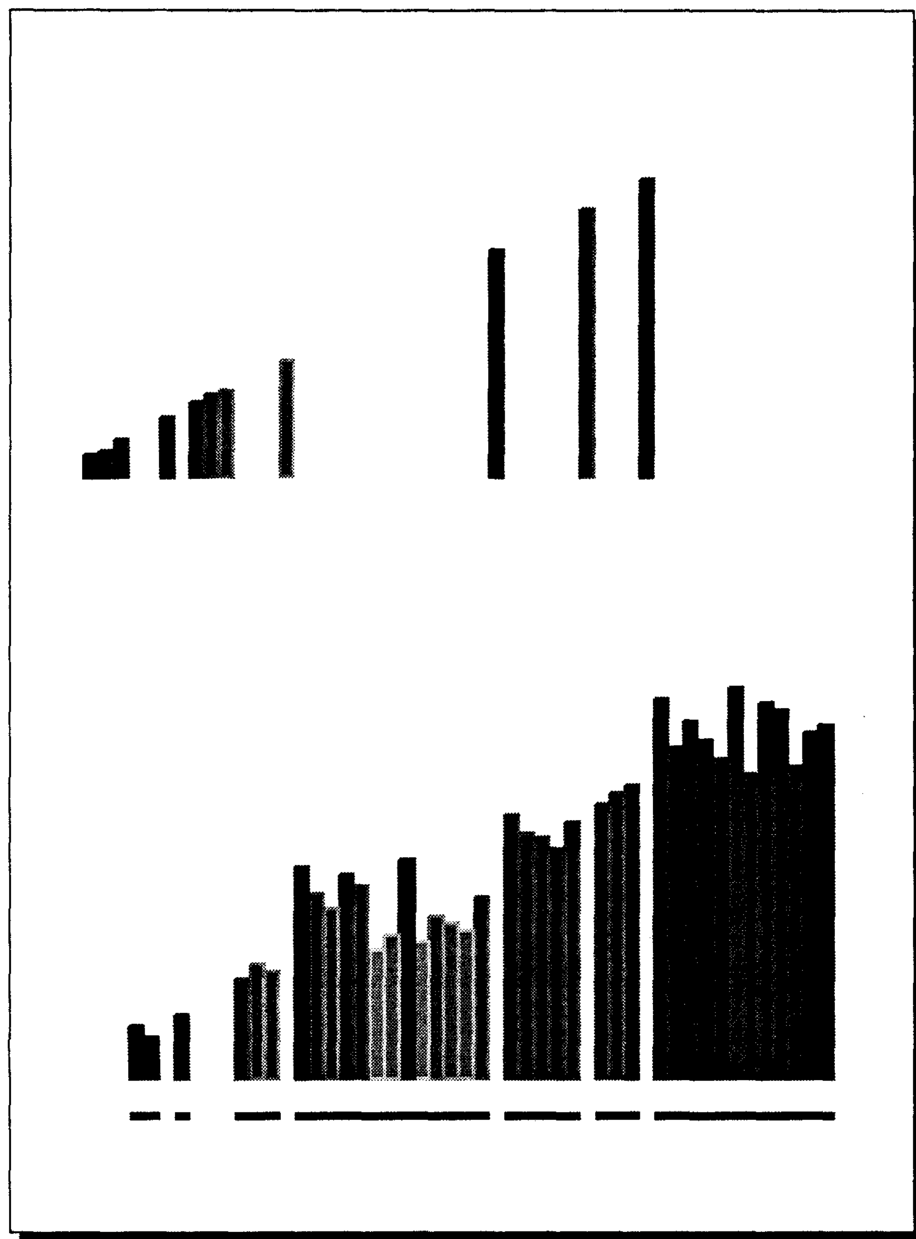


FIG. II.22 - *Représentation structurelle*

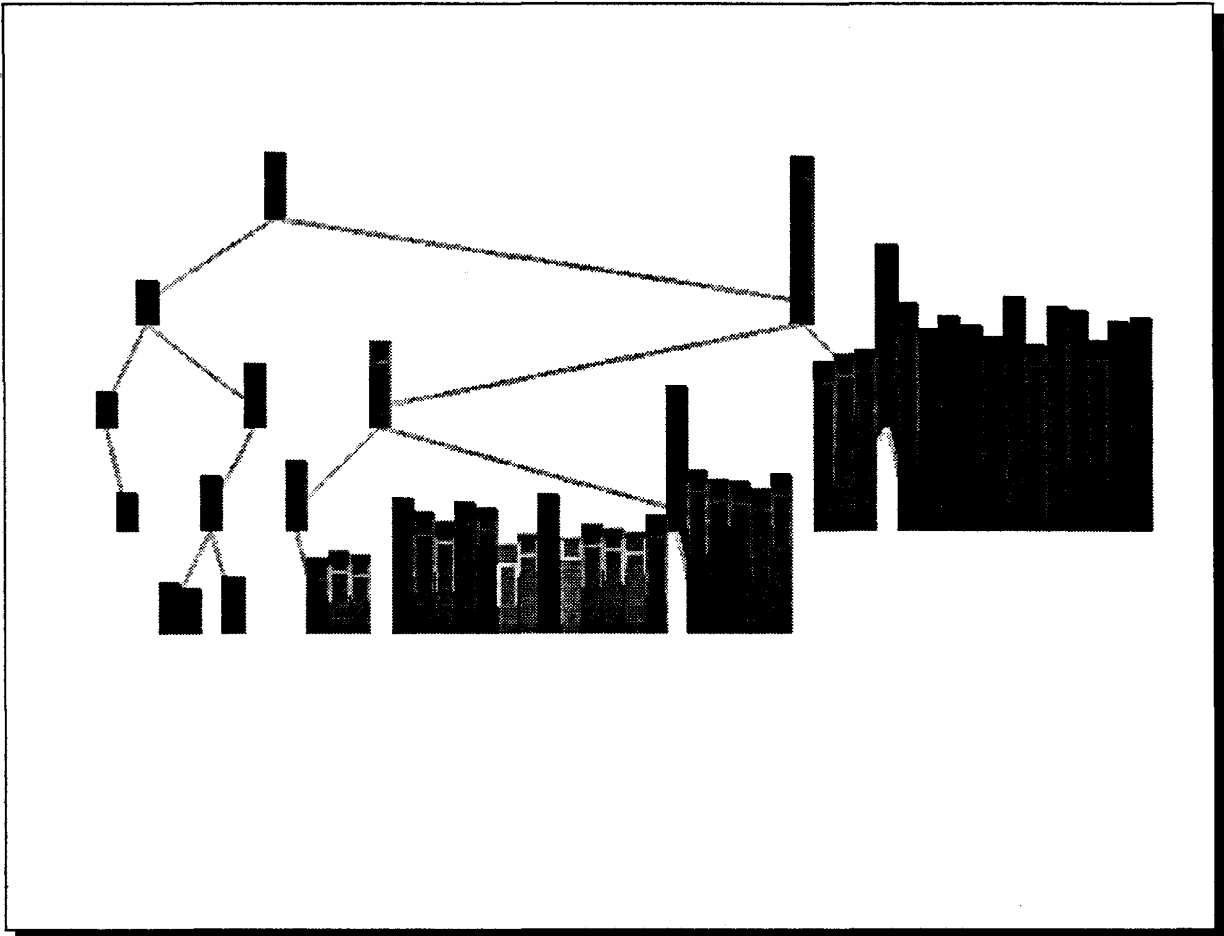


FIG. II.23 - *Représentation synthétisée*

Manuelle Dans les systèmes de visualisation à instrumentation manuelle, l'utilisateur doit lui-même modifier le programme et insérer le code nécessaire à la visualisation. Il rajoute généralement des appels de routines de visualisation prédéfinies.

Automatique L'instrumentation automatique signifie que le système est capable de produire une visualisation sans l'intervention manuelle de l'utilisateur. Il analyse le code puis insère les routines adéquates à la visualisation (selon le niveau d'abstraction, cela peut relever de l'intelligence artificielle). Bien sûr une instrumentation automatique n'empêche pas l'utilisateur de personnaliser la visualisation générée automatiquement.

4.2 Systèmes de visualisation de programmes parallèles

Les environnements graphiques de déverminage et d'analyse de performances que nous avons vus au premier chapitre tels que IVE et LIVE, sont appelés aussi systèmes de visuali-

sation. Dans cette partie nous en avons choisi d'autres, car d'une part nous voulons éviter de reprendre les mêmes systèmes qu'au premier chapitre, et d'autre part ceux que nous allons décrire ont été particulièrement conçus et dédiés à la visualisation de programmes (pour ces derniers systèmes nous avons plus d'informations concernant l'aspect visualisation et ayant trait à nos critères, qu'en ce qui concerne les systèmes du chapitre précédent). Nous retrouverons cependant, pour montrer comment les systèmes du premier chapitre peuvent entrer dans notre classification, le deux systèmes IVE et LIVE.

Dans notre description, nous insisterons surtout sur la méthode d'instrumentation qui nous semble la tâche la plus difficile dans la visualisation. Une description plus approfondie se trouve dans [KS93]. Le tableau 4.2 donne la classification de ces systèmes selon les critères que nous venons de décrire.

TAB. II.2 - *Classification des systèmes de visualisation de programmes*

	Portée de la visualisation			Niveau d'abstraction			
Systèmes	Code	Données	Algorithme	Directe	Structurelle	Synthétisée	Instrumentation
ANIM			S/D[a]	P[b]	P	P	manuelle
Belvedere		D	D	⋆	⋆		automatique
IVE		D		⋆	⋆	⋆	manuelle
LIVE		D	D	⋆	⋆		semi-automatique[c]
Pavane		D	D	⋆	⋆	⋆	manuelle
VISTOP			D	⋆	⋆		automatique
Voyeur	D	D		⋆	P	P	manuelle
Zeus	D	D	D	⋆	⋆	⋆	manuelle

[a] S = Statique, D = Dynamique
[b] P = Possible, puisque c'est l'utilisateur qui développe les routines graphiques
[c] Une approche médiane entre l'instrumentation manuelle et automatique

Pour la suite nous avons choisi de partager les systèmes en trois groupes, selon la méthode d'instrumentation : manuelle, automatique, ou entre les deux une instrumentation semi-automatique.

4.2.1 Systèmes à instrumentation manuelle

Voyeur D. Socha et ses collaborateurs à l'université de Washington ont développé un système de visualisation appelé Voyeur [SBN89]. L'utilisateur localise des événements importants dans le programme, tels que les changements dans les structures de données ou les appels et retours des procédures, il insère ensuite des appels de routines. Ces dernières envoient lors de l'exécution un simple message textuel au système de visualisation pour lui indiquer l'état du programme. Le système anime enfin des vues construites manuellement par l'utilisateur en programmant par exemple sous XWindow. Comme son flot d'entrée est purement textuel se résumant à des messages ASCII, ce système de visualisation est compatible avec n'importe quel programme utilisateur capable d'émettre ce flot d'entrée. Les auteurs ont créées des vues pour des programmes exécutés sur un simulateur d'architecture MIMD à mémoire distribuée, et pour des programmes multi-tâches exécutés sur un vrai multi-processeurs à mémoire partagée. L'avantage des vues créées manuellement est leur fidélité au modèle conceptuel du programme; elles montrent une idée plus claire de l'exécution du programme [Pri90].

IVE L'environnement IVE (Integrated Visualization Environment), développé par M. Friedell *et al.* [FLK+91] à l'université de Harvard (Massachusetts), permet de visualiser des programmes massivement parallèles s'exécutant sur une machine SIMD (dans ce cas la Connection Machine). L'utilisateur doit insérer lui même les appels de routines permettant de montrer les données. Ces routines se basent sur le langage DMPL (Data Mapping Programming Language) dédié au partitionnement et au placement des données. Lorsqu'elles sont exécutées, les routines envoient un certain résultat, généralement le contenu d'un tableau, à un processus permettant de le visualiser sous plusieurs formes. La visualisation peut être directe, montrant par exemple le placement des données d'un tableau sur une grille de processeurs. Elle peut être structurelle, le système de visualisation utilise par exemple les couleurs pour mettre en évidence les communications de voisinage. La visualisation peut être également synthétisée: l'utilisateur peut voir des formes géométriques qui n'existent pas explicitement dans le programme, telles que deux planètes en collision !

Zeus Le système de visualisation Zeus est la dernière évolution du système BALSA [Bro88], un pionnier de la visualisation graphique mis en œuvre en 1983 pour traiter des programmes Pascal. Zeus développé par M. Brown est dédié à la programmation multi-tâches [Bro91]. Il visualise des programmes écrits en Modula-3. L'utilisateur peut voir simultanément plusieurs vues synchronisées du programme. Zeus supporte la visualisation de codes (la routine courante avec mise en évidence de l'instruction qui s'exécute), la visualisation de données et la visualisation d'algorithmes. L'utilisateur doit lui-même annoter le programme en insérant des appels de routines de visualisation. En plus de la bibliothèque de routines prédéfinies, Zeus accepte de nouvelles routines écrites par l'utilisateur. Selon Roman et Cox [RC93] (ceux qui ont

proposé les différents niveaux d'abstraction), ce système offre tous les niveaux d'abstraction. La dernière version de Zeus inclut une visualisation 3-D [PBS93].

ANIM À la différence des autre systèmes, ANIM permet en plus de la visualisation dynamique, la visualisation statique en générant des copies d'écran à insérer par exemple dans un document. Il a été développé par Bentley et Kerninghan [BK91] aux laboratoires AT&T Bell. Il permet d'animer des programmes ou de rejouer des scripts générés lors de l'exécution. En effet, l'utilisateur peut annoter le programme de sorte qu'il génère un ensemble de commandes textuels dans un fichier de script. L'utilisation des commandes textuelles permet à ANIM d'être indépendant de tout langage. ANIM visualise des algorithmes de manipulation de matrices, des programmes d'analyse numériques, des algorithmes parallèles, etc. Les routines d'annotation sont de très bas niveau (« line, text, boxe, circle »), elles permettent d'un côté de réaliser les visualisations souhaitées mais posent au même temps un handicap à l'utilisateur qui doit tout dessiner lui même. L'utilisateur peut par ailleurs réaliser l'abstraction qu'il souhaite.

Pavane Le système Pavane [GCCWP92], développé à l'université de Washington (St. Louis), permet la visualisation de programmes concurrents écrits en Swarm ou en C. À l'inverse des systèmes utilisant un style impératif, où l'utilisateur doit insérer des appels de routines dans le code, Pavane utilise un style déclaratif. L'utilisateur définit (déclare) une transformation entre l'état du programme et l'image à visualiser ; il lie les attributs des objets graphiques aux valeurs des variables utilisées dans le programme. Selon Roman et Cox [RC93], Pavane est le premier système à utiliser cette approche. Le premier avantage d'un tel style est de pouvoir visualiser des images plus facilement que dans le cas où l'utilisateur doit lui même définir et isoler les événements importants à la visualisation. Les spécifications compactes constituent aussi un avantage de ce style. Pavane permet seulement la visualisation de données et d'algorithmes. Il offre par contre tous les niveaux d'abstractions. Nous avons présenté, lors de la définition du deuxième critère de classification, des images de Pavane montrant les différents niveaux d'abstraction. Pavane permet également la visualisation et la manipulation des images en 3-D.

4.2.2 Systèmes à instrumentation automatique

Belvedere Il est souvent difficile de visualiser des programmes concurrents en offrant une abstraction assez proche du modèle mental humain. Considérons une application qui s'exécute sur un hypercube où les nœuds doivent réaliser une séquence de communications, chacune selon une dimension. Si les processus communiquent indépendamment les uns des l'autres sans synchronisation, les vues correspondantes n'auront aucune signification. L'utilisateur verra des

communications dans tous les sens, puisque les événements surviennent dans n'importe quel ordre. A. Hough et J. Cuny de l'université du Massachusetts [AC90] ont proposé des techniques de réordonnancement des événements. Ils les regroupent de sorte que la visualisation soit significative. Ils essayent de construire au début des ordonnancements cohérents, permettant de préserver l'ordre partiel des événements imposés par la séquentialité des processus et les dépendances inter-processus. Dans l'exemple de l'hypercube, l'ordonnancement consiste à regrouper les événements selon la phase de communication : une phase pour chaque dimension. À noter qu'avec certains types de dépendances, par exemple les dépendances croisées, ces ordonnancements ne sont pas réalisables.

Ces techniques ont été employées par le système de visualisation « Belvedere » développé par les mêmes auteurs. Ce système permet la visualisation dynamique de codes et d'algorithmes (par exemple : des processus communicants). Il offre une abstraction structurelle de la visualisation à partir d'une instrumentation automatique. Il visualise par exemple la topologie de la machine cible ainsi que les communications qui s'y déroulent.

VISTOP Thomas Bemmerl et ses collaborateurs [BB93] ont mis en œuvre un environnement intégrant plusieurs outils de programmation distribuée appelé TOPSYS (TOols for Parallel SYStems). Un de ces outils est dédié à la visualisation de programmes à passage de message : VISTOP (VISualization TOol for Parallel systems). TOPSYS a été implémenté avec la bibliothèque MMK (Multiprocessor Multitasking Kernel). MMK supporte un style de programmation orienté objet et offre trois types d'objets prédéfinis : tâches, boîtes aux lettres, et sémaphores. Les tâches communiquent (de manière synchrone ou asynchrone) via les boîtes aux lettres et peuvent être synchronisées à l'aide des sémaphores. L'utilisateur peut définir de nouveaux objets composés en construisant interactivement un graphe. VISTOP visualise le programme à travers ces objets en montrant les relations qui les lient ; à chaque objet lui est associé un icône. L'animation de l'algorithme consiste à bien positionner les icônes, montrer le flot de contrôle à l'aide des flèches, et rajouter dans chaque icône un petit texte expliquant son état. L'utilisateur n'est pas obligé de voir tous les objets, il sélectionne seulement ceux qu'il souhaite voir et peut choisir également de commencer la visualisation à partir d'un endroit quelconque du programme (il se sert d'un autre outil DETOP -DEbugging TOol for Parallel systems). L'instrumentation se fait de manière automatique. Lors de l'exécution d'un programme, une couche d'acquisition de données récupère les informations et les dépose dans une file d'attente où elles sont lues par l'animateur. VISTOP offre deux niveaux d'abstraction : directe et structurelle. TOPSYS tourne sur plusieurs plate-formes : des machines multi-processeurs IPSC/2 et IPSC/860 ainsi que sur un réseau de stations SPARC (SUN).

4.2.3 Systèmes à instrumentation semi-automatique

LIVE La plus part des systèmes de visualisation que nous avons vus travaillent avec une instrumentation manuelle. Il est en effet très difficile de réaliser automatiquement des vues montrant un comportement abstrait du programme. Les concepteurs du système de visualisation LIVE (Lockheed[9] Integrated Visualization Environment) ont choisi une approche médiane [LSCA93] pour visualiser des programmes data-parallèles. Ils qualifient leur système de visualisation de semi-automatique. La partie manuelle, faite donc par l'utilisateur, se base sur un langage de spécification formelle (Larch). L'utilisateur écrit un programme où il associe à chaque structure de données à visualiser une description formelle[10]. Celle-ci doit spécifier les opérations pouvant être appliquées à la structure. Pour une pile par exemple, il doit spécifier dans le langage Larch les opérations « empiler, dépiler, taille, sommet, pileVide ». Ce programme est indépendant du programme à visualiser : il peut être réutiliser avec n'importe quel programme. Vient ensuite la partie automatique du système. Elle consiste à créer, à partir d'une base de règles et du programme de spécification, des modèles (*templates*) de visualisation paramétrés; des vues paramétrées. Le choix de la visualisation dépend des erreurs attendues ou suspectées dans le programme. Enfin, la visualisation consiste à montrer ces vues en remplaçant les paramètres par ceux du programme en question.

LIVE permet la visualisation de données et d'algorithmes en se basant sur les changements dans les structures de données et les opérations qui leurs sont appliquées. Il offre une visualisation dynamique avec deux niveaux d'abstraction : abstraction directe et structurelle.

5 Problèmes généraux et perspectives

Les systèmes de programmation visuelle offrent une représentation de plus haut niveau et plus expressive de certains aspects du programme. Ils font abstraction des détails textuels traditionnels, tels que la syntaxe et la spécification textuelle linéaire [Mye90b, Rae85, AB89]. C'est un avantage majeur, mais il reste néanmoins des problèmes qu'il faut résoudre.

Les concepteurs de langages de programmation textuelle bénéficient d'outils tels que Lex et Yacc pour la spécification des langages. Ce n'est cependant pas le cas dans le domaine de la programmation visuelle. Les travaux de David McIntyre s'inscrivent dans ce contexte ; il a proposé dans sa thèse [McI92] un système appelé VAMPIRE (VisuAl Metatools for Programming Iconic enviRonmEnts) dédié à la génération de langages à programmation icônique. Mais il reste beaucoup à faire dans ce domaine.

Un autre problème fondamental est le manque de bases théoriques pour le domaine. Le manque d'un équivalent dans le domaine visuel à Backus-Nur Form [Nau60] par exemple empêche la description complète est précise des langages visuels. Ceci dit il y a quelques

9. Lockheed est le nom du laboratoire où le système a été développé

10. Les auteurs prévoient d'étendre la spécification au comportement du flot de contrôle.

travaux de recherche qui tentent de combler ce vide, notamment les travaux de Sherif El-Kassas [EK94], de David McIntyre [McI92], et de Eric Golin [GR90].

Plusieurs chercheurs se posent aussi la question du transfert ou d'échange de programmes visuels. Le problème ne se pose pas pour les programmes textuels, car tout le monde sait lire un texte ASCII. Or, en ce qui concerne les programmes visuels, il n'y a pas de format universel que tout le monde peut utiliser pour lire un programme développé dans tel ou tel langage de programmation visuelle. Là aussi le problème de portabilité reste ouvert.

Les langages icôniques présentent quelques inconvénients. Kenneth Lodding [Lod82] fait remarquer que certains icônes sont intrinsèquement ambigus et d'autres peuvent être interprétés seulement dans un certain contexte. Pour assurer l'interprétation correcte d'un icône, nous devons considérer attentivement la conception de l'image, la légende associée à l'image, et le contexte dans lequel l'icône apparaît. Là encore un effort doit être fait pour trouver un ensemble d'icônes universellement acceptés (un peu comme le langage des signes pour les sourds-muets).

Les systèmes de visualisation de programmes ne sont pas utilisés comme outils de génie logiciel, car la plupart exigent du programmeur de changer manuellement le code source. Il doit insérer des routines de visualisation et parfois en écrire des nouvelles. Quelque soit le résultat qu'il peut penser obtenir, il est souvent réticent de le faire surtout lorsqu'il s'agit d'un programme de plusieurs milliers de lignes. La solution serait de mettre en œuvre des systèmes de visualisation capable de modifier le source de manière automatique. Sans ce type de système, la visualisation de programme se limitera à un domaine éducatif.

6 Conclusion

Nous avons défini, dans la première partie de ce chapitre, les termes souvent employés avec les langages visuels. Nous avons séparé en particulier la programmation visuelle de la visualisation de programmes. Après avoir comparé plusieurs classifications de langages de programmation visuelle, nous avons proposé la notre suivant les critères que nous avons présentés. Nous avons décrit ensuite les différentes techniques de spécification de programmes visuels. Chaque technique a été illustrée par la description d'un langage. À la fin de la partie réservée aux langages de programmation visuelle, nous avons décrit quelques langages dédiés au parallélisme en montrant comment ils rentrent dans la classification. La deuxième partie du chapitre a traité des systèmes de visualisation de programmes. Comme dans la partie précédente, à la suite d'une comparaison de plusieurs classifications de systèmes de visualisation, nous avons proposé la notre en présentant les critères que nous avons adoptés. Enfin plusieurs systèmes de visualisation ont été décrits en particulier selon cette classification.

Nous avons fait ressortir, le long des deux chapitres (I et II), des points intéressants qui peuvent avoir un apport important en ce qui concerne la programmation visuelle ou un environnement visuel autour du parallélisme de données (*cf.* les paragraphes *Vers un data-parallélisme visuel*). Ces différents points ont été particulièrement pris en compte pour la mise en œuvre de notre environnement HELPDraw, que nous verrons dans le chapitre suivant.

Chapitre III

L'environnement de programmation HELPDraw

L'objectif principal de notre environnement visuel HELPDraw est de traduire automatiquement, à partir de manipulations directes et interactives, la conception et la pensée data-parallèle du programmeur en un programme data-parallèle compilable. HELPDraw doit permettre aux programmeurs de développer leurs codes indépendamment du langage et de l'architecture de la machine cible. Le programmeur doit avoir les moyens d'exprimer de manière directe sa pensée data-parallèle. Il doit pouvoir dessiner son programme tout en observant simultanément le résultat visuel des opérations data-parallèles qu'il utilise. Cet aspect devrait permettre aux utilisateurs (surtout les scientifiques) de mieux appréhender le parallélisme de données.

Le succès et l'efficacité d'un tel environnement réside dans le support visuel sur lequel il se base. Plus le support est proche du raisonnement humain, plus l'environnement devient aisé et naturel à utiliser.

Dans le parallélisme de données, chaque objet est un ensemble d'éléments homogènes. Généralement le langage permettant le développement data-parallèle déclare ses objets sous forme de tableaux. L'algorithme numérique data-parallèle repose principalement sur cette notion de tableaux. Certaines manipulations faisant interagir plusieurs objets, nécessitent des descriptions géométriques de sous-objets, par exemple : la diagonale d'une matrice, la i^{eme} ligne ou colonne, etc.

Connaissant ces structures ou objets géométriques, le programmeur est capable de préciser la façon de les placer les uns par rapport aux autres pour les mettre en rapport et enfin déclencher des opérations de calculs. L'interaction par exemple d'un vecteur avec la colonne d'une matrice permet de déduire l'information d'alignement entre ces deux objets (matrice

et vecteur). Pour permettre au programmeur de réaliser naturellement ces alignements, un modèle de programmation doit offrir un espace virtuel qui servira de référentiel à l'alignement des données (voir par exemple le template de HPF).

Dans un algorithme numérique, le programmeur peut être amené à faire interagir un objet successivement avec plusieurs autres objets, par exemple un vecteur avec chaque colonne d'une matrice. Le modèle qui supporte ces algorithmes doit donc permettre d'exprimer des migrations d'objets à travers le référentiel. Dans le cadre géométrique, ces migrations peuvent se modéliser par des manipulations géométriques appliquées globalement aux objets, par exemple des déplacements ou des rotations. La géométrie sous-jacente à ces manipulations peut être naturellement traduite à l'aide d'un environnement graphique.

C'est dans cet esprit que nous avons mis en œuvre l'environnement HELPDraw qui utilise comme support visuel pour la programmation data parallèle le modèle géométrique HELP (Hyper-Espace et Langages Parallèles) [Laz95]. Ce modèle se base sur la notion d'hyper-espaces et la manipulation d'objets data-parallèles (DPO) à l'intérieur de ces hyper-espaces. Dans ce modèle, les opérations de communication sont clairement séparées des opérations de calcul. Les communications sont traduites par des opérations géométriques appliquées aux DPO (rotation, déplacement, expansion, etc.).

HELPDraw distingue deux niveaux de programmation. Le premier concerne le développement des instructions data-parallèles ou scalaires, et le second permet de développer des blocs d'instructions (des constructions de contrôle data-parallèles et scalaires).

- Concernant le premier niveau, HELPDraw se base particulièrement sur la « programmation par les exemples ». Le programmeur définit dans un éditeur graphique d'opérations géométriques des hyper-espaces dans lesquels il visualise et manipule géométriquement des exemples de DPO. Les opérations qu'il applique peuvent être paramétrées avec des exemples de valeurs afin de permettre leur visualisation. Le code data-parallèle correspondant est ensuite généré; le programmeur l'insère là où il le souhaite dans son programme présent dans un éditeur de texte.

  En plus de l'éditeur d'opérations géométriques et toujours dans ce même niveau de programmation, HELPDraw offre un autre éditeur graphique complémentaire pour faciliter le développement des instructions : l'éditeur graphique d'expressions.

- Le deuxième niveau de programmation concernant donc les blocs d'instructions est assuré à travers l'éditeur graphique de blocs d'instructions. Celui-ci permet au programmeur de choisir le bloc qu'il veut développer (sélection d'un icône), ensuite toutes les instructions développées viennent automatiquement s'insérer dans ce bloc.

Nous présentons, dans ce chapitre, le modèle de base HELP. Nous décrirons ensuite l'en-

vironnement HELPDraw. Nous montrerons en particulier le développement des instructions de base puis dans un niveau d'abstraction plus élevé les blocs d'instructions.

1 Le modèle de programmation data-parallèle géométrique HELP

Le modèle géométrique HELP utilise la notion d'hyper-espace. Le programmeur définit un ou plusieurs hyper-espaces. Les objets data-parallèles (DPO) qu'il va manipuler sont positionnés dans ces hyper-espaces. Dans ce modèle, les transferts de données sont clairement séparés des traitements. On distingue ainsi deux niveaux de programmation : un niveau microscopique dans lequel sont déclenchés les calculs locaux aux points actifs de l'hyper-espace, et un niveau macroscopique qui met en œuvre les opérations géométriques de migration de données.

Le modèle HELP a été mis en œuvre dans notre équipe. Ce sont les travaux de Dominique Lazure exposés dans sa thèse [Laz95].

1.1 Les hyper-espaces

La manipulation d'objets qui interagissent est interne à un hyper-espace. Celui-ci est un ensemble géométrique de points dans lequel les objets parallèles (matrices, vecteurs ...) seront positionnés et manipulés. L'hyper-espace est défini comme un référentiel cartésien de points de coordonnées strictement positives. Le point de l'hyper-espace est l'entité au sein de laquelle se fera le calcul correspondant aux éléments locaux appartenant aux DPO. L'hyper-espace joue le rôle d'une machine virtuelle dont les processeurs représentent les points du référentiel.

La sémantique basée sur le référentiel permet de séparer l'espace d'activité et les références aux éléments d'objets qui interagissent. L'espace d'activité n'est plus déterminé en fonction des index de ces éléments. Le modèle d'exécution induit par cette sémantique consiste d'abord à définir l'activité sur certains points du référentiel, ensuite d'évaluer localement en tout point actif, l'expression data-parallèle courante. Ainsi, l'indice des opérandes présents sur ce point n'a plus de signification, le processeur virtuel opère sur les données présentes localement. Ceci fait qu'une expression calculatoire n'engendre aucune communication implicite.

Un même algorithme peut définir plusieurs hyper-espaces, de taille et de forme différentes, en fonction de sa spécificité. Le programmeur peut faire évoluer ses données dans un espace à une, deux dimensions, ou plus. La taille de chacune des dimensions est définie selon l'algorithme à réaliser.

1.2 Les objets data-parallèles

Les DPO sont alloués sur des ensembles compacts de points de l'hyper-espace (figure III.1). Ils représentent les sous-espaces sur lesquels vont pouvoir s'effectuer les traitements. Leur position et leur forme peuvent être modifiées, mais l'association DPO-hyper-espace est figée. Chaque point de l'hyper-espace est virtuellement le support mémoire d'un élément de l'objet.

Un DPO est par défaut dynamique. Il peut changer de position, de taille et de forme. Ceci permet au programmeur de faire évoluer ses données librement à travers le temps et l'hyper-espace. Le domaine d'allocation varie par l'application d'une opération d'association (*cf.* 1.3.4).

HELP définit les objets réguliers (vecteur, matrice, ...) et un type d'objet moins régulier : les diagonales. Pour un plan donné, seules les diagonales parallèles à la diagonale principale peuvent être définies. Nous verrons plus loin (*cf.* III.1.5), que HELPDraw permet de manipuler d'autres objets moins réguliers : les triangles.

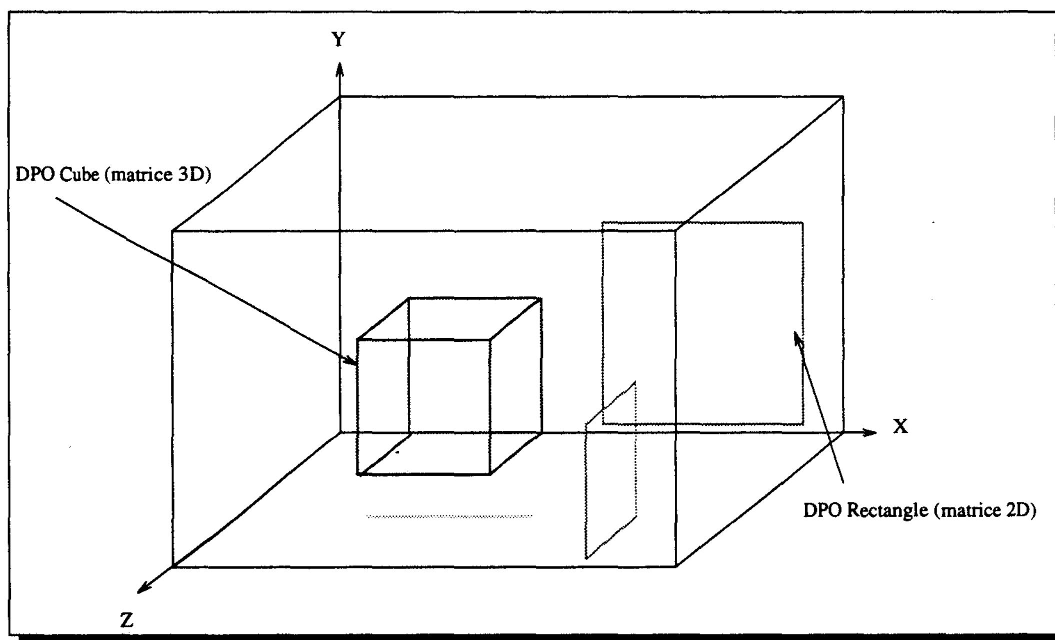


FIG. III.1 - *Exemples de* DPO *dans un Hyper-Espace 3-D*

1.3 Niveau microscopique : les opérations calculatoires

Les traitements microscopiques sont locaux aux points de l'hyper-espace. Ils consistent en l'application de fonctions ou opérations arithmétiques et logiques sur les éléments de DPO appartenant au même point. Le domaine sur lequel les calculs vont être déclenchés est distingué

des objets intervenant dans ces calculs. Ce domaine est appelé *domaine de conformité.*

1.3.1 Domaine de conformité

Dans le modèle HELP, une expression data parallèle est vue comme une hiérarchie de segments conformes. Un segment conforme est une séquence d'opérations data-parallèles calculatoires qui vérifient la règle de conformité. Celle-ci est respectée lorsque tous les objets intervenant dans le segment conforme recouvrent un même ensemble de points. Cet ensemble est nommé domaine de conformité du segment, il est spécifié par le programmeur (implicitement ou explicitement).

Une expression est vue comme un arbre de segments conformes. Cet arbre est construit selon les trois règles suivantes :

1. on appelle segment conforme principal d'une expression, le segment conforme qui apparaît au niveau haut de l'arbre de segments conformes formant l'expression ;
2. l'appel d'opérations géométriques (de communications) crée un segment conforme de niveau inférieur ;
3. un domaine de conformité est associé à tout segment conforme ;

La figure III.2 montre un exemple concret d'une hiérarchie de segments conformes. (À noter que nous avons représenté l'application d'une opération macroscopique par le constructeur point « . ». C'est cette syntaxe qu'utilise HELPDraw et C-HELP.)

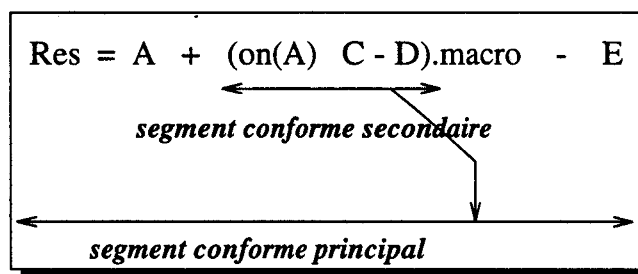


FIG. III.2 - *Exemple d'hiérarchie de segments conformes*

Pour préciser le domaine de points sur lequel un segment conforme est évalué, le programmeur spécifie l'espace géométrique de cette évaluation à l'aide de constructeur `on`; on parle alors de domaine explicite ou contraint (*cf.* 1.3.2). HELP permet également de masquer certains points du domaine de conformité à l'aide du constructeur `where`; on parle alors de domaine masqué (*cf.* 1.3.3).

1.3.2 Domaine contraint : constructeur « on »

Le constructeur on permet de contraindre le domaine de conformité au domaine d'allocation du DPO argument (on(dpo)), pour le segment conforme dans lequel il apparaît. Les objets qui interagissent sous le contrôle d'un on(dpo) doivent englober dpo, *cf.* figure III.3 (à gauche). Il est possible de réduire successivement le domaine contraint par l'imbrication de constructeurs on. Le domaine contraint résultant est le domaine d'allocation du DPO argument du dernier on.

Il est possible également d'utiliser le on sur une instruction d'affectation. Dans ce cas, le domaine contraint ne s'applique que sur le segment principal de l'expression en partie droite de l'opérateur d'affectation, *cf.* figure III.3 (à droite).

Le constructeur on peut être étendue à un bloc d'instructions. Pour chaque instruction, le segment conforme principal est alors contraint au domaine de conformité explicité.

À noter que lorsqu'un segment conforme ne comporte pas de on, tous les DPO qui apparaissent dans le segment doivent être alloués sur les mêmes points de l'hyper-espace.

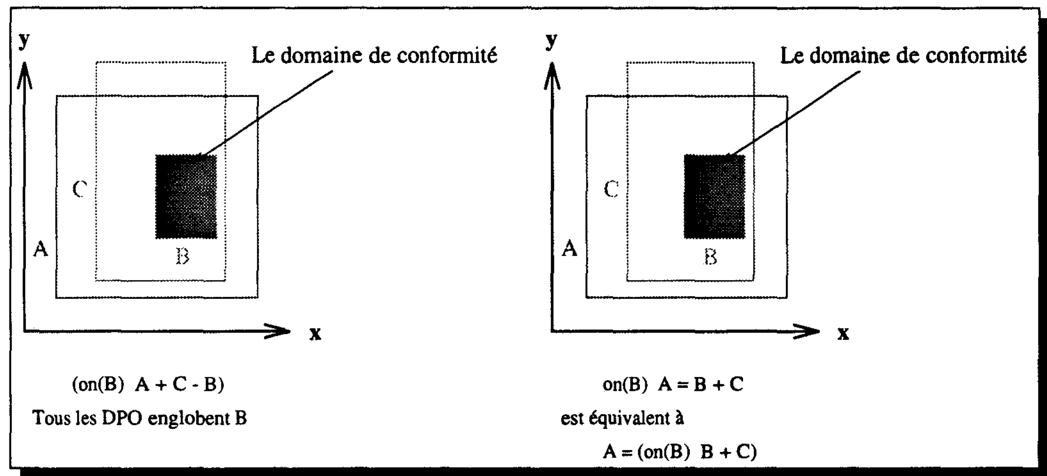


FIG. III.3 - *Exemples d'application du constructeur on*

1.3.3 Domaine masqué : constructeur « where »

HELP permet à l'aide du constructeur **where** de réduire l'activité à un sous-ensemble de points du domaine de conformité. Les points du domaine pour lesquels le résultat de l'évaluation de l'expression data-parallèle « expr_DP » (du where(expr_DP)) est vraie, définissent le domaine d'activité. Dans le cas général, un segment conforme est composé d'un constructeur on définissant le domaine de conformité, puis d'un constructeur **where** définissant le domaine

d'activité.

Le **where** n'a aucune influence sur les segments conformes inférieurs, dont le résultat intervient dans le segment conforme soumis au constructeur **where**. Lorsqu'il est appliqué à une instruction d'affectation, le masque ne s'applique que sur le segment conforme principal de l'expression en partie droite. Les affectations se font sur les points du domaine d'activité de cette évaluation du segment principal de l'expression.

Comme le **on**, le **where** peut s'appliquer sur un bloc d'instructions (il s'applique à chaque instruction). Il peut y avoir aussi plusieurs masques sur un même segment conforme, dans ce cas l'évaluation devient la conjonction de toutes les conditions data-parallèles.

1.3.4 Association

HELP donne au programmeur la possibilité de manipuler des objets dont l'allocation (taille, forme, position) peut changer au cours de l'exécution. Pour ce faire, il applique l'opérateur d'association symbolisé par « $\leftarrow$ ». L'association s'applique sur un DPO, par exemple :

```
dpoRes <- expr_DP
```

L'expression data-parallèle de droite (**expr_DP**) doit vérifier la règle de conformité. L'opération permet de réallouer l'objet « **dpoRes** » sur ce domaine de conformité et lui affectera les valeurs de l'expression.

1.3.5 Affectation injective

L'instruction d'affectation, spécifiée par « = », ne modifie pas le domaine d'allocation du DPO en cours d'affectation. Une copie du résultat de l'évaluation de l'expression en partie droite est effectuée sur chacun des points qui ont fait l'objet de cette évaluation. Le DPO en partie gauche doit englober l'ensemble de ces points. On parle ainsi d'injections.

1.4 Niveau macroscopique : les opérations géométriques

À travers les opérations géométriques, HELP permet aux objets d'être complètement dynamiques ; ils peuvent changer de position, de taille, et de nombre de dimensions. L'expression géométrique offre un vision globale sur la migration de l'objet, ce qui diffère complètement de la vision sous-jacente à la manipulation des indices que nous retrouvons dans les langages Fortran.

Les communications traduites par les opérations géométriques sont effectuées au niveau macroscopique. Les opérations macroscopiques manipulent globalement les DPO sans déclencher d'opérations microscopiques sur leurs éléments. L'application d'une suite d'opérations géométriques crée un nouveau segment conforme. La règle de conformité s'applique donc sur l'expression qui va faire l'objet de la migration (*cf.* figure III.2), indépendamment de la conformité du segment d'utilisation du résultat de cette migration. Le résultat produit par une opération macroscopique est un DPO temporaire, qui doit entrer en conformité avec le segment supérieur dans lequel il est consommé (*cf.* figure III.4). Les opérations géométriques peuvent s'appliquer sur des DPO variables ou des DPO expressions :

```
(dpo1 + dpo2).suite_de_macro
```

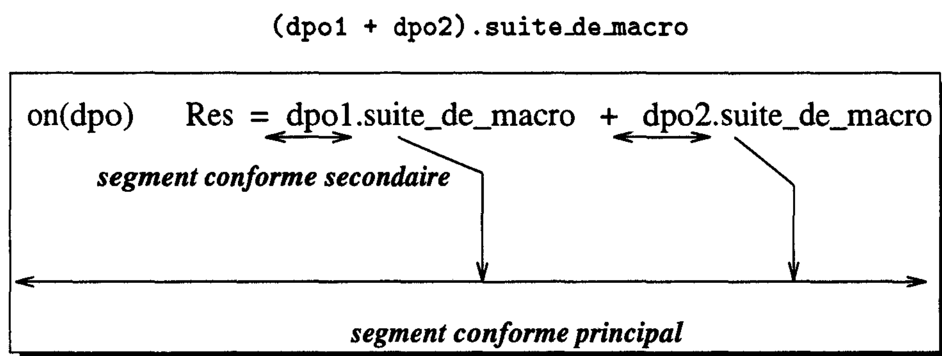


FIG. III.4 - *Application des opérations géométriques*

Le modèle suggère plusieurs opérations géométriques, que nous regroupons selon la cardinalité de l'ensemble des points de l'objet résultat relativement à l'objet source :

1. Les opérations bijectives : elles regroupent les opérations de déplacement et de rotation.
2. Les opérations réplicatives.
3. Les opérations sélectives : où on trouve les opérations d'extraction de sous-objets.
4. Les opérations de réduction.

Ces opérations sont parfois nommées différemment en C-HELP ou en HELPDraw (*cf.* 1.4, chapitre IV : 1.3).

La liste des opérations géométriques retenues pour HelpDraw

L'application d'une suite d'opérations macroscopiques produit un temporaire à partir de la forme, de la position et du contenu du DPO source et de la fonctionnalité des opérations apparaissant dans la liste. Il est cependant interdit d'appliquer une opération géométrique

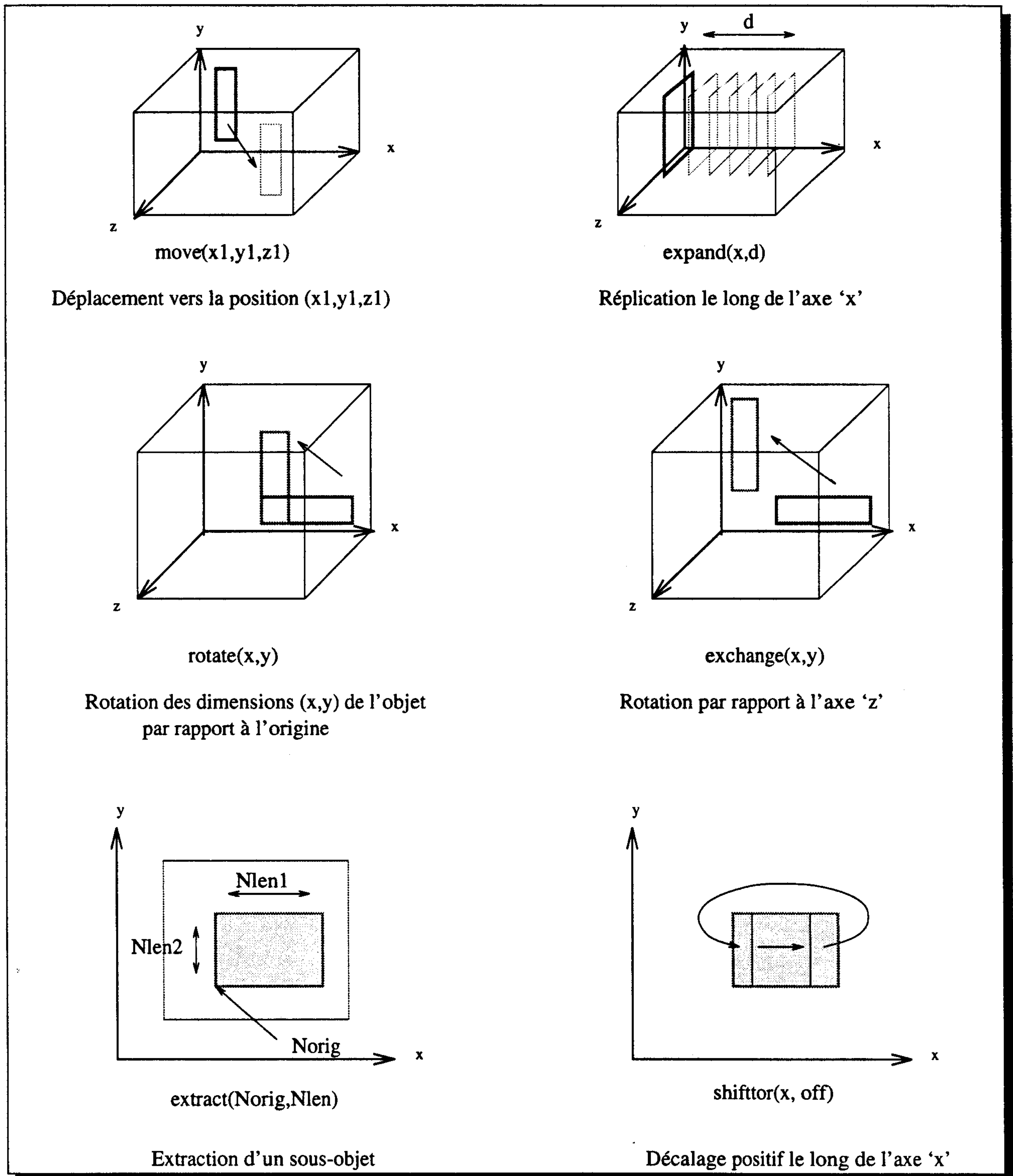


FIG. III.5 - *Exemple d'opérations géométriques*

BU LILLE

qui ne génèrera pas un objet reconnu par le modèle (objet parallépipédique, ou diagonale). Le programmeur ne peut pas par exemple réaliser une expansion d'une diagonale, puisque le DPO résultat n'est pas défini dans le modèle.

Nous retrouvons en HELPDraw les quatre types d'opérations géométriques définis dans le modèle HELP. En voici la description (la figure III.5 montre des exemples d'opérations) :

a/ Les opérations bijectives

Déplacement Cette opération permet de déplacer un objet à l'intérieur de l'hyper-espace (en fait c'est une copie de l'objet qui est déplacée, puisque chaque opération macroscopique crée un temporaire). La forme du DPO résultat est identique à celle du DPO source. Seule l'origine change. (Opération MOVE.)

Rotations Les opérations de rotation créent une copie du DPO source en changeant l'orientation par rapport aux dimensions. L'opération EXCHANGE échange les dimensions de l'objet et en même temps les coordonnées de l'origine (rotation par rapport à un axe). Le temporaire créé par cette opération ne sera pas donc forcément alloué sur les mêmes points que ceux du DPO source. Les autres opérations que nous allons voir ne changent par contre pas l'origine. L'opération ROTATE opère une rotation par rapport à l'origine du DPO source. Seules les dimensions sont interchangées. Deux autres opérations concernent les diagonales : ROTATE_TODIAG opère une rotation d'une ligne vers une diagonale et ROTATE_TOLINE fait l'inverse, de diagonale à une ligne.

Décalages L'opération CIRCULAR_SHIFT opère un décalage torique le long d'une dimension donnée de longueur `off`. Lorsque `off` est positif (resp. négatif), le décalage se fait vers les coordonnées croissantes (resp. décroissantes). L'opération « `mirror` » effectue une opération de miroir du DPO source sur son intervalle d'allocation le long de la dimension `dim` (le premier élément est échangé avec le dernier, le second avec l'avant dernier, etc.). Le décalage ne s'applique pas aux diagonales.

b/ Les opérations de réplication

L'opération de réplication ou d'expansion EXPAND permet de copier un objet le long de toute une dimension ou seulement d'une distance « d » (d copies). Les formes géométriques des DPO source et résultat ne sont pas identiques. Par exemple, l'expansion d'une ligne donne un DPO rectangle dont le contenu est n fois cette ligne.

c/ Les opérations d'extraction

Ces opérations permettent d'extraire une partie du DPO source. Le résultat est un DPO dont la forme est définie dans le modèle. Le programmeur peut extraire un sous-objet rectangulaire (EXTRACT_SUBDPO), une ligne parallèle à un des axes de l'hyper-espace (EXTRACT_LINE), ou une diagonale (EXTRACT_DIAG).

d/ Les opérations de réduction

Ces opérations permettent d'opérer des réductions le long d'une dimension. La réduction donne un DPO de longueur 1 sur la dimension concernée, de coordonnée égale à celle de l'origine du DPO source sur cette dimension. L'opération REDUCEADD (resp. REDUCEMUL, REDUCEOR, REDUCEAND, REDUCEMAX, REDUCEMIN) calcule la somme (resp. le produit, la somme logique, le produit logique, le maximum, le minimum) des éléments du DPO source sur la dimension `dim` spécifiée.

Nous avons décrit le modèle géométrique data-parallèle HELP. Ce modèle offre un support géométrique à la programmation data-parallèle adapter à la visualisation : des hyper-espaces (1, 2 ou 3-D), des formes géométriques des DPO ainsi que des manipulations géométriques de ces DPO (déplacement, changement de forme, etc).

Ceci dit, en ce qui concerne HELPDraw, nous avons préféré de ne pas être complètement dépendant des définitions de HELP, c'est pourquoi il y a eu des extensions par rapport au modèle HELP.

1.5 Les triangles

Au delà de ce qui a été défini dans le modèle HELP, HELPDraw propose la manipulation d'une nouvelle forme d'objets data-parallèles, ceci dans le but d'offrir le plus d'abstraction possible au programmeur. Nous définissons un nouveau type de formes géométriques: les triangles.

Nous sommes partis du fait que le programmeur peut être amené dans plusieurs applications scientifiques à manipuler des matrices triangulaires. Tel que nous avons défini HELP, la manipulation de ce type d'objets peut se faire par l'utilisation d'un objet rectangulaire auquel il faut appliquer un domaine d'activité. Or, l'utilisateur préfère certainement prendre directement un triangle et le manipuler en tant que tel. C'est pourquoi HELPDraw propose ce type d'objets. Notre ambition n'est pas d'offrir des objets qui ne sont définis dans aucun

langage data-parallèle, mais d'éviter au programmeur de gérer lui-même les masques à appliquer à des objets réguliers. HELPDraw fait donc abstraction de cette gestion ; le programmeur manipule simplement et directement un triangle.

HELPDraw accepte seulement trois formes de triangles sur un plan. La figure III.6 montre les trois formes que nous pouvons définir dans un plan [xy]. L'autre forme est éliminée parce qu'on ne lui connaît pas d'origine valable.

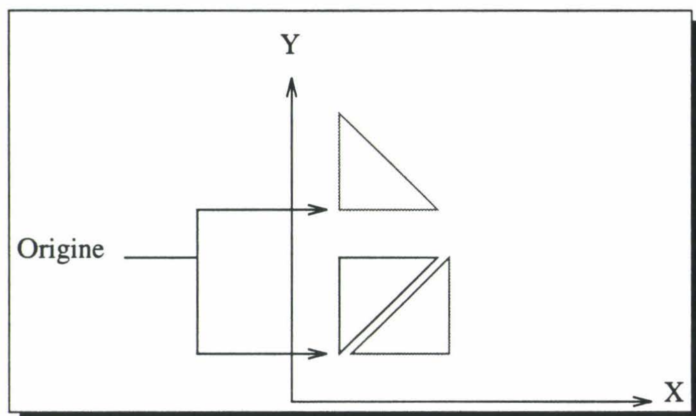


FIG. III.6 - *Les trois formes de triangles acceptées dans un plan donné*

Les opérations géométriques qu'on peut appliquer aux triangles sont :

Déplacement Le programmeur dispose de l'opération de déplacement (MOVE) qui s'applique comme pour tout autre objet.

Rotation Il est possible également d'appliquer des rotations de triangles (opération ROTATE_TRIANGLE). Les seules opérations autorisées à ce niveau sont celles qui correspondent aux opérations ROTATE du carré englobant le triangle.

Extraction Le programmeur peut extraire un sous-objet triangle : opération (EXTRACT_TRIANGLE).

Aucune autre opération géométrique n'est acceptée sur les triangles.

1.6 Un exemple d'algorithme data-parallèle conçu selon le modèle HELP

Soit le vecteur $V(n)$, on veut calculer la matrice $M(n \times n)$, telle que :

$$M(i,j) = V(i) + V(j)$$

Dans le modèle HELP, le programmeur doit pouvoir calculer tous les éléments de M en parallèle et sans communications implicites. Pour cela, il lui suffit de placer avec chaque

élément M(i,j) les deux autres éléments : V(i) et V(j) ; les trois éléments doivent être ensembles sur le même point d'un hyper-espace.

Le programmeur crée un hyper-espace (n×n) où il alloue deux DPO : M(n×n) et V(n), *cf.* figure III.7(1). Pour aligner les éléments V(i) avec les M(i,j), il effectue un échange de dimensions (x,y), *cf.* figure III.7(2). Il obtient un nouveau DPO (temporaire) qu'il répliquera n fois selon l'axe « x » de l'hyper-espace, *cf.* figure III.7(3). De la même manière pour obtenir les V(j) avec les M(i,j), il réplique le DPO V n fois selon l'axe « y », *cf.* figure III.7(4). À la suite de ces opérations géométriques, on retrouve 3 DPO conformes (même origine et même taille) *cf.* figure III.7(5). Il suffit ensuite d'appliquer les opérations microscopiques : somme des deux temporaires et affectation du résultat à la matrice M.

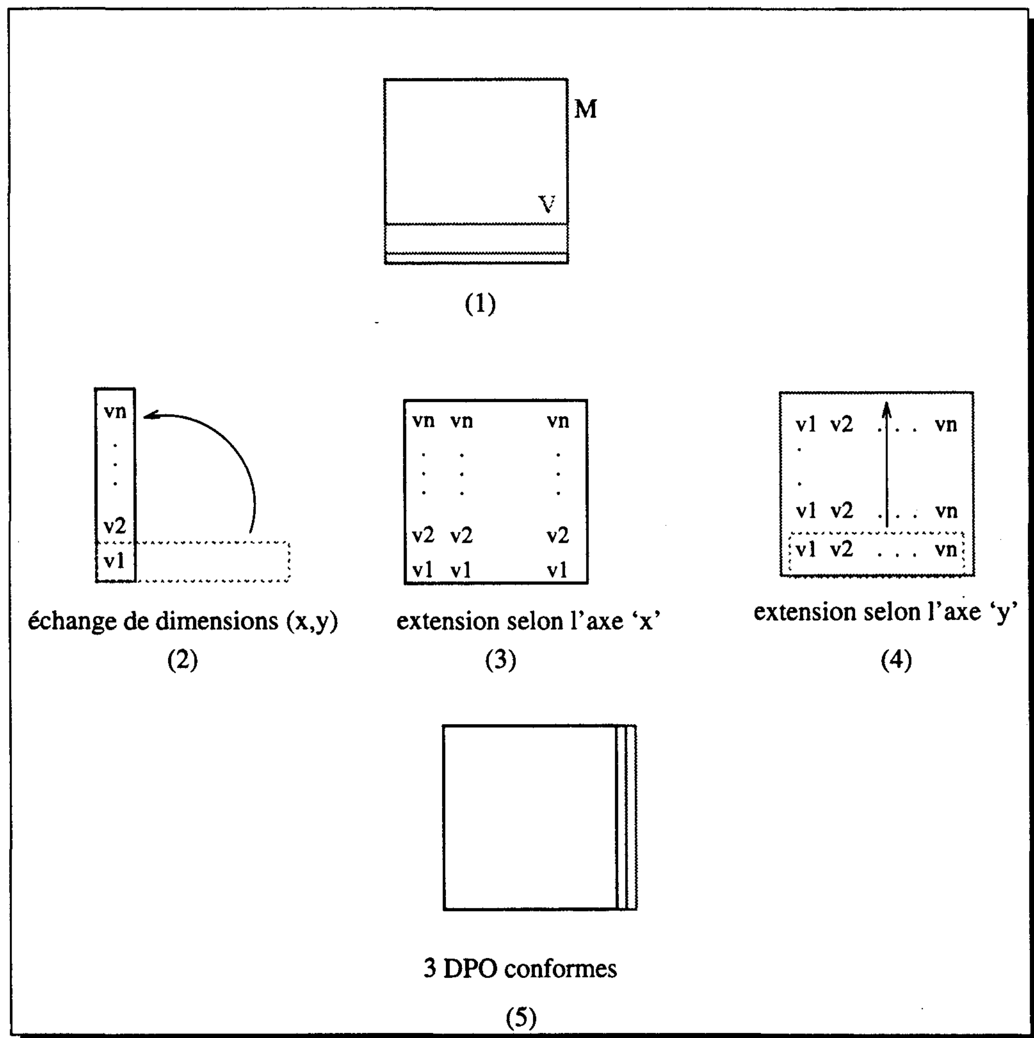


FIG. III.7 - *Un exemple de conception selon le modèle HELP: M(i,j) = V(i)+V(j)*

2 Développement des instructions de base

La technique « *programmation par les exemples* » est utilisée au plus bas niveau de programmation. Elle permet de développer les instructions élémentaires du programme : les expressions arithmétiques ou logiques, comprennent éventuellement des appels de fonctions, des opérations géométriques ainsi que des opérations microscopiques. Ces instructions manipulent aussi bien des scalaires que des objets data-parallèles.

Rappelons la définition de la « *programmation par les exemples* » que nous avons étudiée au second chapitre (*cf.* 3.1.5, page 59). Cette technique permet au programmeur de réaliser des actions sur des exemples d'objets concrets (souvent par manipulation directe), et construit en même temps un programme abstrait.

Nous avons choisi d'utiliser cette technique afin de permettre à l'utilisateur de développer son programme selon le modèle géométrique HELP. Il construit les instructions data-parallèles par la manipulation directe de DPO à l'intérieur des hyper-espaces. La programmation par les exemples nous permet de construire des instructions en manipulant géométriquement et de manière directe des objets, tout en regardant en même temps l'interprétation « visuel » de l'exécution. La conjugaison des deux aspects simultanés, développement et exécution visuelle, assiste d'une part l'utilisateur dans sa conception et lui permet d'autre part de mieux contrôler son programme, puisque celui-ci est virtuellement interprété au fur et à mesure.

La figure III.8 montre le processus suivi par le programmeur pour développer des instructions data-parallèles. Selon la spécificité de l'algorithme, le programmeur peut définir un (ou plusieurs) hyper-espace(s) dans lequel il alloue les DPO nécessaires. Ces définitions se font à la demande : le programmeur n'est pas obligé de créer tous les DPO à la fois. Pour construire ensuite une instruction, le programmeur peut opérer des opérations géométriques sur les DPO. Après chaque opération, HELPDraw crée un DPO temporaire représentant le résultat (si l'objet sur lequel a été appliquée l'opération est lui même temporaire, il sera alors consommé). Les DPO variables ou temporaires peuvent être ensuite combinés par des opérations microscopiques pour construire l'expression voulue. Une fois l'expression réalisée, elle est soit insérée dans un bloc d'instructions, soit insérée directement dans un éditeur de texte avec le reste du programme. De même, lorsqu'il s'agit d'un bloc d'instructions, une fois réalisé il est inséré à son tour dans le programme.

Dans cette section, nous détaillerons chaque étape ainsi que les moyens offerts par HELPDraw pour la réaliser. Nous verrons également la manipulation des scalaires et la réalisation d'instructions scalaires.

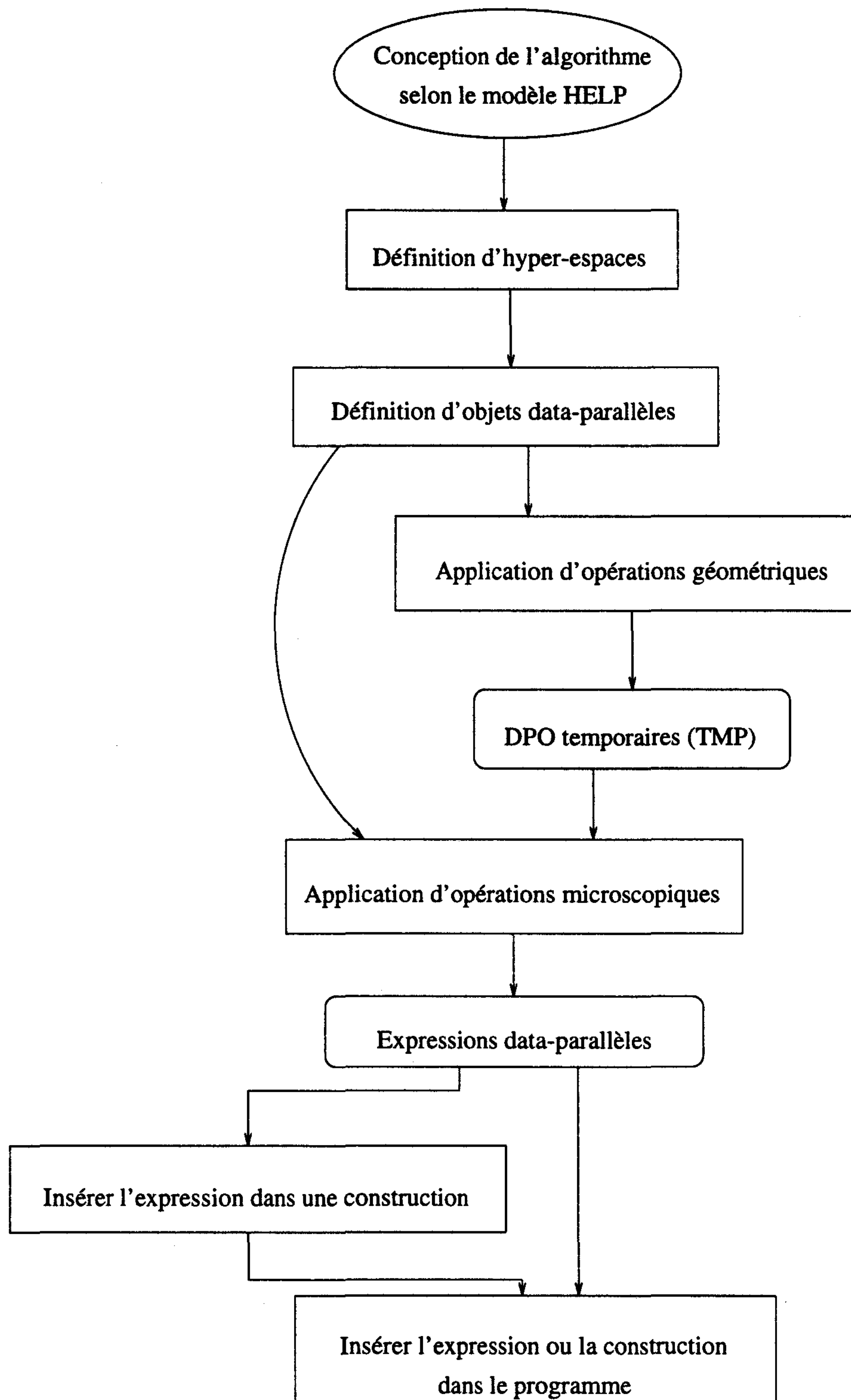


FIG. III.8 - *La chaîne de développement d'une instruction data-parallèle*

2.1 HELPDraw non contextuel

Nous tenons à préciser que HELPDraw ne permet pas le développement d'un programme complet. Il ne permet pas le développement de fonctions, la gestion des paramètres d'entrées/sorties, etc. Il permet le développement d'instructions ou de blocs d'instructions, et c'est le seul contexte qu'il contrôle.

2.2 Définition d'hyper-espaces

En fonction de l'algorithme conçu selon le modèle HELP, le programmeur peut être amené à définir un ou plusieurs hyper-espaces. Pour définir un hyper-espace, il introduit les informations nécessaires à travers une boîte de dialogue. Deux informations sont indispensables : le nom et les dimensions de l'hyper-espace. La figure III.9 montre un exemple de définition d'un hyper-espace 3-D « my_hspace » de taille (500 × 500 × 500).

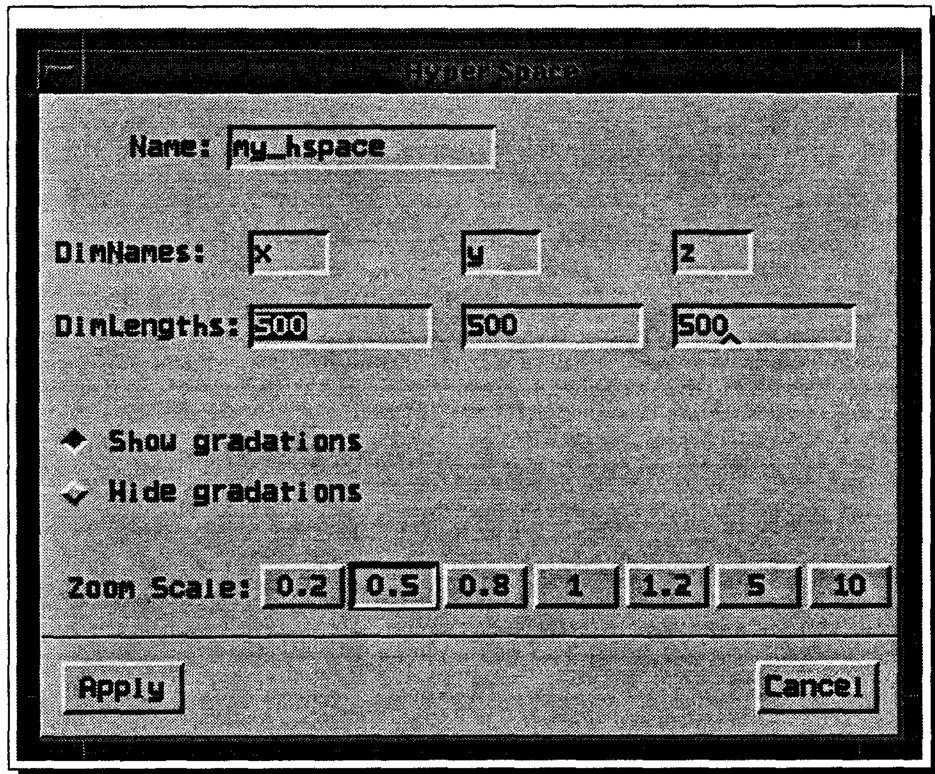


FIG. III.9 - *Un exemple de définition d'un hyper-espace*

HELPDraw offre également la possibilité de naviguer à travers les différents hyper-espaces définis par le programmeur. Lorsqu'un hyper-espace est sélectionné de nouveau, HELPDraw le visualisera dans l'état exact où il était auparavant.

Il reste cependant un problème important qu'est la résolution de la représentation. La représentation de l'hyper-espace est limitée par la taille de la fenêtre où il est défini. Dès que la taille de l'hyper-espace dépasse 500 points, il devient impossible de tout représenter avec une résolution un point par pixel. Pourtant cette précision est nécessaire, surtout lors des manipulations géométriques de DPO. Pour y remédier, HELPDraw offre deux solutions complémentaires. La première se base sur l'échelle de représentation (zoom) : elle permet au programmeur de réduire la vue de l'hyper-espace pour qu'elle soit globale, ou au contraire l'agrandir pour avoir la précision nécessaire au niveau d'une partie de l'hyper-espace. La deuxième solution est liée aux opérations géométriques : elle permet, indépendamment de l'échelle de représentation, de se déplacer dans l'hyper-espace avec un pas de un point. Cette solution sera présentée dans la description des opérations géométriques.

Le nombre de dimensions de l'hyper-espace peut être quelconque, mais HELPDraw ne permet de visualiser qu'au plus trois dimensions à la fois. Pour mettre en œuvre plus de 3-D, il suffit que HELPDraw permette au programmeur de préciser les trois dimensions à visualiser (par défaut il considère les trois premières) ; les autres dimensions seront cachées. Pour définir par exemple un hyper-espace 4-D, le programmeur attribue un nom à chaque dimension (exemple : x,y,z et t) : il choisit ensuite à partir de ces noms la combinaison des trois dimensions à visualiser (exemple : x,y,z). La manipulation de DPO ne se fera que par rapport à ces dimensions. Le programmeur peut à n'importe quel moment changer la combinaison des dimensions à visualiser : il choisit à travers une boîte de dialogue une nouvelle combinaison (exemple : x,y,t). Si cet hyper-espace contient des DPO, leur représentation changera également : le programmeur verra cette fois la partie (x,y,t) au lieu de (x,y,z).

2.3 Définition des objets data-parallèles

Les objets data-parallèles sont eux aussi définis à travers une boîte de dialogue (*cf.* III.10). Un DPO est déclaré par son nom, sa taille et son origine par rapport au référentiel.

Les formes de DPO acceptées par HELPDraw sont des lignes parallèles à un des axes du référentiel, des diagonales principales, des rectangles, des triangles rectangles isocèles, ou des parallélépipèdes rectangles. À part les triangles et les diagonales, les autres formes sont définies par leur origine relative au référentiel (une coordonnée pour chaque dimension de l'hyper-espace) ainsi que leur taille (la longueur sur chaque dimension de l'hyper-espace). Pour un triangle, le programmeur choisit d'abord (dans une fenêtre à part) une des trois orientations possibles dans un plan donné (*cf.* figure III.11). Il précise ensuite, dans un même type de boîte que celui des DPO réguliers, l'origine du triangle (une coordonnée sur chaque dimension) et la longueur d'un côté. La déclaration d'une diagonale se fait presque de la même manière, le programmeur choisit le plan où sera allouée la diagonale, puis indique dans une boîte de dialogue l'origine et la longueur de la diagonale.

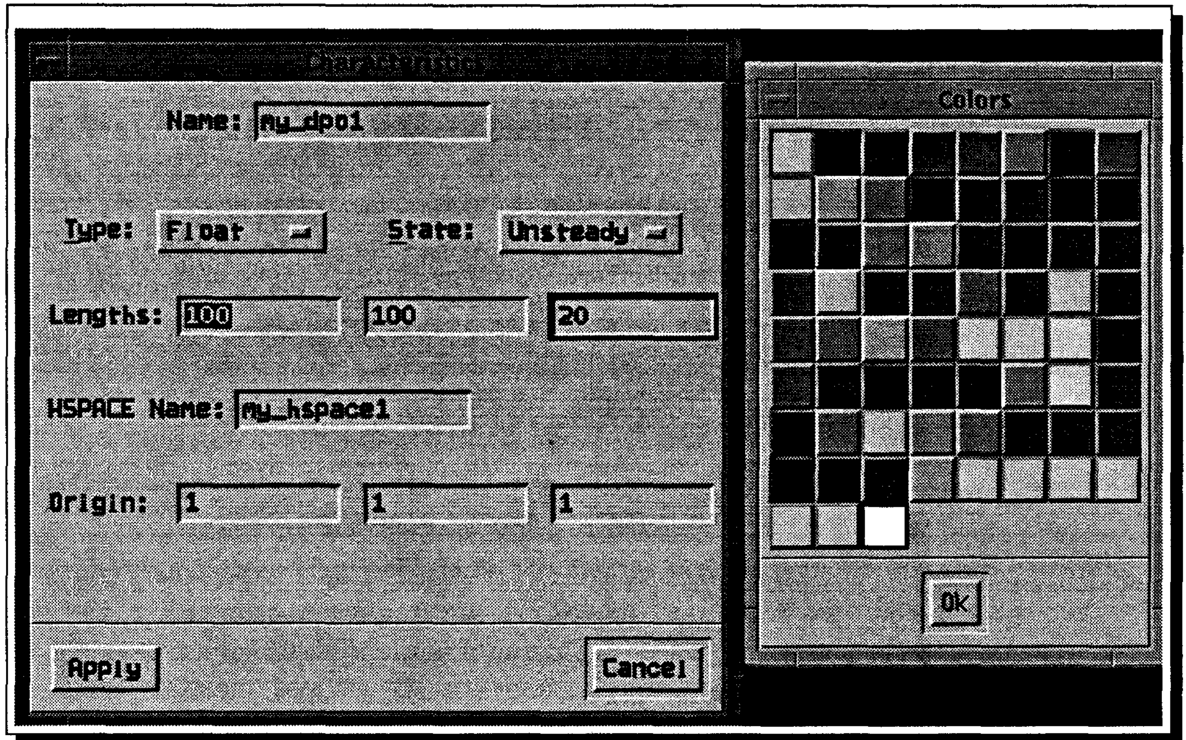


FIG. III.10 - *Un exemple de déclaration de* DPO *réguliers*

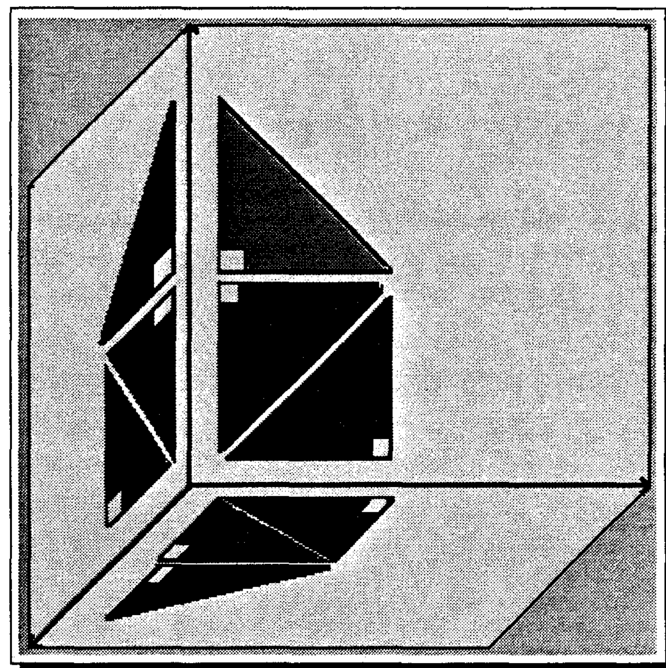


FIG. III.11 - *Les icônes permettant de choisir la forme du triangle*

Un DPO est défini par défaut comme dynamique (**UNSTEADY**) c'est à dire qu'il peut changer de taille ou de position. Si par contre le programmeur sait que le DPO ne va pas subir de changement, il le déclarera comme fixe (**STEADY**) ; la génération de code est simplifiée dans ce cas, en particulier pour des langages tels que HPF.

Le programmeur peut définir autant de DPO qu'il veut dans un même hyper-espace. Lorsqu'ils sont nombreux ou définis aux mêmes emplacements, il est souvent très difficile de les distinguer. De ce fait, HELPDraw offre la possibilité d'associer à chaque DPO une couleur que le programmeur précisera lors de la déclaration (*cf.* figure III.12).

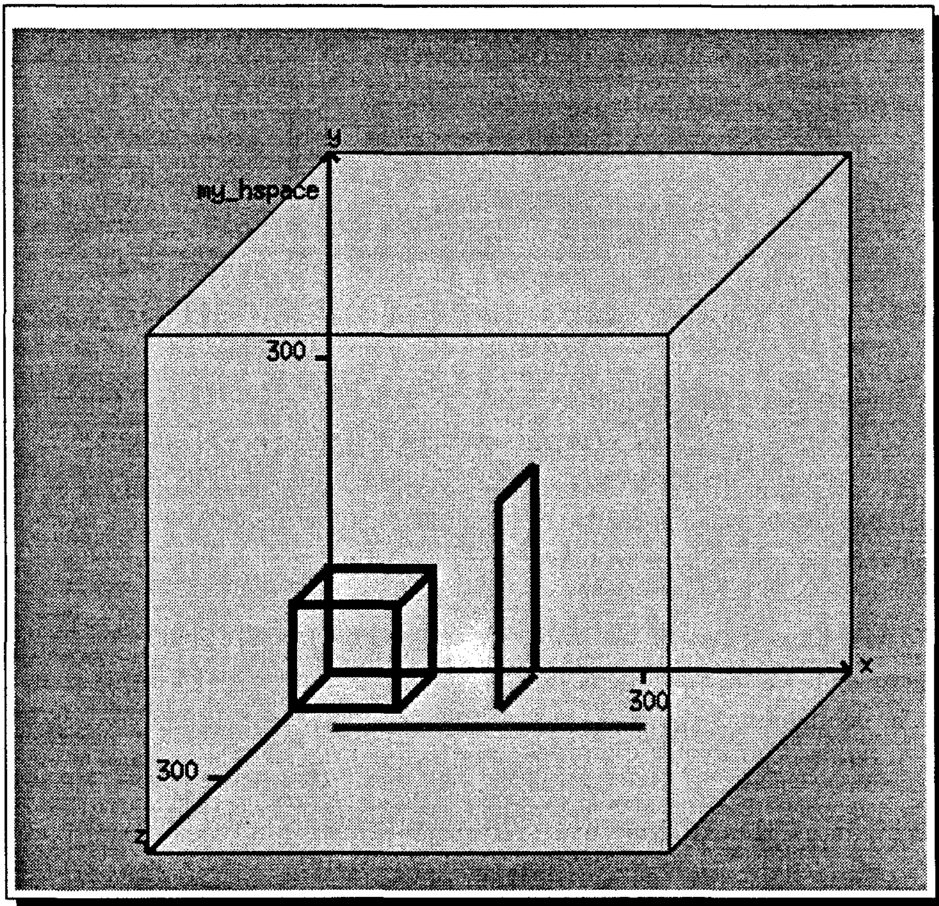


FIG. III.12 - *Exemples de* DPO *définis dans un hyper-espace*

Un DPO peut être directement sélectionné ou dé-sélectionné en cliquant dessus. Un autre problème pourrait apparaître lorsque le programmeur veut sélectionner un DPO parmi plusieurs, en particulier lorsqu'ils sont juxtaposés. Ainsi pour permettre une sélection facile et conviviale, HELPDraw représente dans une liste (DPO_List) le nom du DPO ainsi que la couleur qui lui est associée (*cf.* figure III.13). En cliquant simplement dans cette liste avec la souris, l'utilisateur peut sélectionner ou dé-sélectionner un DPO. Toujours dans le souci d'une meilleure visibilité, des moyens sont offerts au programmeur pour cacher temporairement les

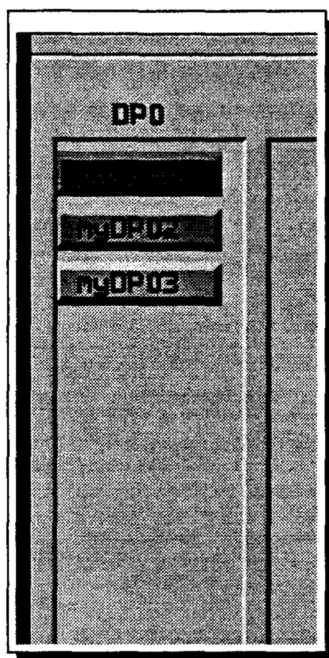


FIG. III.13 - *La liste des* DPO *: une partie de l'éditeur d'opérations géométriques*

DPO qu'il n'utilise pas fréquemment.

À noter également que lorsque le nombre de dimensions de l'hyper-espace est supérieur à trois, HELPDraw ne visualisera du DPO que la partie correspondant à la combinaison de dimensions choisies pour l'hyper-espace.

La figure III.14 est la partie principale de ce que nous avons appelé « éditeur d'opérations géométriques » où l'utilisateur peut définir les hyper-espaces et les DPO. Dans ce qui suit, nous verrons comment ces derniers sont manipulés à l'intérieur de l'hyper-espace.

2.4 Réalisation des opérations géométriques

Le programmeur peut appliquer l'ensemble des opérations géométriques définies dans le modèle géométrique. Celles-ci sont réalisées dans l'éditeur d'opérations géométriques de HELPDraw. Cet éditeur est le cœur de l'environnement ; son objectif essentiel est d'assister le programmeur dans la conception de son algorithme data-parallèle de manière interactive et conviviale. Le programmeur développe pas à pas son algorithme en observant l'interprétation graphique de chaque manipulation géométrique appliquée aux DPO.

Le programmeur peut appliquer ces opérations soit à travers des menus déroulants soit par manipulation directe des objets. La figure III.15 montre un exemple d'opération géométrique exécutée à partir d'un menu (l'opération **EXCHANGE**). La manipulation directe est une

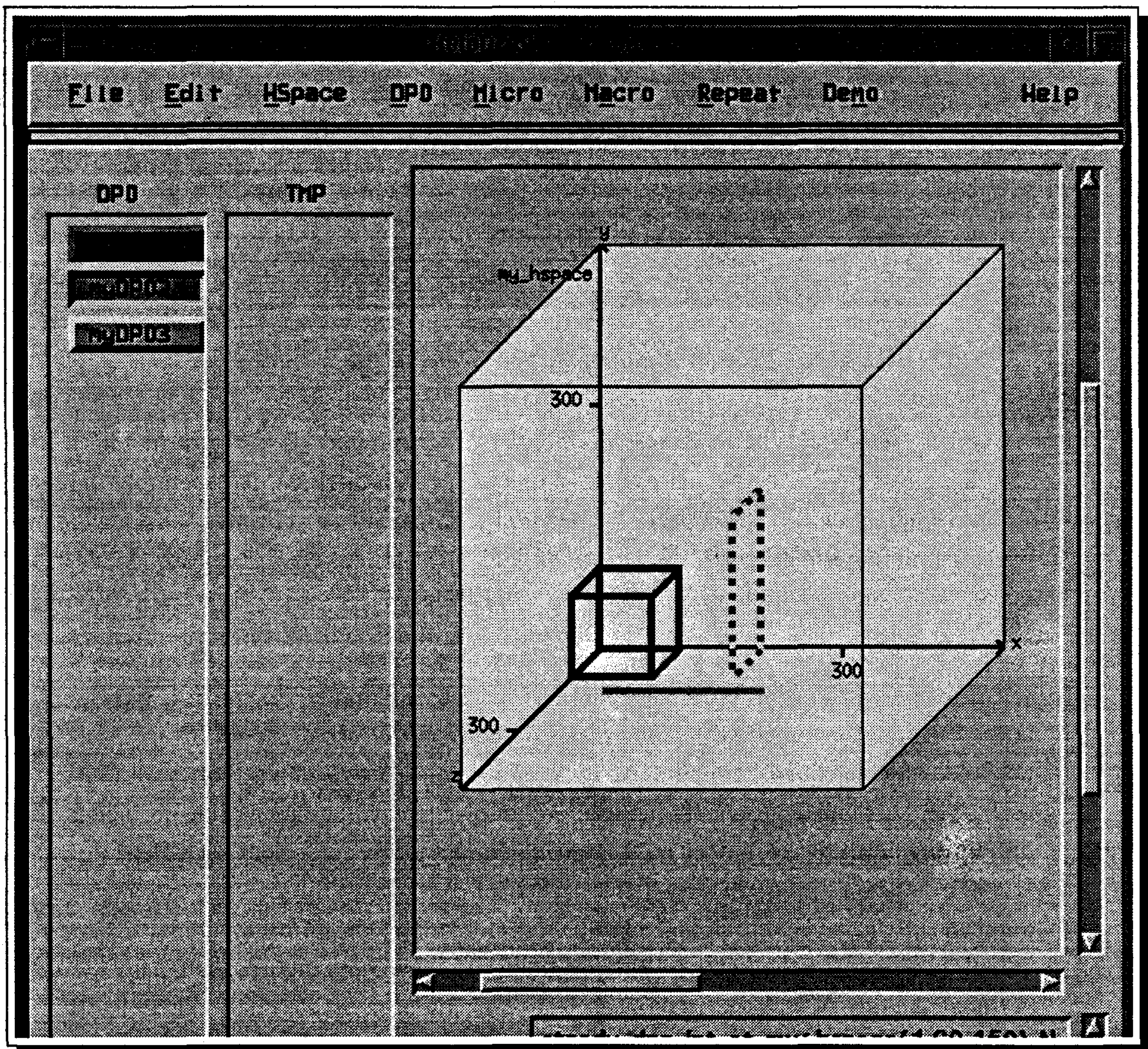


FIG. III.14 - *La partie principale de l'éditeur d'opérations géométriques*

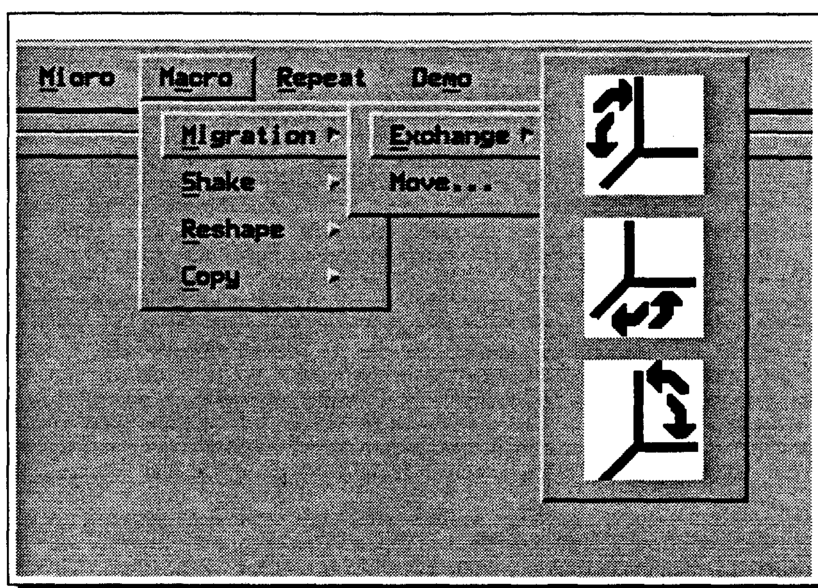


FIG. III.15 - *Application d'une opération géométrique à travers un menu*

des caractéristiques de HELPDraw ; elle permet au programmeur d'opérer de manière plus naturelle. Il pointe sur un DPO puis exécute l'opération géométrique, par exemple déplacer un objet à l'intérieur de l'hyper-espace. Les exemples d'opérations suivantes illustrent mieux cet aspect de manipulation directe. Ils montrent également les cas où cette méthode n'est pas très appropriée (exemple : dans une opération de rotation).

Déplacements C'est l'exemple le plus simple. L'utilisateur pointe sur un DPO puis le glisse là où il souhaite. En fait c'est une copie de ce DPO qui est déplacée. Deux problèmes peuvent être rencontrés cependant. Le premier concerne le déplacement de l'objet en 3-D, et le deuxième concerne la précision du déplacement. (1) Le premier problème vient du fait que la souris se déplace en 2-D sur un écran qui est également 2-D. Pour y pallier, HELPDraw simule le déplacement en 3-D. Il offre la possibilité au programmeur de se déplacer alternativement sur un plan [x,y] (donc z fixe) ou en profondeur c'est à dire z varie mais « x,y » restent fixent. (2) Le second problème, que nous avons déjà évoqué dans la définition des hyper-espaces, concerne la précision permettant de retrouver la position de destination exacte. Ceci vient de l'échelle (point/pixel) de représentation de l'hyper-espace. Un pixel de l'écran peut regrouper plusieurs points de l'hyper-espace. Pour pallier à cet handicap, HELPDraw définit des touches-clavier permettant un déplacement par pas de 1 ou 10 points. Avec la souris, l'utilisateur déplace (rapidement) l'objet vers une zone proche de la position finale, puis l'ajuste avec les touches.

Rotations L'opération « EXCHANGE » permet de faire une rotation par rapport à un axe, alors que l'opération « ROTATE » le fait par rapport à l'origine du DPO. Le programmeur choisit d'abord le type d'opération (ROTATE ou EXCHANGE), puis montre le sens de la rotation en

tirant l'objet par une de ses extrémités (différente de l'origine) vers le côté de la rotation. À l'inverse de l'opération de déplacement précédente, pour la rotation nous trouvons plus commode d'utiliser les menus ; il est plus facile de sélectionner directement le sens de la rotation que de sélectionner une extrémité puis tirer vers l'axe correspondant. Une autre solution plus conviviale serait de représenter les différentes formes de rotation sur des icônes, l'utilisateur n'aura qu'à cliquer sur l'icône correspondante pour l'exécuter sur le DPO sélectionné. La figure III.15 montre les deux possibilités (menu ou icône) pour choisir le sens d'orientation de la rotation.

Décalages Les opérations de décalage circulaire « **`circular_shift`** » et de miroir « **`mirror`** » sont réalisées à travers des boîtes de dialogues, où le programmeur spécifie les paramètres de l'opération (dimension, ainsi que la longueur du décalage pour la première). Comme cette opération ne manipule pas l'objet de manière globale, il est difficile de considérer une manipulation directe.

Réplication Le programmeur exécute l'opération « EXPAND » soit à travers des menus et des boîtes de dialogue où il précisera les paramètres de l'opération (*cf.* figure III.16), soit par manipulation directe. Dans ce dernier cas, il pointe sur une extrémité de l'objet puis tire selon une dimension donnée jusqu'à ce qu'il arrive au nombre de copies souhaité.

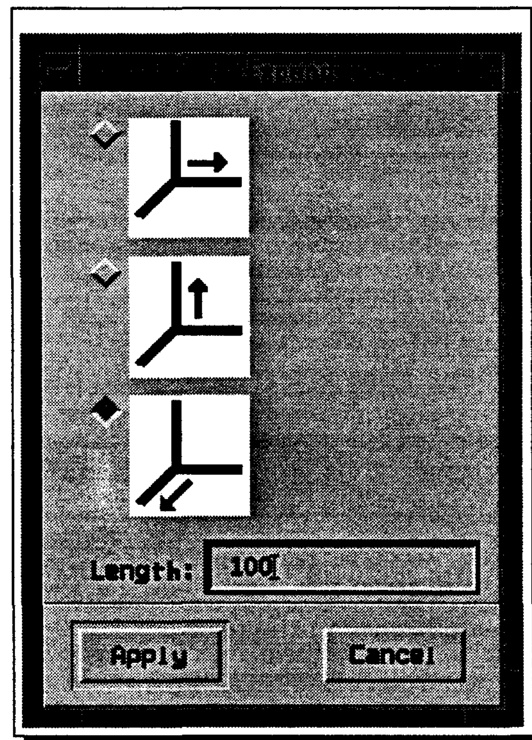


FIG. III.16 - *Application d'une opération d'expansion*

Extractions Les opérations d'extractions sont elles aussi un bon exemple de manipulation directe. Au lieu d'introduire manuellement à travers une boîte de dialogue les caracté-

ristiques du sous-objet à extraire (*cf.* 2.3), HELPDraw offre la possibilité de le découper avec la souris. L'utilisateur se place à l'origine de la partie à extraire puis en déplaçant la souris dessine cette partie (même en 3-D).

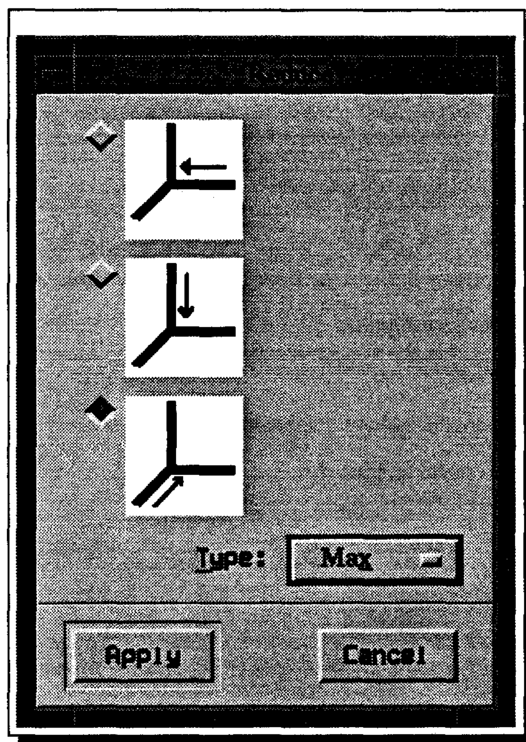


FIG. III.17 - *Application d'une opération de réduction*

Réductions Les opérations de réductions s'effectuent via une boîte de dialogue (*cf.* figure III.17). Le programmeur sélectionne d'une part le sens ou la dimension de la réduction en cliquant sur l'icône correspondante, et choisit d'autre part le type de réduction (addition, multiplication, etc.) dans un menu à options.

Lorsque le programmeur applique une opération à un DPO, un temporaire (TMP) représentant le résultat intermédiaire est automatiquement créé (le DPO source est inchangé). Ce temporaire est un objet data-parallèle représenté avec sa propre couleur. Mais comme il n'a pas de nom (à l'inverse des DPO-utilisateur) et pour qu'il soit mieux distingué, HELPDraw le mettra dans une liste (TMP_List) différente de celle des DPO (DPO_List), *cf.* figure III.18. De plus ce TMP a la particularité d'avoir un historique indiquant la suite d'actions qui a mené à sa création. Par exemple si l'utilisateur applique l'opération géométrique « geomOp1 » sur un DPO « Vect1 », le temporaire généré aura l'historique : « *Vect1.geomOp1* ». Maintenant, si à ce même temporaire lui est appliquée une autre opération « geomOp2 », il sera consommé et HELPDraw génère un nouveau TMP qui aura, cette fois-ci, l'historique :

« *Vect1.geomOp1.geomOp2* »[1]. Cette notation « dpo.geomOp1.geomOp2... » est une expression textuelle directe des manipulations géométriques faites par l'utilisateur. Elle aurait pu être une notation visuelle : une suite de symboles visuels (icônes) où chacun représenterait un type d'opération. Le but est simplement d'avoir sous les yeux la trace de création d'un opérande.

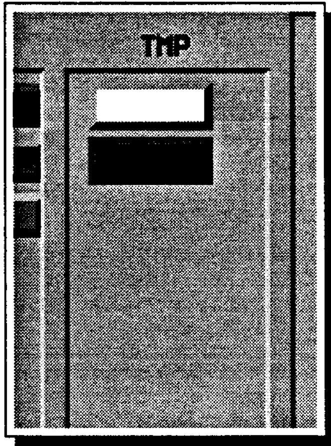


FIG. III.18 - *La liste des* DPO *temporaires*

2.5 Application des opérations microscopiques

Nous étudierons dans cette section l'application des opérations microscopiques et en particulier la spécification des domaines contraints et masqués.

2.5.1 Opérations arithmétiques et logiques

HELPDraw offre toutes les opérations arithmétiques et logiques. Avant d'appliquer une opération microscopique, le programmeur doit assurer la règle de conformité que nous avons décrite auparavant. Pour appliquer une opération le programmeur sélectionne d'abord le ou les DPO concernés (temporaires ou non), puis à travers un menu exécute l'opération, *cf.* figure III.19. La spécification du domaine contraint ou masqué (constructeurs `on` et `where` se fait d'une autre manière (*cf.* 2.5.2)).

À part les constructeurs et les opérations d'affectation et d'association, les autres opérations microscopiques créent un temporaire pour représenter le résultat intermédiaire. Comme dans les opérations géométriques, ce temporaire aura un historique décrivant la trace de sa création. Par exemple l'addition des deux TMP « *dpo1.exchange(x, y)* » et

1. Pour une raison de cohérence, nous considérons qu'un DPO non temporaire a aussi un historique qui n'est rien d'autre que son nom.

« $dpo2.expand(y, 100)$ » crée un autre TMP dont l'historique est : « $dpo1.exchange(x, y)$ + $dpo2.expand(y, 100)$ ». Dès qu'une opération d'affectation ou d'association est effectuée, tous les temporaires sont consommés et le résultat est affecté au DPO membre gauche d'une expression.

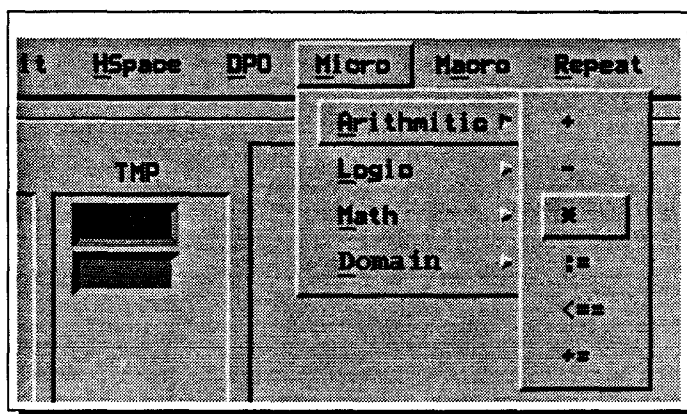


FIG. III.19 - *Un exemple d'application d'une opération microscopique*

Ce principe de construction d'historique permet à HELPDraw de mémoriser la suite des opérations appliquées par le programmeur. C'est de cette même façon que HELPDraw construit le code data-parallèle correspondant à l'instruction développée par le programmeur. Ce code est généré à la suite d'une affectation ou d'une association.

La figure III.20 montre brièvement les différentes étapes que pourrait suivre le programmeur pour le développement d'une expression. L'exemple d'expression choisi est :

$$Mat = Vect1.geomOp1.geomOp2 \ \mu op \ Vect2.geomOp3.geomOp4$$

(Nous supposons que l'hyper-espace et les DPO sont déjà définis).

2.5.2 Domaines contraints et masqués

En HELPDraw, nous retrouvons le principe de la hiérarchie des segments conformes du modèle HELP. Le programmeur peut appliquer un domaine contraint ou masqué sur toutes les opérations arithmétiques ou logiques, par exemple[2] (*cf.* figure III.21) :

```
on(A) Res = (on(A1) where(E>F) E - (on(E.macro) F+C.macro)) +
    A1 - (on(A2) A+B - ((C+D).macro)) + (on(D.macro) C+D)
```

2. L'exemple n'a aucun sens particulier, nous l'avons choisi surtout pour montrer une expression avec plusieurs niveaux de segments conformes (il est certes peu probable qu'un algorithme utilise autant de niveaux dans une expression !!).

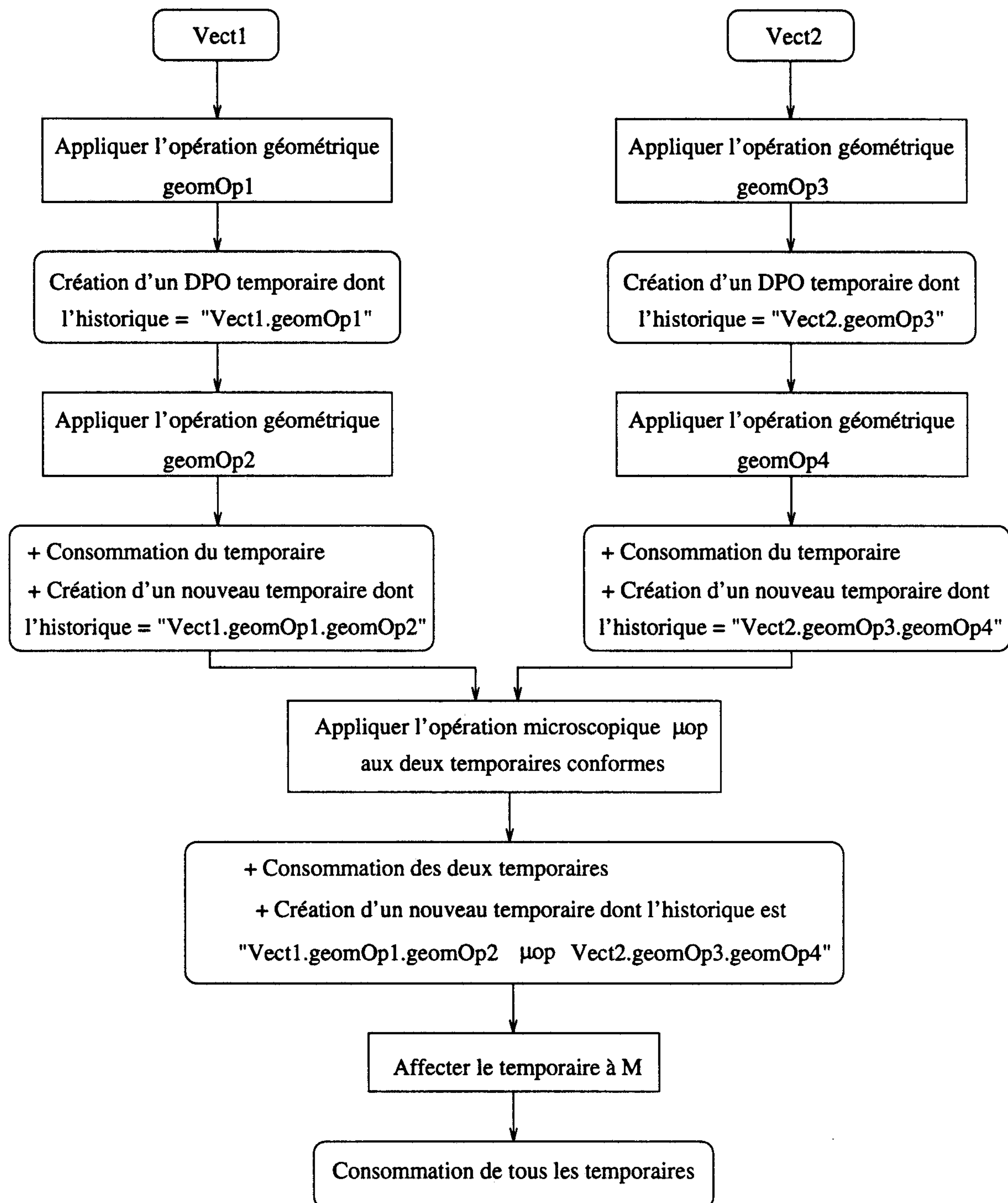


FIG. III.20 - *Un exemple de développement d'une expression sous HELPDraw*

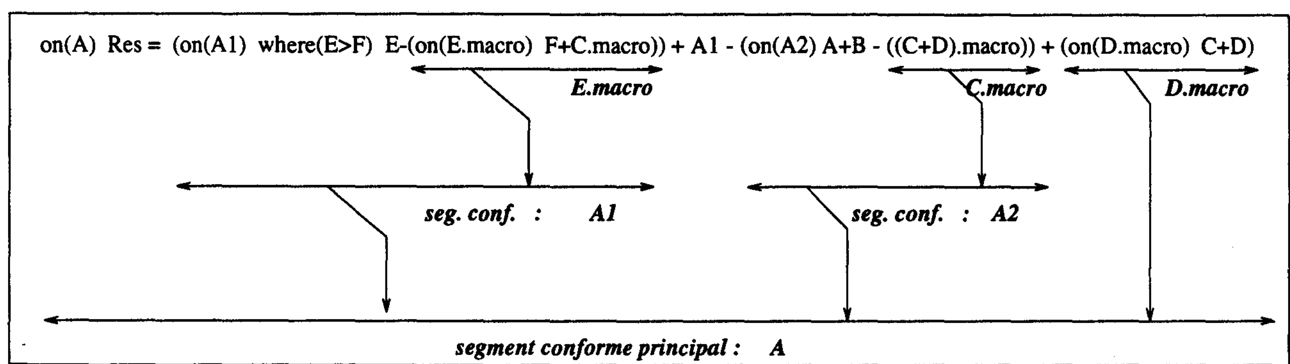


FIG. III.21 - *Représentation des segments d'une expression*

Pour ce faire, HELPDraw offre un gestionnaire de domaines permettant de définir pour chaque instruction une hiérarchie de domaines. La figure III.22 en montre un exemple. L'utilisation du gestionnaire est simple ; elle se fait en trois étapes. Le programmeur crée un domaine contraint, construit la (sous-)expression, puis désactive le domaine.

À chaque fois que le programmeur veut préciser un domaine, il crée un nœud dans la hiérarchie. Ce nœud prend la couleur et le nom de l'objet définissant le domaine (il n'y a pas de nom s'il s'agit d'un temporaire). Par exemple le premier nœud de la figure III.22 représente: `on(A)`. Lorsqu'il veut préciser un domaine masqué, il crée un nœud frère à celui du domaine contraint définissant le segment. Par exemple le nœud blanc de la figure III.22 représente `where(tmp)`, et le segment conforme est de la forme: `on(A1) where(tmp)...`

Le programmeur peut définir également un nouveau segment sans préciser un domaine contraint. Par exemple :

```
on(A2) A + B - ((C+D).macro)
```

l'opération `C+D` n'est pas contrainte par le domaine `A2`, mais le temporaire résultant de `(C+D).macro` l'est. Pour pouvoir réaliser ce type de segment, il crée un nœud vide comme le montre la figure III.22 (*cf.* un nouveau segment sans précision d'un « on »).

La construction des différentes sous-expressions peut se faire indépendamment. Le programmeur sélectionne un nœud donné, puis commence à développer la sous-expression. Celle-ci devient liée au domaine. Le programmeur peut reprendre le développement d'une sous-expression à n'importe quel moment. Lorsque le temporaire correspondant à la sous-expression est sélectionné dans l'éditeur géométrique, le nœud représentant le domaine le plus proche le devient également. Si le programmeur sélectionne par exemple le temporaire « $A + B$ », le nœud $A2$ est automatiquement sélectionné pour montrer le segment auquel appartient le temporaire et donc le contexte de développement.

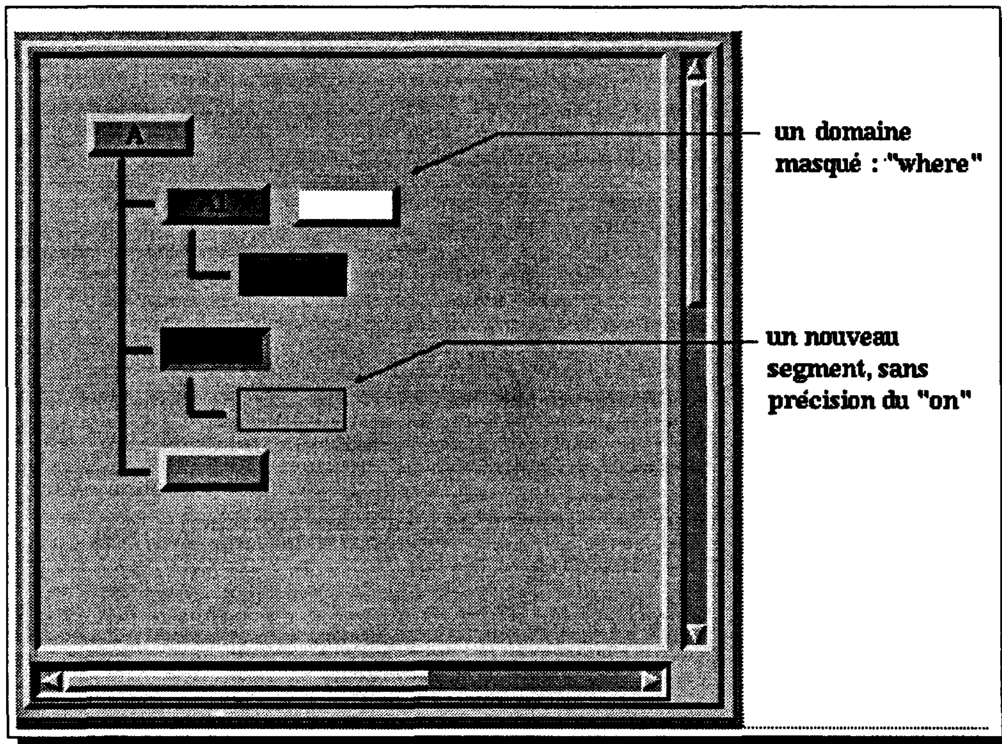


FIG. III.22 - *Définition et sélection de domaine*

En ce qui concerne la construction des historiques, lorsque le programmeur termine le développement d'une sous-expression, il désactive le domaine. Le nœud correspondant est donc supprimé de la hiérarchie (le nœud supprimé ne doit pas posséder de fils). En même temps l'historique de la sous-expression devient : `on(domain) sous-expr` ou `where(domain) sous-expr`.

Exemple

Voici un exemple qui montre les trois étapes principales de développement d'un segment conforme :

1. préciser un domaine contraint, exemple : `dpo2`
2. construire la sous-expression, exemple : `A+B`
3. désactiver le domaine, l'historique devient : `on(dpo2) A+B`.

Naturellement, lors de la construction d'une sous-expression, le programmeur peut créer d'autres domaines fils (imbriqués), d'où la hiérarchie de domaines. Il ne peut cependant définir qu'une seule hiérarchie à la fois.

2.6 Les règles de construction d'historiques

L'application de ces règles permet de retrouver, après chaque opération micro ou macroscopique, l'historique du nouvel objet résultant de cette opération. L'historique s'obtient après la dérivation complète du membre gauche de la règle correspondant à l'opération effectuée. (Nous précisons par ailleurs que ces règles sont gérées de manière interne par HELPDraw ; c'est transparent à l'utilisateur.)

Après l'étude des règles de construction d'historiques concernant les opérations microscopiques (*cf.* 2.6.1), nous verrons celles correspondant aux opérations macroscopiques (*cf.* 2.6.2).

2.6.1 Les règles concernant les opérations microscopiques

Les règles suivantes (*cf.* table III.1) montrent exactement comment se fait la construction des historiques. Les règles des opérations microscopiques sont les mêmes que celles utilisées dans n'importe quel langage manipulant des expressions arithmétiques et logiques. La seule différence réside dans l'ajout des constructeurs de domaine contraint et masqué « `on` » et « `where` ». Or, les constructeurs sont pris en compte dans la grammaire définissant les expressions de la même façon qu'un simple opérateur unaire, mais de priorité inférieure aux opérateurs classiques. La figure III.23 montre à titre indicatif l'ordre de priorité des opérateurs. Dès lors que le programmeur réalise ses opérations une par une, c'est à sa charge de respecter la priorité des opérateurs. S'il veut réaliser par exemple : `A+B*C`, il doit d'abord effectuer la multiplication pour faire ensuite l'addition. S'il fait l'inverse, HELPDraw considère qu'il s'agit de : `(A+B)*C`.

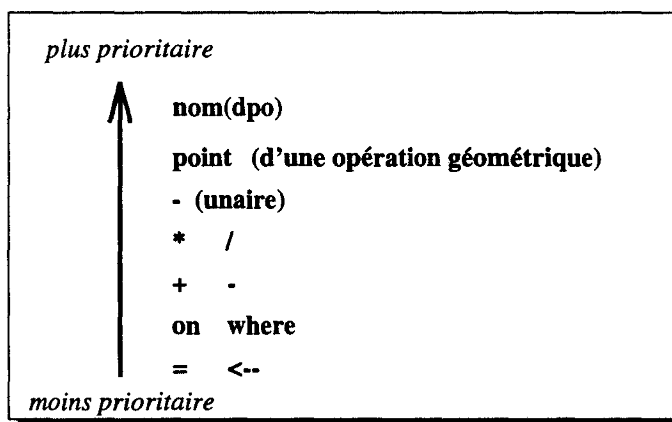


FIG. III.23 - *Gestion de priorités des opérateurs*

TAB. III.1 - *Les règles utilisées pour les opérations microscopiques*

- Tout DPO a deux propriétés : h et p
  - h : son historique — $h(\text{arg}) \rightarrow$ l'historique associé à *arg*
  - p : sa priorité (*cf.* III.23) — $p(\text{arg}) \rightarrow$ la priorité associée à *arg*
- Tout DPO variable a ses propriétés initialisées à :
  $h = nom(\text{DPO})$
  $p = p(\ nom(\text{DPO})\ )$
- Pour toute opération microscopique, calculer les propriétés du résultat en appliquant l'une des règles suivantes :

Soit la fonction f qui renvoie l'une des deux valeurs selon la priorité de **op** :

$$f(\text{arg, op}) \longrightarrow (\ h(\text{arg})\ ) \quad \underline{\textbf{si}}\ p(\text{arg}) < p(\text{op})$$
$$\longrightarrow h(\text{arg}) \quad \underline{\textbf{sinon}}$$

R_1	[*opBinaire, arg1, arg2*]	:	$h =$	f(arg1,opBinaire) **opBinaire** f(arg2,opBinaire)
			$p =$	p(opBinaire)
R_2	[*opUnaire, arg*]	:	$h =$	***opUnaire*** f(arg, opUnaire)
			$p =$	p(opUnaire)
R_3	[*désactiver-on, domaine, arg*]	:	$h =$	**on**(h(domaine)) h(arg)
			$p =$	p(on)
R_4	[*désactiver-where, domaine, arg*]	:	$h =$	**where**(h(domaine)) h(arg)
			$p =$	p(where)
R_5	[*désactiver-segt-non-contraint,arg*]	:	$h =$	h(arg)
			$p =$	p(on)

2.6.2.1 Les règles

Comme nous l'avons déjà décrit, l'historique de tout DPO variable est le nom de ce DPO. Ainsi, l'historique d'une sous-expression peut être un nom de DPO (lorsque la sous-expression n'est rien d'autre que la référence à un DPO variable) ou l'historique du temporaire représentant les opérations effectuées. Lorsque le programmeur applique un opérateur binaire, HELPDraw applique la règle R_1. L'historique est construit par la composition de l'historique des deux arguments de l'opération (*arg*1 et *arg*2) ainsi que l'opérateur. Lorsqu'un argument est un temporaire résultant de l'application d'une opération de priorité inférieure à celle de l'opérateur courant, HELPDraw met l'historique de cet argument entre parenthèses (appel de la fonction f), sinon il reste tel qu'il est. Si les deux arguments de l'opération binaire « $*$ » sont respectivement `A+B` et `C.macro` (le point est plus prioritaire que le « $*$ » qui est plus prioritaire que le « $+$ »), l'historique correspondant au produit devient :

```
(A+B) * C.macro
```

Après chaque règle, on recalcule le nouveau niveau de priorité p correspondant au résultat de l'opération. Pour une opération binaire, le résultat aura la priorité de l'opérateur appliqué.

La prise en compte de l'application d'un opérateur unaire se fait de la même manière. L'historique est construit par la composition de l'opérateur et l'historique de l'argument (règle R_2). Si l'opérateur unaire est par exemple « - », en fonction de la priorité, l'historique peut être : `-(A+B)` ou sans parenthèses : `- C.macro`.

Si la sous-expression développée est liée à un domaine contraint ou masqué, dès que le programmeur désactive ce domaine, HELPDraw applique la règle R_3 ou R_4 pour compléter l'historique de la sous-expression. Il compose le `on`, en précisant l'historique du domaine, avec l'historique de la sous-expression (noté $h(arg)$ dans la règle). La dernière règle (R_5) correspond à un segment sans précision de domaine (un nœud vide dans la hiérarchie). La priorité ici est la même que celle d'un « `on` » ou d'un « `where` ».

2.6.2.2 Exemple

Nous allons suivre pas à pas la construction de l'historique correspondant à l'expression suivante :

```
on(dpo) Res = (on(dpo2) A+B) - C.macro + dpo3 -
        (on(dpo4) where(E>F) (E-F).macro)
```

Bien que le développement de plusieurs parties de cette expression puisse se faire en parallèle, pour faciliter les explications, nous le ferons séquentiellement.

1. On précise d'abord le domaine (ou segment) principal explicité par l'objet dpo. Ce domaine étant sélectionné, on crée un nouveau nœud correspondant au domaine dpo2 afin de réaliser la sous-expression `A+B`. En appliquant l'opérateur binaire « + » sur les arguments `A` et `B`, un temporaire (soit $Tmp1$) est créé. HELPDraw déclenche alors la règle R_1 et appel deux fois la fonction f pour construire l'historique correspondant au temporaire. L'historique devient « `A+B` ». On désactive ensuite le domaine contraint dpo2, HELPDraw applique alors la règle R_3 ; l'historique du nouveau temporaire $Tmp2$ devient « `on(dpo2) A+B` ». Notons que la dernière opération réalisée est un « `on` », d'où la priorité associée à $Tmp2$ est celle du « `on` ».

$$Tmp2 \left\{ \begin{array}{rcl} h & = & on(dpo2)\ A+B \\ p & = & p(on) \end{array} \right.$$

2. Pour réaliser la deuxième sous-expression, on sélectionne d'abord le domaine principal, on sélectionne ensuite le DPO C pour lui appliquer alors l'opération géométrique `macro`. Comme nous le verrons plus loin (*cf.* 2.6.2), HELPDraw applique une autre règle correspondant aux opérations macroscopiques afin de construire l'historique « `C.macro` » pour le temporaire $Tmp3$. Notons que le dernier opérateur correspondant à $Tmp3$ est le point.

$$Tmp3 \begin{cases} h & = & C.macro \\ p & = & p(.) \end{cases}$$

3. On peut déjà réaliser la soustraction. On sélectionne les deux temporaires $Tmp2$ et $Tmp3$, puis on applique l'opérateur binaire « `-` ». On obtient un temporaire $Tmp4$. HELPDraw applique alors la règle R_1, l'historique dépend de la fonction f pour les deux arguments ($Tmp2$ et $Tmp3$). Or puisque l'opérateur « `-` » est de priorité supérieure à celle du « `on` » (priorité de $tmp2$), l'historique de $Tmp2$ est mis alors entre parenthèses. Ce n'est par contre pas le cas en ce qui concerne $Tmp3$ étant donné que le « `-` » est de priorité inférieure à la sienne (priorité du point). L'historique de $Tmp4$ devient par conséquent « `(on(dpo2) A+B) - C.macro` ». Notons que la dernière opération correspondant à $Tmp4$ est le moins binaire.

$$Tmp4 \begin{cases} h & = & (on(dpo2)\ A+B) & - & C.macro \\ p & = & p(-) \end{cases}$$

4. On effectue de la même manière : $Tmp4$ `+ dpo3`. HELPDraw applique pour la construction de l'historique, la règle R_1. Appelons le temporaire créé $Tmp5$, son historique est alors :

« `(on(dpo2) A+B) - C.macro + dpo3` »

Notons que la dernière opération est réalisée par l'opérateur binaire « `+` ».

$$Tmp5 \begin{cases} h & = & (on(dpo2)\ A+B) & -C.macro & + & dpo3 \\ p & = & p(+) \end{cases}$$

5. Pour réaliser la dernière sous-expression, on crée un nœud correspondant au domaine contraint `dpo4` (un nœud fils du domaine principal). On effectue ensuite une opération logique « > » avec les arguments `E` et `F`. Le temporaire créé a un historique « `E>F` » (règle R_1). On crée après un second nœud, cette fois correspondant à un domaine masqué « `where` ». Ce nœud frère du précédent est explicité par le temporaire `E>F`. L'opérateur binaire « `-` » est appliqué ensuite aux DPO `E` et `F`. HELPDraw applique de nouveau la règle R_1 pour construire l'historique « `E-F` ». Le temporaire créé lui est appliquée une opération macroscopique `macro`,

l'historique devient « `(E-F).macro` ». HELPDraw a appliqué des règles correspondant aux opérations géométriques (*cf.* 2.6.2). Il suffit maintenant de désactiver successivement les domaines masqué et contraint. En désactivant le « **`where`** », HELPDraw applique la règle R_4. L'historique de $Tmp6$ devient alors :

« `where(E>F) (E-F).macro` »

De même la désactivation du domaine contraint mène à l'application de la règle R_3, l'historique évolue vers :

« `on(dpo4) where(E>F) (E-F).macro` »

Notons que la dernière opération est un « `on` ».

$$Tmp6 \begin{cases} h &= on(dpo4)\ \ where(E > F)\ \ (E - F).macro \\ p &= p(on) \end{cases}$$

6. La soustraction $Tmp5 - Tmp6$, mène HELPDraw à appliquer la règle R_1. Notons ici que $Tmp5$ n'est pas mis entre parenthèses puisque sa priorité ($p(Tmp5) = p(+)$) est la même que celle de l'opérateur courant. L'historique devient alors pour tout le membre droit de l'expression :

« `(on(dpo2) A+B) - C.macro + dpo3 - (on(dpo4) where(E>F) (E-F).macro)` »

7. Pour l'opération d'affectation, une fois effectuée, HELPDraw applique la règle R_1. L'historique construit ne correspond plus à aucun temporaire puisque les opérations d'affectations consomment tous les temporaires. Mais comme ces opérations désactivent en plus les domaines restants, HELPDraw applique alors la règle R_3. L'historique avant la règle R_3, est :

« `Res = (on(dpo2) A+B) - C.macro + dpo3 - (on(dpo4) where(E>F) (E-F).macro)` »

En désactivant le domaine principal, l'historique devient enfin :

```
on(dpo) Res = (on(dpo2) A+B) - C.macro + dpo3 -
        (on(dpo4) where(E>F) (E-F).macro)
```

2.6.2.3 Discussion

Nous avons développé une instruction où il y a plusieurs niveaux de segments conformes. À noter que l'utilisateur de HELPDraw n'est pas obligé de concevoir ces instructions de cette manière. Il peut arriver à la même sémantique, au même résultat en utilisant des extractions de sous-objets au lieu des spécifications de domaines contraints. Nous reviendrons sur cette possibilité plus loin dans le chapitre V, où on montre que le « **on** » peut effectivement être substitué par des extractions (*cf.* 2.5.2).

2.6.2 Les règles concernant les opérations géométriques

Une opération géométrique peut être appliquée à une sous-expression. La table III.2 montre les règles permettant la construction de l'historique résultant de l'application d'une opération géométrique.

TAB. III.2 - *Les règles utilisées pour les opérations macroscopiques*

- Rappelons la fonction f :

$$f(\text{arg}, \text{op}) \longrightarrow (\ h(\text{arg})\ ) \quad \underline{\text{si}}\ p(\text{arg}) < p(\text{op})$$
$$\longrightarrow h(\text{arg}) \quad \underline{\text{sinon}}$$

- Pour toute opération macroscopique *macro*, calculer les propriétés du résultat en appliquant la règle suivante :

R_6 [*macro*] : $h = f(\text{arg}, .)\ .\ g(\textbf{macro})$
$p = p(.)$

r_1	g(move)	$\longrightarrow$	**move(x$_1$,y$_1$,z$_1$)**
r_2	g(rotate)	$\longrightarrow$	**rotate(dim1,dim2)**
r_3	g(exchange)	$\longrightarrow$	**exchange(dim1,dim2)**
r_4	g(rotate_todiag)	$\longrightarrow$	**rotate_todiag([dim1 dim2])**
r_5	g(rotate_toline)	$\longrightarrow$	**rotate_toline(dim)**
r_6	g(rotate_triangle)	$\longrightarrow$	**rotate_triangle(dim1,dim2)**
r_7	g(circular_shift)	$\longrightarrow$	**circular_shift(dim,off)**
r_8	g(mirror)	$\longrightarrow$	**mirror(dim)**
r_9	g(expand(dim,d))	$\longrightarrow$	**expand(dim,d)**
r_{10}	g(expand(dim))	$\longrightarrow$	**expand(dim)**
r_{11}	g(extract_subdpo)	$\longrightarrow$	**extract_subdpo(Norig,Nlen)**
r_{12}	g(extract_diag)	$\longrightarrow$	**extract_diag([dim1,dim2],Norig,Nlen)**
r_{13}	g(extract_triangle)	$\longrightarrow$	**extract_triangle([dim1,dim2],Norig,Nlen)**
r_{14}	g(reduceOP[a])	$\longrightarrow$	**reduceOP(dim)**

[a] reduceOP : reduceadd, reducemul, reduceor, reduceand, reducemax, reducemin

L'objet source des opérations géométriques peut être un DPO variable ou un DPO-expression (exemple : « `on(D) A + B` »). C'est pourquoi, nous avons utilisé l'appel de la fonction f dans la règle R_6. Ensuite l'appel de la fonction g permet de retrouver l'historique du résultat de l'opération macroscopique. Selon l'opération effectuée, g est dérivée par l'une des sous-règles r_i ($i = 1$ à 14). Si le programmeur applique « `move(x1,y1,z1)` », l'historique devient par exemple : « `(on(D) A + B).move(x1,y1,z1)` ».

2.7 Développement directe d'expressions data-parallèles

L'application des opérations géométriques une par une est importante puisque l'utilisateur voit concrètement l'interprétation visuelle de chaque opération. Ce n'est par contre pas le cas pour les opérations microscopiques qui n'apportent pas au programmeur un résultat visuel significatif. La forme visuelle du résultat d'une opération microscopique est souvent la même que celle des DPO sources ou du domaine contraint (elle peut être différente dans le cas d'une opération d'association). De même la gestion des priorités et des segments conformes est parfois fastidieuse. Ainsi, dans les cas où le programmeur peut séparer la partie microscopique de la partie macroscopique, il peut vouloir réaliser d'un seul coup une expression ou des parties de l'expression, par exemple :

$$Res \;=\; dpo1 \;+\; dpo2 \;*\; dpo3 \;-\; dpo4$$

ou encore avec des domaines contraints ou masqués :

$$on(dpo)\; Res \;=\; dpo1 \;+\; (on(dpo2)\; dpo3)\; \ldots$$

Pour le faire HELPDraw offre une autre alternative aux menus : un éditeur graphique pour le développement des expressions.

L'éditeur d'expressions (*cf.* figure III.24) permet de développer des expressions data-parallèles où les noms de variables sont formels. Lors de l'utilisation d'une expression, ces noms doivent être remplacés par les vrais noms de DPO définis dans un hyper-espace. Par exemple dans l'expression : « $Res \;=\; Arg1 \;+\; Arg2 \;-\; Arg3$ », les noms (Res et $Argi$) ne sont qu'à titre indicatif. Dans cette partie nous verrons comment le programmeur construit une expression, de quels composants disposent-il, et enfin comment il utilise cette expression.

2.7.1 Construction d'une expression

La construction d'une expression consiste à assembler des composants (des icônes) en un arbre, *cf.* figure III.25. Cet assemblage se fait par manipulation directe ; le programmeur glisse avec la souris des composants vers l'éditeur d'arbres (une partie de l'éditeur d'expressions),

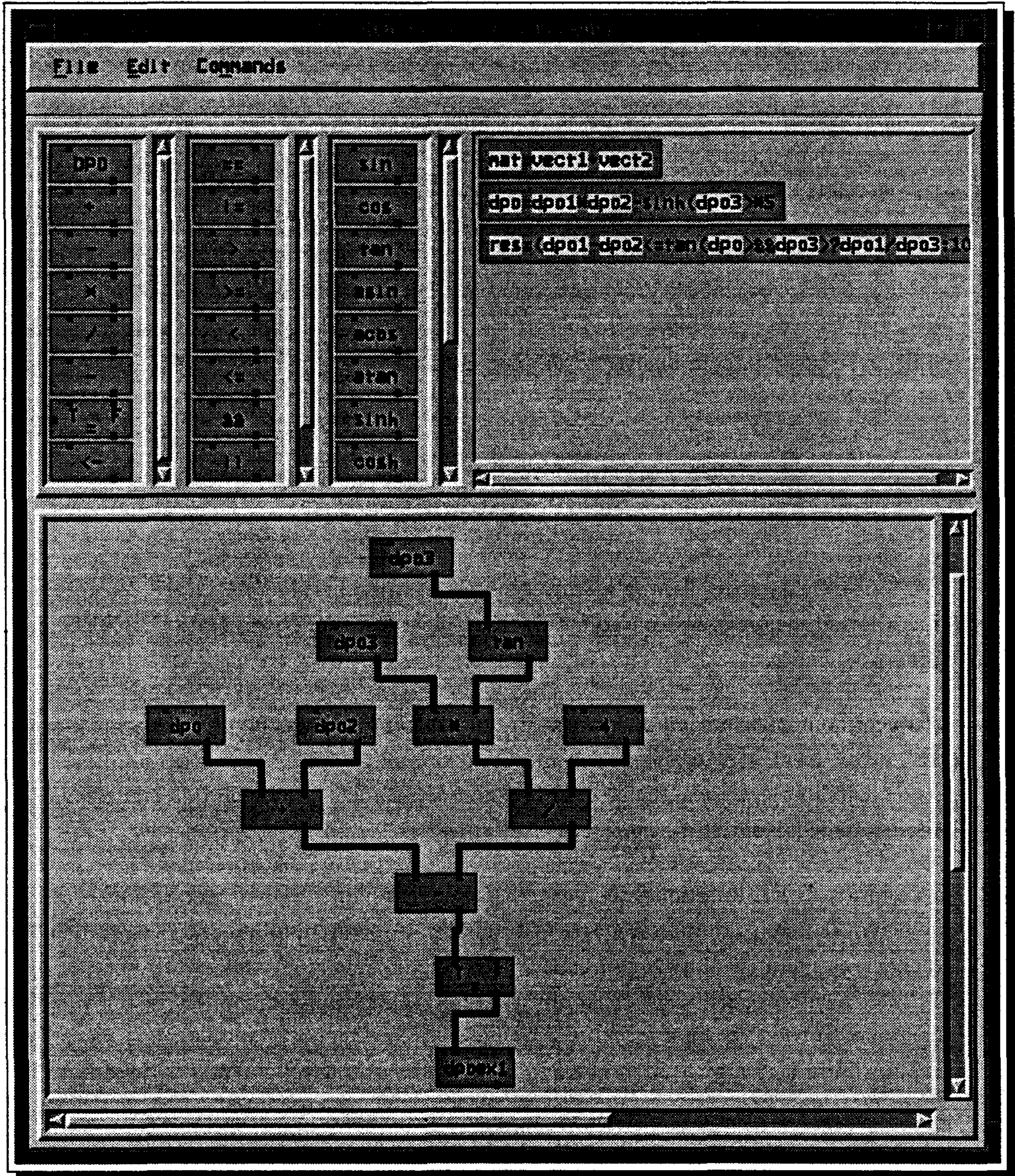


FIG. III.24 - *L'éditeur de construction d'expressions*

puis les lie en tirant des arcs d'un composant à l'autre. Pour une raison de lisibilité, l'éditeur d'expressions offre en particulier une fonction permettant de redessiner l'arbre avec la meilleure disposition des composants et en évitant tout croisement d'arcs.

Comme il est parfois lent de construire un arbre, une autre alternative complémentaire est offerte par l'éditeur. Elle permet d'introduire textuellement l'expression ou une partie de celle-ci. Dès qu'elle est introduite, l'expression textuelle est interprétée en un arbre. Le programmeur peut l'utiliser telle qu'elle est ou l'inclure dans un autre arbre. L'expression textuelle est actuellement écrite dans une syntaxe C.

Au fur et à mesure de l'assemblage interactive des composants, HELPDraw vérifie la syntaxe de ce qui est construit ; le programmeur ne peut lier un composant à un autre que si c'est cohérent avec la syntaxe d'une expression HELPDraw. Quant à l'expression textuelle, une fois introduite, elle passe par un analyseur lexico-syntaxique qui après s'être assuré de sa justesse la traduira en un arbre.

2.7.2 Les différents composants d'une expression

L'éditeur offre dans des listes plusieurs types de composants : l'icône « **nom** », les opérateurs arithmétiques et logiques, les constructeurs **on** et **where** des fonctions mathématiques, et des fonctions prédéfinies.

L'icône « nom » permet au programmeur d'introduire les noms de DPO. Ce sont des noms virtuels qui peuvent être d'ailleurs les mêmes (exemple : $dpo = dpo + dpo$). Pour des raisons de lisibilité, le programmeur peut introduire des noms permettant de rendre la lecture de l'expression significative (exemple : $res = dpo1 + dpo2$).

Les opérateurs arithmétiques incluent en particulier l'opérateur d'association représenté par la flèche (comme dans le langage C-HELP) et l'opérateur d'affectation qui peut être éventuellement conditionnel. La figure III.25 montre un exemple d'affectation conditionnelle. L'icône a trois entrées : celle de gauche représente le résultat du test booléen ; selon ce résultat une des deux entrées en haut est considérée. Cet opérateur représente en fait l'affectation conditionnelle du langage C (étendue aux expressions data-parallèles) :

$$res = cond \, ? \, expr1 \; : \; expr2$$

Le programmeur peut utiliser aussi des fonctions mathématiques : trigonométriques, logarithmiques, racine carré, etc. Par exemple :

$$res = (dpo1 + dpo2 - SIN(dpo3)) \, / \, SQRT(dpo4)$$

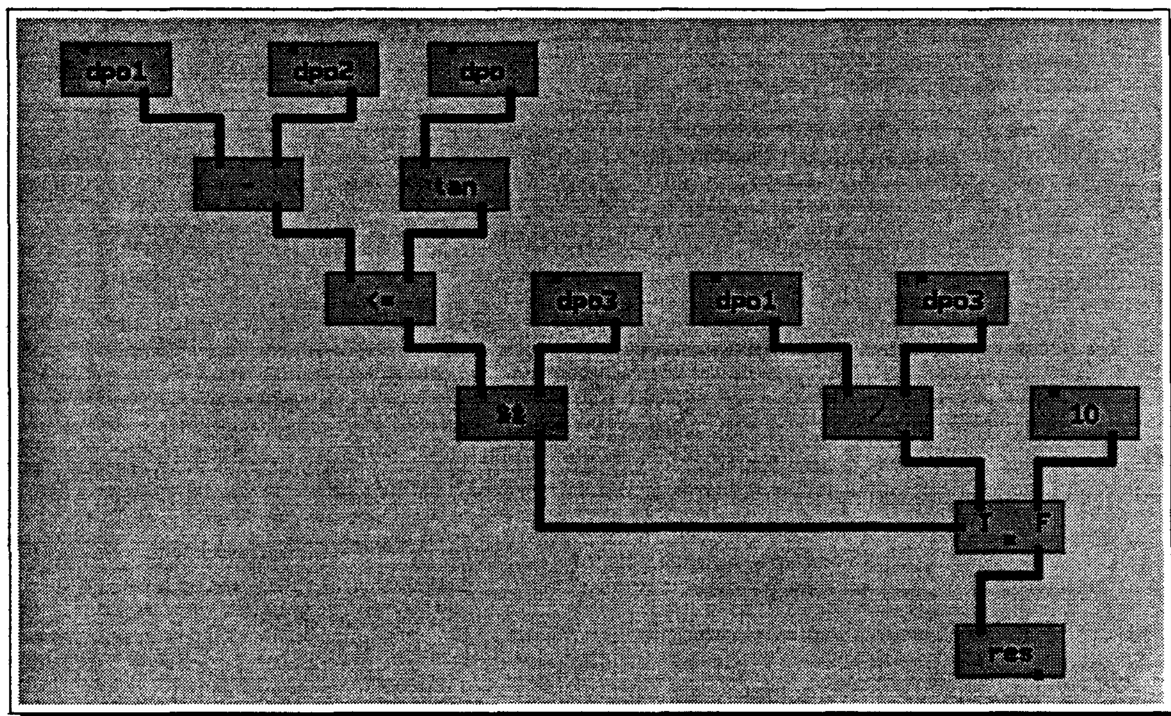


FIG. III.25 - *L'arbre d'un exemple d'affectation conditionnelle:* « $res = (dpo1 - dpo2 <= tan(dpo)\ \&\&\ dpo3)\ ?\ dpo1/dpo3 :\ 10$ »

Il dispose également de quelques fonctions prédéfinies que nous retrouvons dans plusieurs langages data-parallèles, telles que : inversion ou multiplication de matrices.

2.7.3 Utilisation d'une expression

Une fois l'expression construite, l'éditeur l'encapsule dans un seul objet graphique (un icône-expression). Celui-ci représentera la forme textuelle de l'expression, *cf.* figure III.26. Les noms de DPO, que nous avons appelés arguments formels, sont mis en évidence avec une autre couleur.

Lorsque le programmeur veut utiliser un icône-expression, il le glisse d'abord vers l'éditeur d'opérations géométriques où les vrais DPO sont définis (temporaires ou non). Il procède ensuite à la correspondance entre les arguments formels et les DPO. Cette correspondance se fait de manière très simple. Pour chaque argument, le programmeur clique avec la souris sur le DPO puis sur l'argument formel de l'expression. Celui-ci aura tout de suite sa couleur changée vers celle du DPO. La figure III.27 montre un exemple d'icône-expression après correspondance.

La correspondance DPO-arguments n'est rien d'autre que le remplacement des arguments formels de l'expression par l'historique des DPO. Il ne restera enfin au programmeur

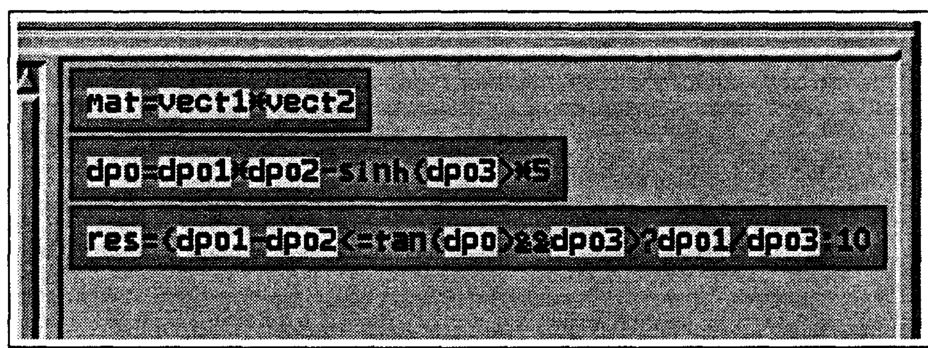


FIG. III.26 - *Exemple d'une liste d'expressions encapsulées*

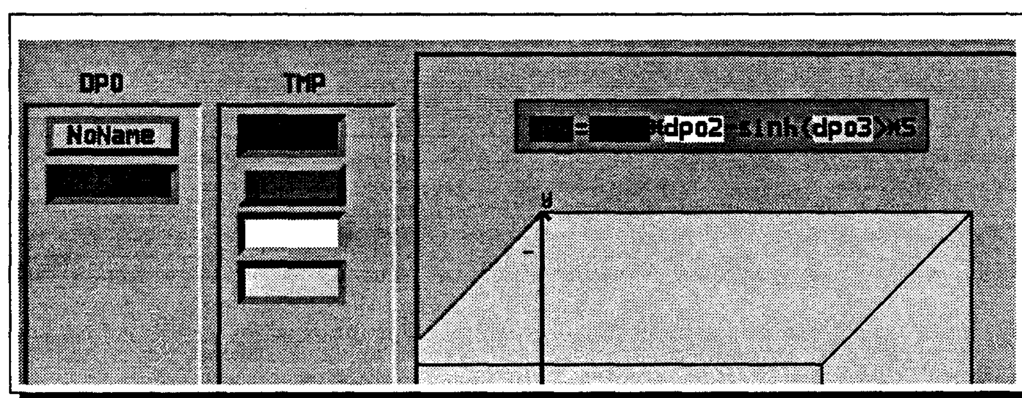


FIG. III.27 - *Un exemple d'icône-expression après correspondance*

qu'à exécuter l'expression. Si le programmeur fait la correspondance par exemple entre les arguments d'un icône-expression « $dpo1 + dpo2$ » et deux DPO dont les historiques sont ($A.exchange(x, y)$ et $B.expand(x, 100)$), l'exécution génèrera un nouveau temporaire dont l'historique est « $A.exchange(x, y) + B.expand(x, 100)$ ».

2.8 Manipulation des scalaires

Pour le développement de ses algorithmes, le programmeur a besoin aussi de manipuler des scalaires. Ceux-ci peuvent être utilisés à plusieurs endroits : une expression data-parallèle qui donne un résultat scalaire, une fonction qui renvoi un scalaire, une expression entièrement scalaire, une variable scalaire utilisée comme paramètre d'une opération macroscopique (exemple : $A.expand(x, var1)$), etc.

HELPDraw, via son éditeur d'expressions, permet de définir des scalaires : variables ou constantes. Le programmeur les définit interactivement à l'aide de boîtes de dialogue. Lorsque la valeur d'une variable est nécessaire pour l'interprétation visuelle, HELPDraw demandera au programmeur d'introduire une valeur-exemple. Pour exécuter par exemple l'opération géo-

métrique $A.expand(x, var1)$, HELPDraw doit connaître la valeur de $var1$. Si le programmeur donne une valeur « 100 » pour $var1$, HELPDraw réalisera une expansion d'une distance de 100 (c'est ce qui sera visualisé), mais le code généré correspondra à une opération paramétrée, il y aura $var1$ plutôt que 100.

Le programmeur peut construire une (sous-)expression scalaire. Il utilise l'éditeur d'expressions comme pour les expressions data-parallèles. Les opérateurs et les fonctions mathématiques, à l'exception de l'opérateur d'association, peuvent être appliqués sur des scalaires. Ils sont alors interprétés comme opérateurs et fonctions scalaires au lieu de data-parallèles. Mais comment HELPDraw peut distinguer une expression scalaire d'une expression data-parallèle?

HELPDraw se base sur le nom des variables utilisées dans l'expression. Si toutes ces variables sont définies comme scalaires, alors l'expression l'est aussi. Il est possible de connaître la classe d'une variable car en ce qui concerne les scalaires, le programmeur ne peut référencer que les variables déjà définies. S'il a défini $var1$ et $var2$ comme variables scalaires, l'expression $var1 + var2$ est donc considérée scalaire.

2.9 Intégration du code dans un programme

Nous avons vu comment HELPDraw réussit à garder la trace de toutes les opérations faites par le programmeur (*cf.* 2.4 et 2.5). Il se base sur les temporaires et leur historique générés après chaque opération. HELPDraw utilise le même principe pour générer le code des instructions dans le langage cible. Au fur et à mesure de l'application des opérations, il construit le code. Dès que le programmeur termine le développement d'une instruction, HELPDraw génère le code qui lui correspond. Nous considérons qu'une expression est terminée, lorsque le programmeur exécute une affectation ou une association.

HELPDraw génère actuellement un code dans un des deux langages data-parallèles : C-HELP ou HPF (nous avons réservé les deux prochains chapitres à la génération de codes dans ces deux langages). Le code (déclarations et instructions) est généré dans une fenêtre que nous appelons « Buffer », *cf.* figure III.28. Si le programmeur accepte ce code, il sera automatiquement copié dans l'éditeur de texte de HELPDraw à l'endroit choisi par le programmeur.

Indépendamment du code généré par HELPDraw, le programmeur écrit ce qu'il veut dans l'éditeur de texte. Il écrit en particulier les entêtes de programme et de fonctions, des commentaires, etc. Lorsqu'il arrive à l'endroit où il veut écrire du code pouvant être généré par HELPDraw, en particulier des instructions data-parallèles, il passe au développement sous les autres éditeurs de HELPDraw.

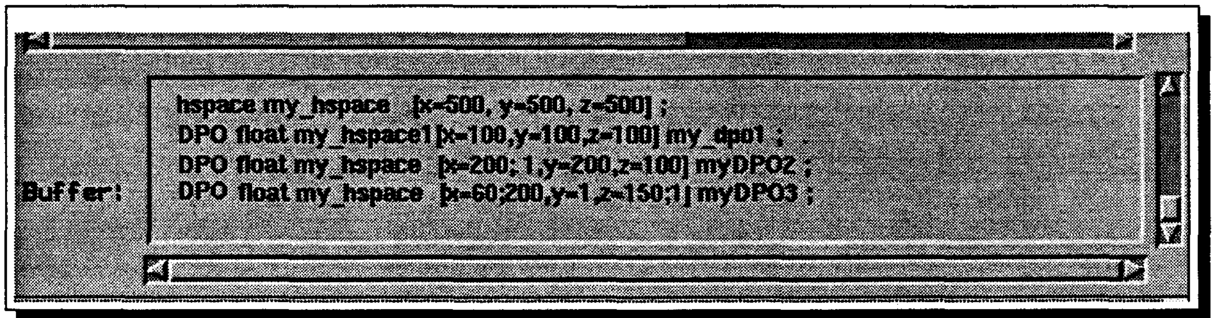


FIG. III.28 - *Génération de code (ici* C-HELP*)*

3 Développement de blocs d'instructions

HELPDraw offre un deuxième niveau de développement de codes data-parallèles : le développement de blocs d'instructions. Le programmeur précise d'abord le type de bloc qu'il souhaite développer : une simple séquence d'instructions, une construction conditionnelle « `If_Then_Else` », une construction de domaine contraint par bloc « `On` », une construction de domaine masqué par bloc « `Where_Elsewhere` », une construction itérative « `While` » ou « `Repeat` ». Il passe ensuite au développement des instructions constituant ce bloc. Ces instructions sont réalisées comme nous l'avons décrit auparavant. Une fois que tout le bloc est constitué, HELPDraw génèrera le code correspondant.

3.1 L'éditeur de blocs d'instructions

HELPDraw offre un autre éditeur graphique pour le développement des blocs d'instructions. La figure III.29 montre une vue globale de cet éditeur. Le programmeur dispose d'une liste d'icônes représentant chacun un type de bloc. Pour en construire un, le programmeur glisse d'abord l'icône correspondant dans l'espace de travail de l'éditeur. L'icône sera visualisé comme une boîte ouverte ; dans l'exemple de la figure III.29, c'est un bloc « `Where_Elsewhere` » qui est sélectionné. Toutes les instructions développées (scalaires ou data-parallèles) sont ensuite insérées dans cette boîte. Elles sont construites dans l'ordre, tout en étant liées au contexte du bloc. Si celui-ci contient par exemple une condition, le développement des instructions se fera en fonction de cette condition. La description ci-dessous des différents blocs illustrera mieux ce principe de développement.

3.1.1 Le constructeur de domaine contraint par bloc « on »

Le « `on` » est un constructeur data-parallèle. Il permet d'expliciter le domaine de conformité pour un bloc d'instructions. La figure III.30 montre une représentation graphique du

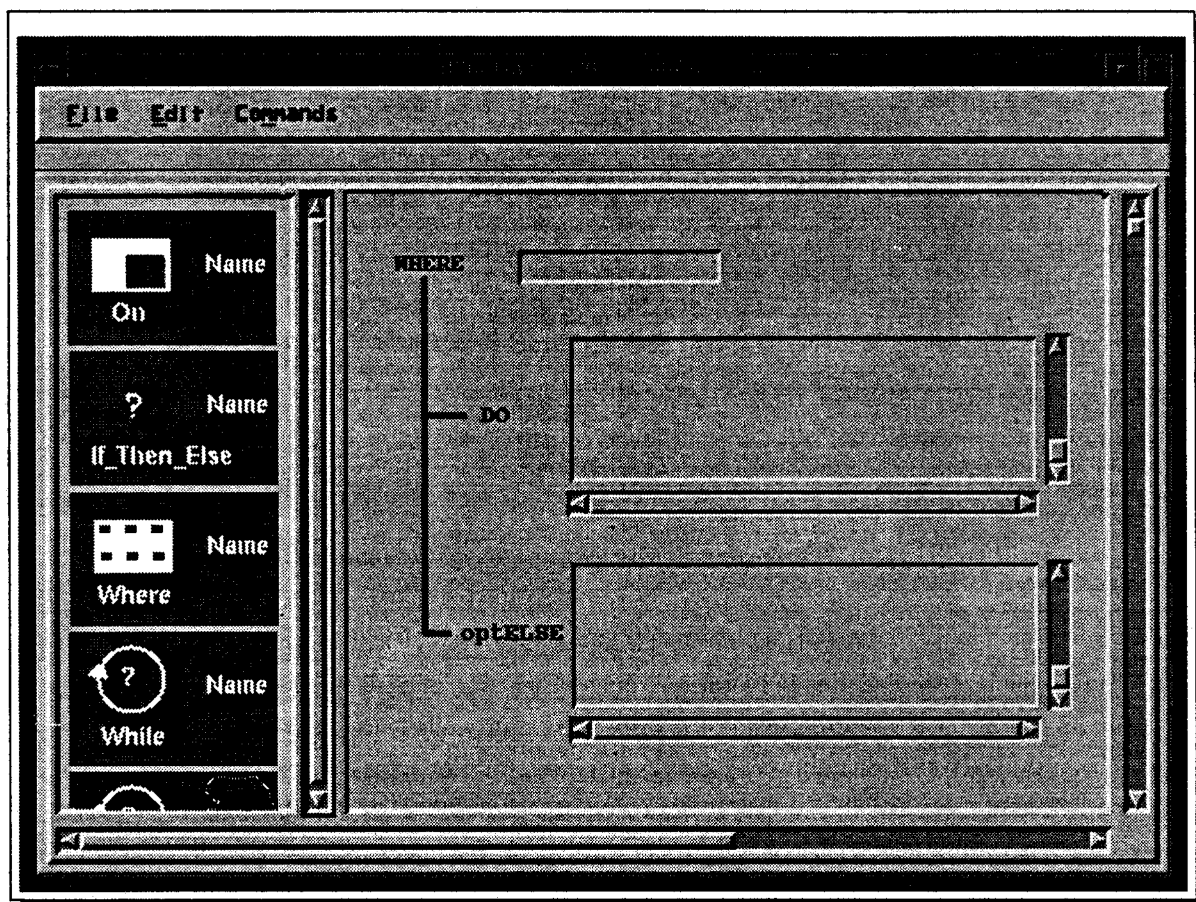


FIG. III.29 - *L'éditeur de blocs d'instructions*

constructeur « **on** ». Le programmeur précise d'abord le domaine de conformité (le DPO du « **on** »), puis passe au développement des instructions. Le domaine de conformité est spécifié en sélectionnant le DPO concerné dans l'éditeur d'opérations géométriques puis en cliquant sur l'emplacement qui lui est réservé dans la boîte du « **on** ». Cet emplacement prendra ensuite la couleur du DPO. Dans le cas où c'est un temporaire, il ne pourra être consommé qu'à la fin du bloc, puisque HELPDraw doit vérifier à chaque interprétation visuelle les règles établies dans le modèle géométrique (ici, l'argument du **on** permet de vérifier la règle de conformité). Le temporaire est donc présent mais inaccessible pendant tout le bloc.

Les instructions développées du bloc sont insérées dans l'ordre les unes après les autres.

3.1.2 La construction conditionnelle « If_Then_Else »

La condition est une expression logique dont le résultat est obligatoirement scalaire. Cette expression est spécifiée avec les autres instructions élémentaires. Un emplacement est réservé

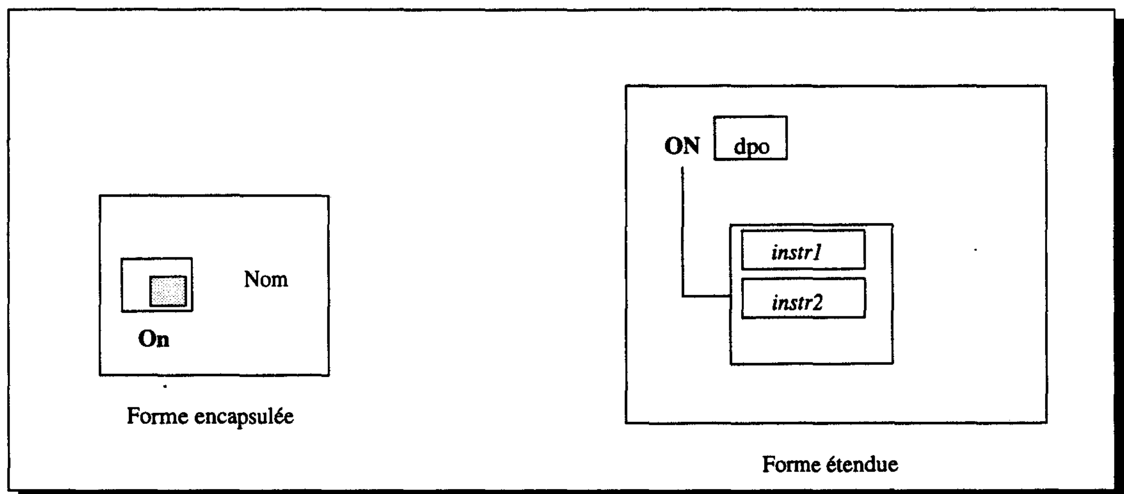


FIG. III.30 - *La représentation graphique de la construction « On »*

à la partie « Then » et éventuellement un autre pour le « Else » qui est optionnel. Le programmeur peut y insérer des instructions élémentaires. Celles-ci ne peuvent être développées que si la condition a été déjà insérée (l'interprétation visuelle relative aux instructions peut dépendre de la condition). Le programmeur construit l'expression logique qui correspond à la condition, puis glisse l'icône-expression dans la boîte. À l'inverse du « on », ici le TMP de la condition (un scalaire) est consommé aussitôt qu'il est inséré. Les instructions sont ensuite développées indépendamment du résultat de la condition.

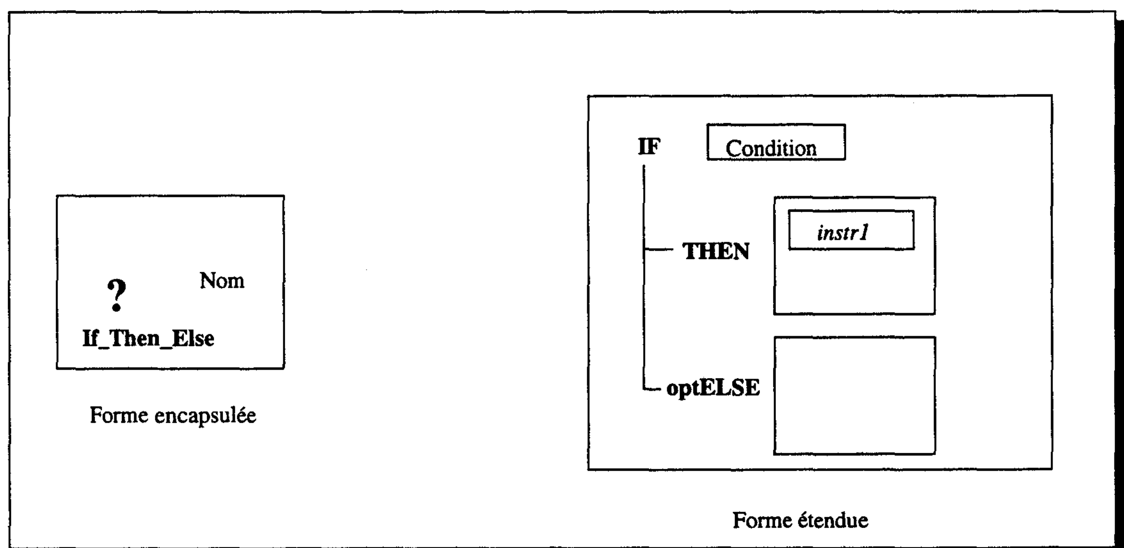


FIG. III.31 - *La représentation graphique de la construction « If_Then_Else »*

La figure III.31 montre l'icône du « `If_Then_Else` » et un exemple du contenu de la boîte. Le « Else » est montré désactivé avec le sous-mot « opt », dès que le programmeur insère quelque chose dans la boîte, le « Else » s'activera (« opt » disparaîtra).

3.1.3 Le constructeur de domaine masqué « Where_Elsewhere »

La construction conditionnelle « `Where_Elsewhere` » est l'équivalent parallèle du « `If_Then-Else` ». Le résultat de l'expression logique de la condition est data-parallèle, par exemple « *dpo*1 > *dpo*2 ». L'ordre de développement et de consommation des TMP est le même que dans le cas du « **on** ». Les TMP de la condition data-parallèle ne sont consommés qu'à la fin du bloc. La figure III.32 montre l'icône du « `Where_Elsewhere` » et un exemple du contenu de la boîte.

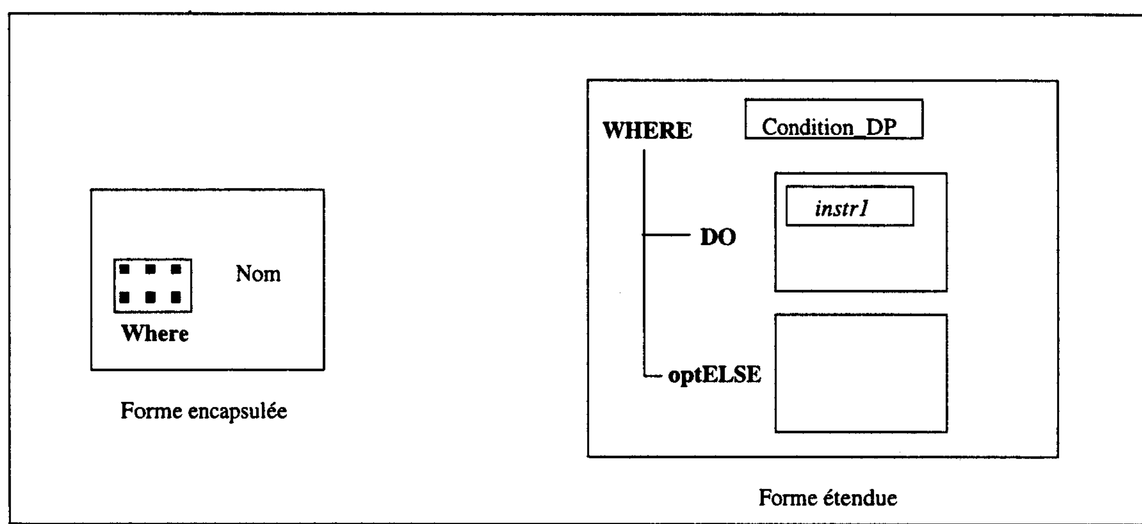


FIG. III.32 - *La représentation graphique de la construction « Where_Elsewhere »*

3.1.4 La construction « While »

La condition est une expression logique dont le résultat est scalaire. La figure III.33 donne la représentation graphique de cette construction. la consommation des TMP et l'ordre de développement de la condition et des instructions sont les mêmes que dans le cas de la construction « `If_Then_Else` ».

3.1.5 La construction « Repeat »

La construction « `Repeat` » est un cas particulier de la boucle « While ». Elle permet de répéter un certains nombre de fois et inconditionnellement une suite d'instructions. La figure III.34 montre sa représentation graphique.

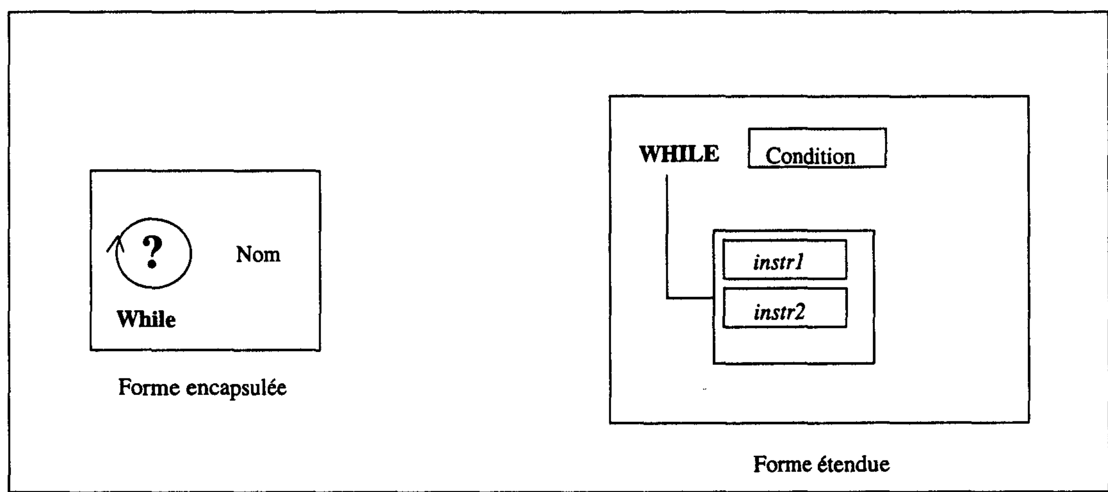


FIG. III.33 - *La représentation graphique de la construction « While »*

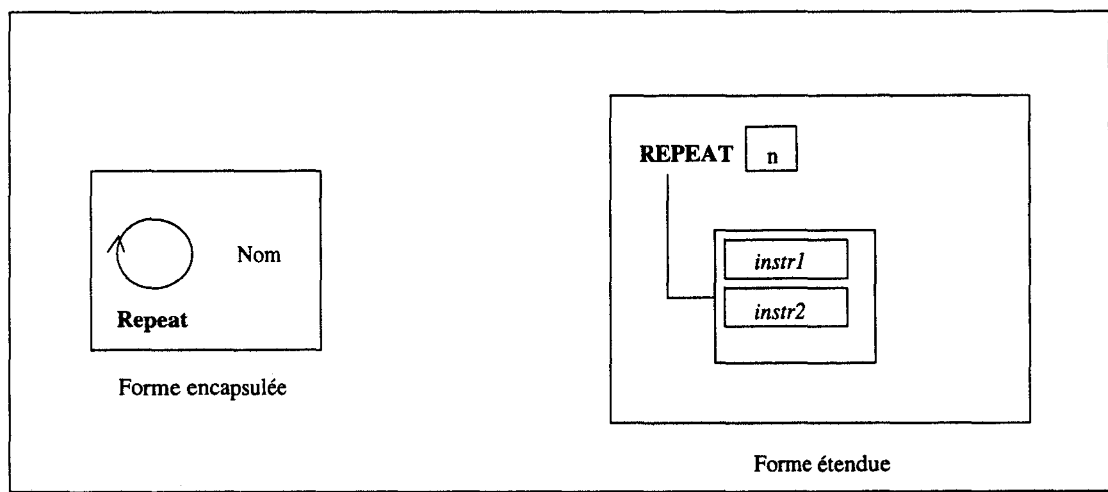


FIG. III.34 - *La représentation graphique de la construction « Repeat »*

3.2 Génération de code pour un bloc d'instructions

Après le développement de chaque instruction du bloc, HELPDraw génèrera dans le « Buffer » le code data-parallèle C-HELP ou HPF correspondant. Lorsque tout le bloc d'instructions est réalisé, HELPDraw insèrera le code data-parallèle équivalent dans l'éditeur de texte à l'endroit indiqué par le programmeur.

4 Conclusion

Nous avons vu dans ce chapitre que la géométrie régulière des données constitue un modèle de programmation compatible avec le développement d'une application scientifique. Nous avons montré à travers HELPDraw que le modèle géométrique est un bon support pour la visualisation. Nous avons montré également que l'utilisateur de HELPDraw peut construire ces instructions data-parallèles par de simples manipulations qui de plus sont en adéquation avec son raisonnement naturel.

Au delà de cette interactivité, notre objectif est de pouvoir générer un code data-parallèle utile, donc exécutable sur des machines massivement parallèles. C'est ce que nous allons montrer dans les deux prochains chapitres: la génération de code C-HELP puis HPF.

Chapitre IV

Génération automatique de code C-HELP

Autant le préciser tout de suite, HELPDraw n'est pas destiner à représenter une couche au-dessus de C-HELP, HPF, ou un quelconque langage. C'est un environnement de programmation visuel qui traduit une conception data-parallèle basée sur le modèle géométrique, en un programme transcodé en C-HELP, ou en HPF. Ceci signifie en particulier que d'une part ce qui est développé sous HELPDraw ne se traduit pas forcément directement dans le langage cible et que d'autre part il ne permet pas nécessairement de réaliser tout ce qui est offert par le langage cible. Ceci est encore plus vrai en ce qui concerne HPF, comme nous le verrons dans le chapitre suivant.

Le langage data-parallèle C-HELP est l'implémentation directe du modèle géométrique HELP dans une syntaxe C. Le développement de ce langage a été dans le seul but de valider l'idée du modèle HELP. C'est pourquoi nous ne reviendrons pas sur les concepts de base; nous nous intéresserons plutôt à la syntaxe du langage. Il existe actuellement un compilateur prototype C-HELP qui produit un code exécutable[1] sur une MasPar (DECmpp [Bla90]). Pour plus de détails sur le langage, nous nous référons à la thèse [Laz95].

Dans ce chapitre, nous décrivons d'abord le langage C-HELP en particulier la syntaxe qu'il offre. Nous verrons ensuite le code généré automatiquement par HELPDraw. Nous verrons le code qui traduit les définitions des objets virtuels du modèle (l'hyper-espace et les DPO), ainsi que les opérations micro et macroscopiques. Concernant les opérations macroscopiques, nous montrons également comment il est possible de réaliser les opérations C-HELP qui n'existent pas en HELPDraw.

1. Le compilateur C-HELP est en fait un traducteur qui produit à partir d'un programme C-HELP un code MPL [Mas91] qui est à son tour compilé sur la MasPar.

1 Description du langage data-parallèle C-Help

Dans cette section, nous allons décrire le langage C-HELP, précisément la syntaxe qu'il offre. Nous avons partagé cette description en trois étapes : d'abord (1) les déclarations des objets manipulés par le modèle HELP (DPO et hyper-espace), ensuite la description des opérations (2) microscopiques et (3) macroscopiques.

1.1 Déclaration des objets du modèle

1.1.1 Les hyper-espaces

Un hyper-espace est un référentiel de points de coordonnées strictement positives. Il est déclaré par le mot clé **hspace** en précisant le nombre de dimensions et la taille sur chaque dimension. La déclaration par exemple d'un hyper-espace « *my_hspace* » bi-dimensionnel de taille (100,100), s'écrit :

```
hspace my_hspace [x=100, y=100] ;
```

C-HELP offre également la possibilité de préciser des informations concernant la distribution des dimensions sur la machine physique. Le programmeur peut préciser un ordre de priorité entre les dimensions qui permettra au compilateur de privilégier le parallélisme, ou le temps de communications, pour chaque dimension de l'hyper-espace. Ces informations vont diriger le compilateur vers une projection efficace. L'ordre de priorité est construit sous forme d'arbre. Le premier niveau de l'arbre représente le niveau le plus prioritaire en terme de parallélisme.

L'exemple de la figure IV.1 montre les projections que peut obtenir le programmeur suivant différents arbres de priorité. On projette un hyper-espace (6×2) sur une machine comportant une grille à (3×2) processeurs. La projection se fait donc en deux couches. Suivant l'arbre de priorité, le programmeur peut obtenir trois projections.

En plus des dimensions principales de l'hyper-espace (dites primaires), C-HELP définit des dimensions secondaires. Ceci dans le but de pouvoir manipuler des objets moins réguliers que les objets parallépipédiques, précisément les diagonales. Les dimensions secondaires sont construites en composant plusieurs dimensions de l'hyper-espace. Voici un exemple de déclaration permettant de définir une dimension secondaire dans un plan [xy] (*cf.* figure IV.2) :

```
hspace my_hspace3  [x=100, y=100,  d=(x,y)] ;
```

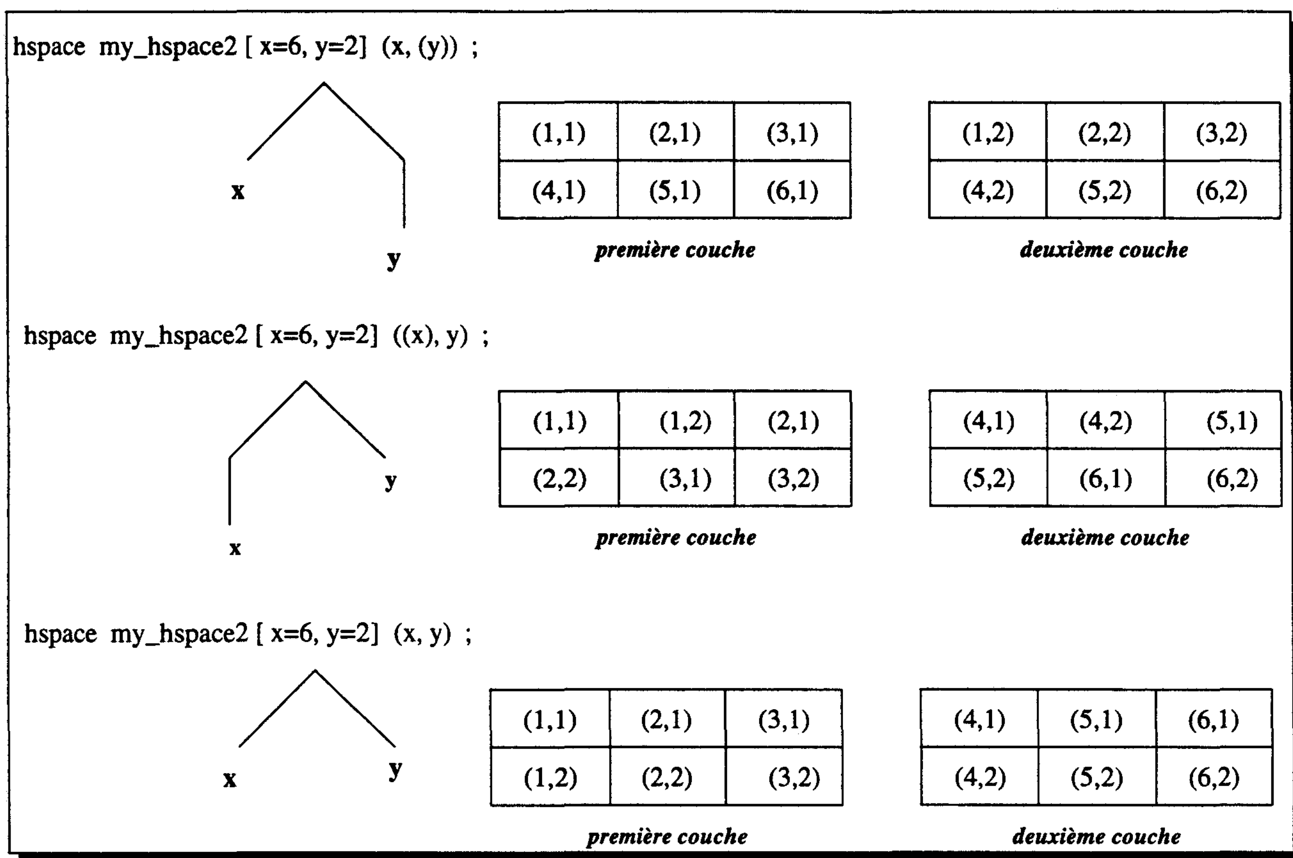


FIG. IV.1 - *Exemples de projections d'un hyper-espace en* C-HELP

1.1.2 Les objets data-parallèles

Par défaut, tout DPO est dynamique par sa taille et par sa position dans l'hyper-espace. Il est possible de déclarer statique un DPO par le mot clé **steady**. Sa position et sa forme dans l'hyper-espace sont alors fixées.

Un DPO est déclaré par le mot clé **dpo** en spécifiant sa taille et sa position sur chaque dimension de l'hyper-espace. La taille et la position peuvent être précisées comme suit :

- *coordonnée de l'origine* : *coordonnée du dernier élément*

- *coordonnée de l'origine* ; *longueur*

- seule la *coordonnée de l'origine* (la longueur vaut alors 1)

- une « * » (l'objet est alors alloué sur toute la dimension).

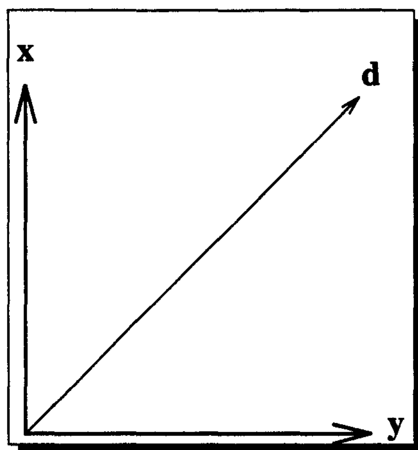


FIG. IV.2 - *Un exemple de dimension secondaire*

La figure IV.3 montre des exemples de DPO déclarés comme suit :

```
hspace    my_hspace4  [x=70, y=70] ;
dpo float my_hspace4  [x=10:40, y=10:60] A ;
dpo float my_hspace4  [x=20;50, y=30;10] B ;
dpo float my_hspace4  [x=20;50, y=20   ] C ;
dpo float my_hspace4  [x=50   , y=*    ] D ;
```

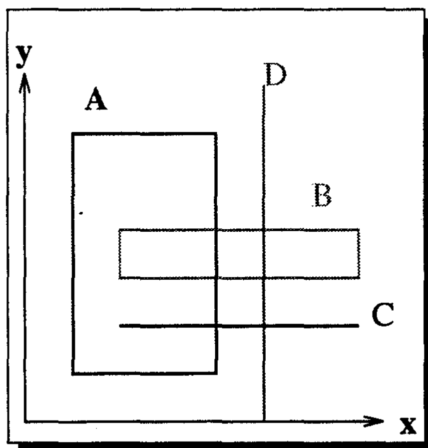


FIG. IV.3 - *Un exemple de déclarations d'objets en* C-HELP

La déclaration d'une diagonale utilisant les dimensions secondaires d'un hyper-espace peut se faire comme suit :

```
hspace    my_hspace5  [x=70, y=70, d=(x,y)] ;
dpo float my_hspace5 [x=20, y=10, d=1;30 ] my_diag ;
```

La diagonale « `my_diag` » de taille 30 est allouée à la position $(20, 10)$.

1.2 Les opérations microscopiques

Nous n'insistons pas sur le niveau microscopique, puisque la syntaxe d'une expression C-HELP est la même qu'en C Standard. La seule différence réside dans les opérations d'affectation (en particulier l'association) ainsi que la spécification des domaines contraints (constructeur `on`) et des domaines d'activité (constructeur `where`). Or, la syntaxe de ces opérateurs et constructeurs, nous l'avons décrite au chapitre précédent lors de la description du modèle HELP (*cf.* chapitre III, section 1.3).

La priorité des opérateurs du langage C est conservée pour l'évaluation d'un segment microscopique. Les constructeurs `on` et `where` ont une priorité inférieure aux opérateurs classiques.

Par exemple, pour le segment conforme `on(A) B+C`, l'addition est déclenchée pour le domaine contraint `A`, Les DPO `B` et `C` doivent englober le domaine d'allocation de `A`. Par contre, lors de l'évaluation du segment conforme `(on(A) B) + C` l'addition est déclenchée sur deux DPO conformes, `C` doit être alloué sur le même domaine d'allocation que `A`.

Une opération macroscopique a une priorité supérieure à tous les opérateurs.

1.3 Les opérations macroscopiques

La syntaxe des opérations macroscopiques disponibles est la suivante :

Déplacements Le programmeur peut spécifier un déplacement d'objet de différentes manières. L'opération « `transabs(dim,off)` » permet de déplacer un objet le long d'une dimension `dim` vers la coordonnée `off` sur cette dimension. L'opération « `transrel(dim,off)` » est la même que la précédente à part que le déplacement `off` est une valeur relative à l'origine de l'objet. Si la coordonnée de l'origine de l'objet est « $orig_{dim}$ » alors `transrel(dim,off)` est équivalente à `transabs(dim, off+orig`$_{dim}$`)`, *cf.* figure IV.4.

Deux autres opérations similaires aux précédentes permettent de spécifier les coordonnées du déplacement sur toutes les dimensions. L'opération « `moveabs(Norig`$_1$`, Norig`$_2$`, ..., Norig`$_n$`)` » déplace l'objet vers une position absolue de l'hyper-espace. La nouvelle position est explicitée par les paramètres « $Norig_i$ ». La dernière opération de déplacement « `moverel(NorigR`$_1$`, NorigR`$_2$`, ..., NorigR`$_n$`)` » déplace l'objet vers une position relative à l'origine de l'objet source. `moverel(NorigR`$_1$`, NorigR`$_2$`, ..., NorigR`$_n$`)` est

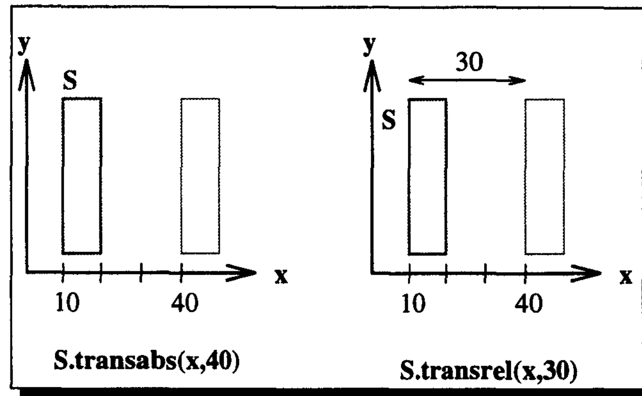


FIG. IV.4 - *Opérations de déplacements en* C-HELP

équivalente à `moveabs(orig`$_1$`+Norig`$_1$`, orig`$_2$`+Norig`$_2$`, ..., orig`$_n$`+Norig`$_n$`)`, où $orig_i$ est l'origine de l'objet source sur la i^e dimension.

Rotations C-HELP définit deux opérations de rotation : **exchabs** et **exchrel**. L'opération « **exchabs(dim1,dim2)** » effectue un échange de coordonnées absolues dans l'hyper-espace. L'autre opération « **exchrel(dim1,dim2)** » effectue par contre un échange de coordonnées dans l'hyper-espace relative à l'origine de l'objet source.

Décalages C-HELP offre deux opérations de décalage : **shifttor** et **flip**. L'opération « **shifttor(dim,off)** » effectue un décalage torique le long d'une dimension **dim** de longueur **off**, vers les coordonnées croissantes si **off** est positif ; vers les coordonnées décroissantes si **off** est négatif. L'opération « **flip(dim)** » effectue une opération de miroir du DPO source sur son intervalle d'allocation le long de la dimension **dim** (le premier élément est échangé avec le dernier, le second avec l'avant dernier, etc.).

Réplications C-HELP offre trois opérations de réplication. La première « **expand(dim)** » réplique l'objet source sur toute la dimension **dim**. Si le DPO source est de taille 1 sur cette dimension, alors le DPO résultat sera alloué sur la dimension complète. Dans le cas contraire, le nombre de réplications est choisi maximal, dans le sens des coordonnées négatives et positives, en conservant les mêmes valeurs sur les points sources (*cf.* figure IV.5(1)). La seconde opération « **expand(dim,d)** » réplique **d** fois l'objet source, vers les coordonnées positives si **d** est positif, vers les coordonnées négatives sinon. La troisième opération « **stretch(dim,d)** » étend le DPO source le long de la dimension **dim**. L'objet résultat possède une largeur **d** fois supérieure à celle du DPO source. Les **d** points de coordonnées les plus petites du DPO résultat reçoivent la valeur du premier point du DPO source, les **d** points suivants reçoivent la seconde valeur, etc. (*cf.* figure IV.5(2)).

Extractions C-HELP définit quatre opérations d'extraction de sous-objets. L'opération « **extrabs(dim,NorigA)** » extrait un sous-DPO de taille 1 sur la dimension **dim**, à partir de la coordonnée absolue **NorigA** dans l'hyper-espace. L'opération « **extrabs(dim,**

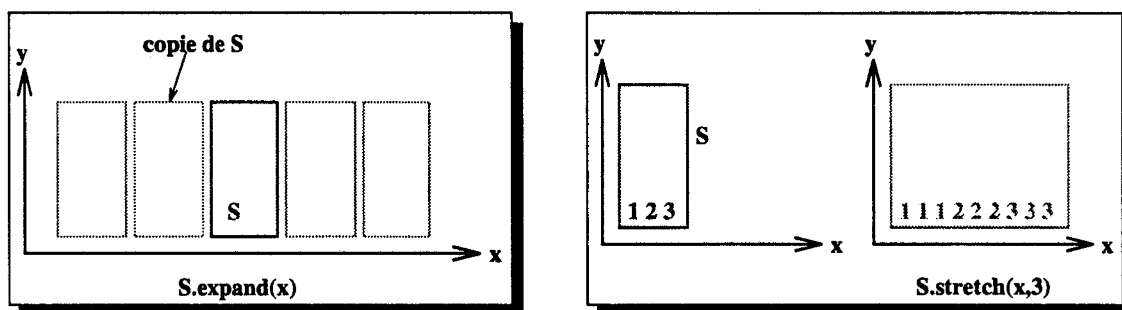


FIG. IV.5 - *Opérations de réplications en* C-HELP

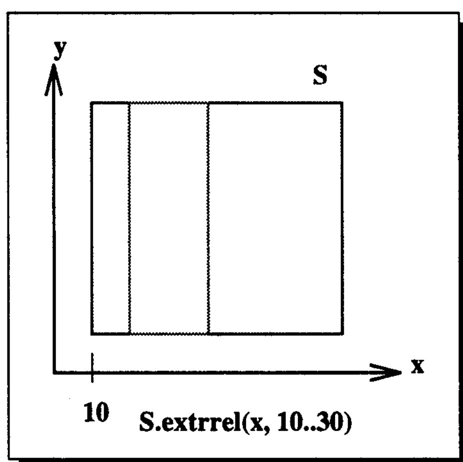


FIG. IV.6 - *Un exemple d'extraction en* C-HELP

`low..up)` » extrait un sous-DPO de taille « $up - low$ », à partir de l'origine absolue `low`. L'opération « `extrrel(dim,NorigR)` » extrait un sou-DPO à partir de l'origine `NorigR` relative à la coordonnée de l'origine du DPO source sur la dimension `dim`. La taille de l'objet extrait sur cette dimension est « $Slen - NorigR + 1$ », où $Slen$ est la taille du DPO source. L'opération « `extrrel(dim,low..up)` » extrait un sous-objet qui va de la coordonnée `low` à la coordonnée `up` relative à l'origine du DPO source sur la dimension `dim` (*cf.* figure IV.6).

Réductions C-HELP offre également des opérations de réduction. Il définit les mêmes opérations que HELPDraw décrites au chapitre précédent (*cf.* 1.4). Les opérations sont : « `reduceadd(dim)` », « `reducemul(dim)` », « `reduceor(dim)` », « `reduceand(dim)` », « `reducemax(dim)` », et « `reducemin(dim)` ».

2 Génération automatique de code

La génération de code C-HELP se fait de manière directe, puisque HELPDraw comme C-HELP sont liés au même modèle de programmation data-parallèle HELP. Que ce soit dans l'un ou l'autre, le programmeur retrouve la notion d'hyper-espace et de manipulations géométriques d'objets data-parallèles. Dans les deux, le programmeur peut exprimer de manière directe la conception géométrique de ses algorithmes. Ainsi, la génération de code automatique faite par HELPDraw est souvent l'expression textuelle directe de ce qui est mis en œuvre sous HELPDraw.

2.1 Déclaration des objets du modèle

2.1.1 Les hyper-espaces

La génération de code concernant les déclarations d'hyper-espaces est directe. Dès que le programmeur introduit les informations concernant l'hyper-espace à travers la boîte de dialogue (*cf.* chapitre III), HELPDraw génère le code dans la syntaxe C-HELP.

Comme nous l'avons vu dans un chapitre précédent, HELPDraw n'utilise absolument pas la notion d'axes secondaires. Or, il est possible que le programmeur soit amené plus tard à définir des diagonales dans son hyper-espace (*cf.* 2.1.2). Ainsi, nous pouvons observer trois alternatives concernant la déclaration d'un hyper-espace :

1. Déclarer l'hyper-espace sans axes secondaires, puis dès que le programmeur définit une diagonale, on change la déclaration de l'hyper-espace (en ajoutant donc un axe secondaire).

2. Déclarer l'hyper-espace en incluant tous les axes secondaires possibles.

3. Au moment de la spécification des informations concernant l'hyper-espace, demander au programmeur s'il compte manipuler des diagonales et sur quel plan.

La première solution est à écarter tout de suite, car une fois qu'un code est généré, le programmeur peut l'insérer n'importe où dans l'éditeur de texte. HELPDraw n'aura plus aucun contrôle sur ce code[2]. Il est donc impossible de le modifier plus tard.

2. Concernant la génération de code, HELPDraw se met au niveau de l'instruction voire au plus bloc d'instructions. Il n'est pas encore lié à un contexte plus global (contexte de fonction ou de tout le programme. L'extension à la gestion de contexte plus global est présentée dans les perspectives (*cf.* chapitre Conclusion)

La deuxième solution est réalisable sans aucune difficulté, mais il est claire que souvent ces axes secondaires ne vont pas servir. Le compilateur C-HELP va les gérer inutilement.

Quant à la troisième solution, elle présente deux problèmes. D'une part c'est contraire au principe de HELPDraw concernant la manipulation des objets peu réguliers qui doivent justement être vus comme tout DPO régulier (il faut les gérer de manière transparente au programmeur), et d'autre part c'est incompatible avec le fonctionnement de HELPDraw lorsqu'il s'agit de générer un code dans un autre langage que C-HELP (exemple : HPF), puisque dans ce dernier cas le programmeur n'aura pas à spécifier quoi que ce soit concernant les diagonales.

Ainsi nous avons choisi la deuxième solution, c'est à dire générer un code incluant la définition de tous les axes secondaires possibles. La première solution pourrait être envisageable dans le futur, dans le cas où HELPDraw pourrait générer un programme complet. Voici un exemple de code C-HELP générer automatiquement par HELPDraw pour la déclaration d'un hyper-espace tri-dimensionnels :

```
hspace my_hspace1 [x=100, y=100, z=100, d1=(x,y), d2=(x,z), d3=(y,z)] ;
```

HELPDraw n'offre pas encore au programmeur le moyen de spécifier la distribution qu'il souhaite afin de privilégier le parallélisme sur telle ou telle dimension. C'est pourquoi il n'y a pas encore de code généré traduisant l'arbre de priorité lors de la déclaration d'un hyper-espace. Ceci ne doit pas cependant poser de problème pour l'intégrer dans HELPDraw. Au niveau de l'interface, il suffit de donner par exemple la possibilité au programmeur de construire visuellement l'arbre de priorité.

2.1.2 Les objets data-parallèles

La génération de code C-HELP concernant les DPO réguliers est directe. Il suffit d'exprimer les caractéristiques introduites par le programmeur à travers la boîte de dialogue (*cf.* chapitre III) dans une syntaxe C-HELP.

Concernant les triangles par contre, comme ils ne sont pas définis dans le langage C-HELP, HELPDraw génère la déclaration d'un DPO carré. Il garde en plus le domaine d'activité définissant le triangle pour pouvoir l'utiliser lors d'une référence à ce triangle. Le code généré automatiquement par HELPDraw concernant le triangle de la figure IV.7, est :

```
dpo int my_hspace[orig1;len1, orig2;len2] my_triangle ;
```

Le domaine d'activité enregistré pour ce triangle est : « $j \leq -i + len1 + 1$ ».

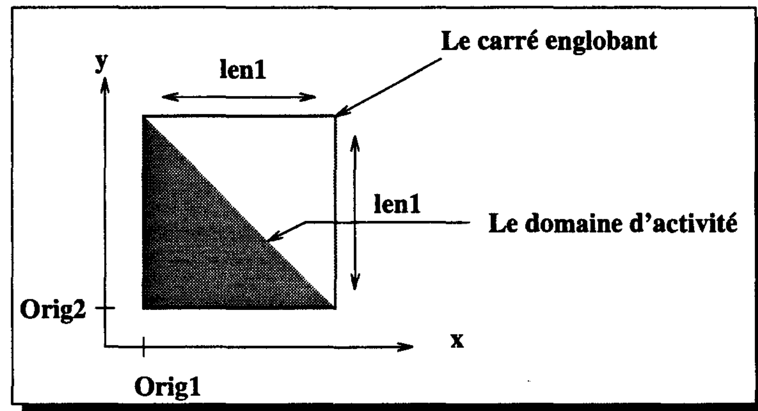


FIG. IV.7 - *Déclaration de triangle en* C-HELP

Étant donné que HELPDraw définit tous les axes secondaires possibles dans un hyper-espace, La génération de code pour la déclaration d'une diagonale devient directe. Une fois que le programmeur spécifie les caractéristiques de la diagonale (en particulier : nom, plan, origine et longueur), HELPDraw génère le code utilisant l'axe secondaire correspondant. La définition par exemple d'une diagonale sur le plan [xy], est :

dpo float my_hspace [x=orig1, y=orig2, $d_{[xy]}$=1;len1] my_diag;

2.2 Application des opérations micro et macroscopiques

L'application de toute opération macroscopique ou microscopique crée un temporaire. HELPDraw associe à chaque temporaire le code C-HELP lui correspondant. Si le programmeur effectue par exemple la somme de deux DPO A et B, HELPDraw associe au temporaire résultant de cette opération, le code : « $A + B$ ». De la même façon, si une autre opération est appliquée à cet objet temporaire, soit l'opération géométrique $expand(x, d)$, un nouveau temporaire est créé. Le code devient alors : « $(A + B).expand(x, d)$ ». Ce traitement est effectué au fur et à mesure pour chaque opérande de l'expression à réaliser. À la fin, lorsque le programmeur applique une opération d'affectation « $=$ » ou d'association « $\leftarrow$ », HELPDraw construit le code final à partir des codes correspondant aux membres droit et gauche et éventuellement le code d'autres constructeurs appliqués à toute l'expression. Si le membre droit est par exemple :

```
(A+B).expand(x,d) - (on(A) C).expand(y,d2)
```

et le membre gauche est Res, l'affectation donnera :

```
Res = (A+B).expand(x,d) - (on(A) C).expand(y,d2)
```

Pour montrer le cas général de la génération de code C-HELP, nous donnons d'abord les règles correspondant aux opérations microscopiques (*cf.* 2.2.1). Nous donnerons ensuite dans la sous-section suivante (*cf.* 2.2.2), les règles correspondant aux opérations macroscopiques.

Ces règles sont utilisées seulement pour la génération de code. Arriver à ce niveau, nous considérons que le programmeur a respecté la syntaxe de HELPDraw lors de l'application de toute opération. Ceci est d'ailleurs vrai, car avant d'arriver à la génération, HELPDraw aurait déjà vérifié et accepté l'application de l'opération. Cette vérification syntaxique se fait lors de l'interprétation visuelle.

2.2.1 Les opérations microscopiques

Les opérations microscopiques ainsi que les constructeurs `on` et `where` sont traduits directement, comme nous l'avons déjà montré en ce qui concerne les historiques maintenus par HELPDraw (*cf.* chapitre III : 2.5). Seules des exceptions existent lors de la manipulation des triangles qui ne sont pas définis en C-HELP.

La table IV.1 donne les règles utilisées pour la génération de code concernant les opérations microscopiques. Le code est construit au fur et à mesure de l'application des opérations. Initialement le code d'un opérande est le nom du DPO sélectionné (exemple : code = `dpo1`). L'application des opérations est presque la même que celle déjà décrite au chapitre précédent pour la construction des historiques. La seule différence réside dans la manipulation des triangles.

Lorsque le programmeur précise un domaine contraint qui est un triangle (exemple : `on(my_triangle)`), HELPDraw utilise la règle R_3 et applique en même temps la fonction g (sous-règle r_1). C'est la deuxième alternative de g qui est déclenché permettant d'ajouter le masque définissant le triangle. Ceci est dû au fait qu'un triangle est déclaré en C-HELP comme un DPO carré. Le domaine spécifié par `on(my_triangle)` est de ce fait tout le carré englobant le triangle. D'où, l'ajout du domaine masqué `where(masque-triangle)` dans la sous-règle r_1 (seconde alternative) qui va définir l'activité seulement sur les points du triangle. Le code généré automatiquement pour `on(my_triangle)`, est :

```
on(my_triangle) where(masque-triangle)
```

Si `my_triangle` est le triangle de la figure IV.7, ce code s'écrira[3] :

3. `crel(DPO,dim)` est une fonction intrinsèque C-HELP. Elle renvoi pour chaque point du DPO passé en paramètre, la valeur de la coordonnée de ce point sur la dimension `dim` relativement à l'origine du DPO.

TAB. IV.1 - *Les règles utilisées pour la génération de code* C-HELP *concernant les opérations microscopiques*

- Tout DPO a deux propriétés : c et p
  - c : son code — c(arg) → le code associé à *arg*
  - p : sa priorité (*cf.* III.23) — p(arg) → la priorité associée à *arg*
- Tout DPO variable a ses propriétés initialisées à :
  $c = nom$(DPO)
  $p = p($ nom(DPO))
- Pour toute opération microscopique, calculer les propriétés du résultat en appliquant l'une des règles suivantes :

Soit la fonction f qui renvoie l'une des deux valeurs selon la priorité de **op** :

f(arg, op)	⟶	(c(arg))	<u>**si**</u> p(arg) < p(op)
	⟶	c(arg)	<u>**sinon**</u>

R_1	[*opBinaire*[a], *arg1*, *arg2*]	: $c =$	f(arg1,opBinaire) **opBinaire** f(arg2,opBinaire)
		$p =$	p(opBinaire)
R_2	[*opUnaire*, *arg*]	: $c =$	***opUnaire*** f(arg, opUnaire)
		$p =$	p(opUnaire)
R_3	[*désactiver-on*, *domaine*, *arg*]	: $c =$	g(domaine) c(arg)
		$p =$	p(on)

r_1	g(domaine)	⟶	**on**(c(domaine))	<u>**Si**</u> *domaine* $\neq$ *triangle*
		⟶	**on**(c(domaine)) **where**(c(masque-triangle))	<u>**Sinon**</u>

R_4	[*désactiver-where*, *domaine*, *arg*]	: $c =$	**where**(c(domaine)) c(arg)
		$p =$	p(where)
R_5	[*désactiver-segt-non-contraint*, *arg*]	: $c =$	c(arg)
		$p =$	p(on)
R_6	[*exception-triangle*[b], arg_1, arg_2]	: $c =$	$c(arg_1)$ **= where**(c(masque-arg_2)) $c(arg_2)$
		$p =$	p(=)

[a] *opBinaire* différent de celui de la règle R_6.
[b] affectation de triangle sans « on » au niveau du segment principal, mais il y a un opérateur d'exception (*cf.* explications).

```
on(my_triangle)
    where( crel(my_triangle,y) ≤ -crel(my_triangle,x) + len1 + 1 )
```

Un problème se pose cependant lorsque l'expression manipule des triangles, sans qu'il y ait un domaine contraint explicite au niveau du segment conforme principal, par exemple :

$$triangle = triangle1 + triangle2 - triangle3/triangle4$$

Le problème ne se pose pas ici pour l'affectation ou l'addition, puisque ces opérations appliquées aux carrés englobants (sans masque) ne causent aucun problème. En ce qui concerne la division par contre, il n'est pas possible d'effectuer l'opération sans masque car il se pourrait qu'il y ait des éléments nuls qui évidemment n'appartiennent pas au triangle mais au carré englobant. Parmi les opérations arithmétiques et logiques, seule la division engendre ce problème. Nous retrouvons ce même problème avec certaines fonctions microscopiques offertes par HELPDraw telles que log, ctg, etc. (*cf.* éditeur d'expression).

Concrètement, ce cas se présente lorsque les deux membres d'une affectation sont des triangles alors qu'aucun constructeur « **on** » global n'est précisé, ni global au membre droit de l'affectation, par exemple

$$triangle = on(t) \ \ldots$$

ni global à toute l'expression, par exemple

$$on(t) \ triangle = \ \ldots$$

Afin de prendre en compte ce cas, dès qu'une division ou une des fonctions microscopiques telles que celles que nous avons citées est appliquée, HELPDraw enregistre une information, soit *exception-triangle* évalué comme suit :

Si *(argument == triangle) et (pas de « **on** » global) et (opérateur ou fonction d'exception)*
alors *exception-triangle = vrai*

Ensuite, lors de la génération de code correspondant à l'affectation « = », il applique la règle R_6 permettant de générer, en plus du code d'une affectation habituelle, un `where` qui masque les points n'appartenant pas au triangle.

2.2.2 Les opérations macroscopiques

Nous utilisons le même principe employé au chapitre précédent pour la construction des historiques correspondant aux opérations macroscopiques, *cf.* III :2.6.2. Bien qu'ici d'une part

le nom des opérations sera celui de C-HELP et d'autre part certaines opérations ne se traduisent pas directement en C-HELP, ce qui nécessitera la combinaison de plusieurs opérations C-HELP. Les règles de la table IV.2 montrent comment HELPDraw construit automatiquement le code à générer. Nous utilisons des règles similaires à celles qui servaient dans le chapitre précédent à la construction des historiques.

Ces règles construisent le code correspondant à un opérande. À la suite de chaque opération géométrique effectuée, HELPDraw applique la règle R_6. Le code du nouveau temporaire résultat est donc construit par la dérivation des deux fonctions f et g. Selon la priorité de l'objet source de l'opération, HELPDraw choisira de dériver f par sa première ou sa seconde alternative (l'objet est mis ou non entre parenthèses). Ensuite selon le nom de l'opération effectuée, la fonction g est dérivée par l'une des sous-règles r_i $(i = 1, 14)$, dont nous donnons la description ci-après.

Déplacements En HELPDraw, le programmeur opère un déplacement absolu de son DPO à l'intérieur de l'hyper-espace. Ce déplacement peut changer les trois coordonnées de l'origine en même temps (*cf.* figure IV.8). C'est pourquoi HELPDraw génère automatiquement l'opération « **moveabs** » de C-HELP.

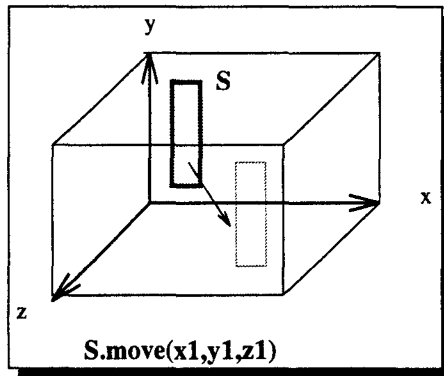


FIG. IV.8 - *Déplacement vers la position (x1, y1, z1)*

Si le code construit par HELPDraw avant l'application d'un déplacement est par exemple :

```
A + B
```

Ce code évoluera en utilisant la sous-règle r_1 (évidemment après la règle R_6) en :

$$\texttt{(A + B).moveabs(}x_1,\ y_1,\ z_1\texttt{)}$$

où (x_1,y_1,z_1) est la nouvelle position du temporaire.

HELPDraw génère l'opération « **moveabs** », même si le programmeur déplace son objet seulement sur une dimension. On suppose que s'il est plus efficace d'utiliser l'opération

TAB. IV.2 - *Les règles utilisées pour la génération de code* C-HELP *concernant les opérations macroscopiques*

- Rappelons la fonction f :

$$
\begin{array}{llll}
f(\text{arg}, \text{op}) & \longrightarrow & (\ c(\text{arg})\ ) & \underline{\text{si}}\ p(\text{arg}) < p(\text{op}) \\
 & \longrightarrow & c(\text{arg}) & \underline{\text{sinon}}
\end{array}
$$

- Pour toute opération macroscopique *macro*, calculer les propriétés du résultat en appliquant la règle suivante :

R_6 [*macro*] : $c =$ $f(\text{arg}, .)\ .\ g(\text{macro})$
$p =$ $p(.)$

r_1	g(move)	⟶	**moveabs(x_1, y_1, z_1)**
r_2	g(rotate)	⟶	**exchrel(dim1, dim2)**
r_3	g(exchange)	⟶	**exchabs(dim1, dim2)**
r_4	g(rotate_todiag)	⟶	**exchrel($\text{dim}_{1\|2}$, $d_{[dim1,dim2]}$)**
r_5	g(rotate_toline)	⟶	**exchrel(d, dim)**
r_6	g(rotate_triangle)	⟶	**exchrel(dim1, dim2)**
r_7	g(circular_shift)	⟶	**shifttor(dim, off)**
r_8	g(mirror)	⟶	**flip(dim)**
r_9	g(expand(dim,d))	⟶	**expand(dim, d)**
r_{10}	g(expand(dim))	⟶	**expand(dim,⌈TmpLen/Slen⌉)**
r_{11}	g(extract_subdpo)	⟶	**extrrel(x, $\text{Norig}_x..\text{up}_x$)** [a] **.extrrel(y, $\text{Norig}_y..\text{up}_y$).extrrel(z, $\text{Norig}_z..\text{up}_z$)**
r_{12}	g(extract_diag)	⟶	**extrrel(dim1, $\text{Norig}_1..\text{up}_1$) .extrrel(dim2, $\text{Norig}_2..\text{up}_2$) .extrrel($d_{[dim1,dim2]}$, 1)**
r_{13}	g(extract_triangle)	⟶	**extrrel(dim1, $\text{Norig}_{dim1}..\text{up}_{dim1}$) .extrrel(dim2, $\text{Norig}_{dim2}..\text{up}_{dim2}$)** [b]
r_{14}	g(reduceOP [c])	⟶	**reduceOP(dim)**

[a] $\text{up}_i = Nlen_i - Norig_i + 1$
[b] L'extraction d'un triangle nécessite également l'enregistrement du masque correspondant à ce nouveau triangle
[c] reduceOP : reduceadd, reducemul, reduceor, reduceand, reducemax, reducemin

« **transabs** », le compilateur C-HELP doit être en mesure d'optimiser le code lui-même.

Rotations En ce qui concerne les opérations de rotation d'objets réguliers : « **rotate** » et « **exchange** » (*cf.* figure IV.9), HELPDraw les traduit directement en leur équivalent en C-HELP. La première est traduite par « **exchrel** » (sous-règle r_2), et la seconde par « **exchabs** » (sous-règle r_3).

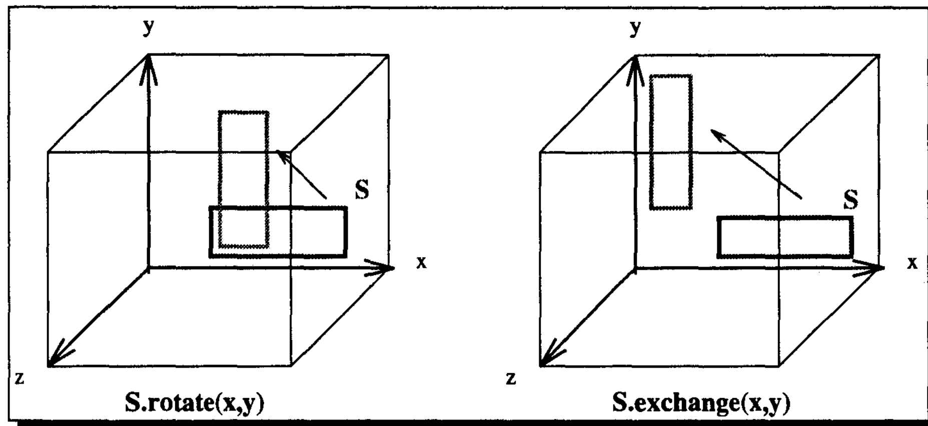


FIG. IV.9 - *Exemples de rotations*

Les opérations de rotation d'une ligne vers une diagonale « **rotate_todiag** » et d'une diagonale vers une ligne « **rotate_toline** », sont traduites par l'opération « **exchrel** » en utilisant la dimension secondaire où la diagonale est (ou va être) allouée (sous-règles r_4 et r_5).

Si une ligne est définie dans un plan [dim$_1$,dim$_2$], parallèlement à la dimension dim_1, sa rotation vers une diagonale se traduit par (sous-règle r_4) :

$$\texttt{exchrel}(\texttt{dim}_1,\ \texttt{d}_{[dim1,dim2]})$$

où « $\texttt{d}_{[dim1,dim2]}$ » est la dimension secondaire définie dans le plan [dim$_1$,dim$_2$]. De même si une diagonale est allouée sur un axe secondaire « **d** », sa rotation vers une ligne parallèle à l'axe **dim**, se traduit par (sous-règle r_5) :

```
exchrel(d,dim)
```

Étant donné que le type d'objet triangle n'existe pas en C-HELP, HELPDraw manipule un objet carré. Ainsi, la rotation d'un triangle se traduit par la rotation du carré englobant. HELPDraw calcule également le nouveau domaine d'activité pour une éventuelle utilisation postérieure du triangle (*cf.* figure IV.10).

La traduction automatique de la rotation du triangle T de la figure IV.10 se traduit par (sous-règle r_6) :

```
T.exchrel (x,y)
```

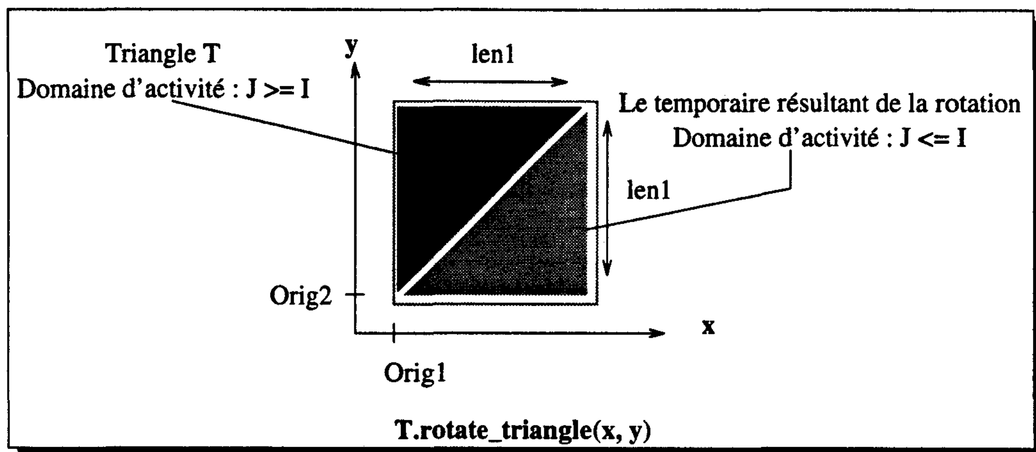


FIG. IV.10 - *Rotation d'un triangle*

Si par la suite ce triangle temporaire résultat est utilisé dans une opération microscopique, réalisant par exemple la somme de deux DPO `A` et `B` dans un domaine contraint par le triangle, HELPDraw génère automatiquement le code suivant :

```
on(T.exchrel(x,y))
    where(crel(T.exchrel,y)<=crel(T.exchrel,x))  A+B
```

Évidemment si l'objet triangle était implémenté en C-HELP, HELPDraw n'aurait pas généré le masque explicité par le constructeur `where`. Nous retenons en outre l'abstraction qu'offre HELPDraw aux utilisateurs quant à la manipulation des triangles.

Décalages Les opérations de décalage offertes par HELPDraw « `circular_shift` » et « `mirror` » se traduisent directement par leur équivalent en C-HELP. La première est traduite par « `shifttor` » (sous-règle r_7), et la seconde par l'opération « `flip` » (sous-règle r_8).

Réplications HELPDraw offre deux façons de réaliser une expansion, une expansion d'une longueur positive déterminée : « `expand(dim,d)` » ou d'un nombre maximal jusqu'à la fin de la dimension : « `expand(dim)` ». Les deux opérations se traduisent par une même opération C-HELP : « `expand` ». La traduction de la première est directe en utilisant la sous-règle r_9. En ce qui concerne la seconde opération, HELPDraw doit calculer le nombre de copies pouvant être allouées, de la position de l'objet source jusqu'à la fin de la dimension (*cf.* figure IV.11). C'est pourquoi il utilise la sous-règle r_{10}, où il précise le nombre de copies comme la division entière de la longueur de l'objet résultat par celle de l'objet source ($\lceil \mathrm{TmpLen/Slen} \rceil$). La longueur du temporaire « *TmpLen* » est calculée lors de l'interprétation visuelle.

C-HELP offre une autre opération d'expansion « `expand(dim)` » répliquant l'objet sur toute la dimension `dim`. La question qui se pose est « est-il possible de la réaliser sous

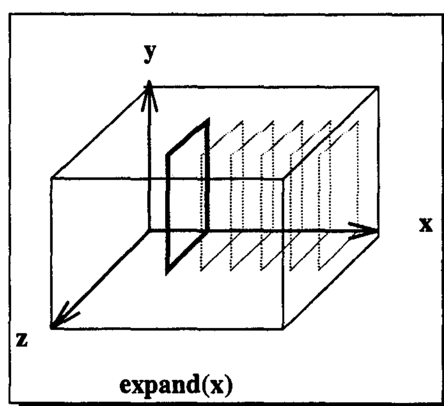


FIG. IV.11 - *L'opération de réplication « expand(x) » de HELPDraw*

HELPDraw? ». La réponse est « oui ». Le programmeur doit déplacer le DPO et effectuer ensuite un **expand**. Pour réaliser par exemple l'équivalent de **expand(x)** du C-HELP, le programmeur doit faire :

$$\texttt{S.move(mod(Sorig}_x\texttt{-1,Slen}_x\texttt{)+1, Sorig}_y\texttt{, Sorig}_z\texttt{).expand(x)}$$ [4]

où *Sorig* et *Slen* sont respectivement l'origine et la longueur du DPO source.

Par contre en ce qui concerne le deuxième type d'opération de réplication offert par C-HELP : l'opération « **stretch(dim,d)** », elle n'est pas réalisable directement sous HELPDraw. Le programmeur doit appliquer plusieurs **extract**, **move**, et **expand** afin d'y arriver, ce qui représente un nombre important de manipulations. Le plus simple serait de l'intégrer en tant qu'opération dans HELPDraw. Ceci n'est pas difficile ; concernant l'interface graphique, l'opération peut être activée par un menu comme les autres opérations de réplications ; la génération de code se fera via une règle similaire à celle d'un **expand**.

Extractions Malheureusement C-HELP n'inclut pas une opération équivalente à « **extract_subdpo** » de HELPDraw permettant d'extraire directement un sous-objet. Pour générer le code correspondant à l'extraction d'un sous-objet régulier, HELPDraw peut être amené à utiliser jusqu'à trois fois l'opération « **extrrel(dim, Norig..up)** ». Pour l'extraction du rectangle de la figure IV.12, HELPDraw génère le code (sous-règle R_{11}) :

$$\texttt{S.extrrel(x,Norig}_1\texttt{..(Nlen}_1\texttt{-Norig}_1\texttt{+1))}$$
$$\texttt{.extrrel(y,Norig}_2\texttt{..(Nlen}_2\texttt{-Norig}_2\texttt{+1))}$$

HELPDraw rencontre le même problème pour l'extraction d'une diagonale « **extract_diag** ». Il doit générer successivement trois opérations « **extrrel** ». Les

4. mod(A,B) = A modulo B.

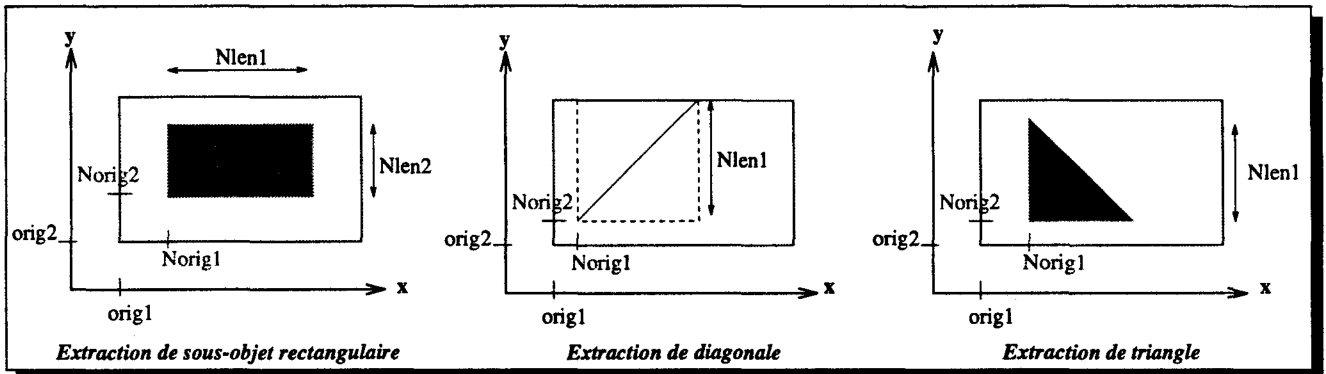


FIG. IV.12 - *Exemples d'extractions de sous-objet*

deux premières permettent d'extraire le plus petit carré englobant la diagonale, puis la troisième extrait la diagonale elle-même. Pour l'extraction de la diagonale de la figure IV.12, HELPDraw génère le code (sous-règle r_{12}) :

```
S.extrrel(x,Norig₁..(Nlen₁-Norig₁+1)).extrrel(y,
    Norig₂..(Nlen₁-Norig₂+1)).extrrel(d_xy,1))
```

Étant donné que le triangle est représenté en C-HELP par le carré englobant, son extraction « **extract_triangle** » correspond donc à celle du carré. HELPDraw détermine en plus le domaine d'activité définissant le triangle extrait pour d'éventuelles utilisations postérieures, comme nous l'avons déjà vu avec l'opération de rotation de triangle. Pour l'extraction du triangle de la figure IV.12, HELPDraw génère le code (sous-règle r_{13}) :

```
S.extrrel(x,Norig₁..(Nlen₁-Norig₁+1))
 .extrrel(y,Norig₂..(Nlen₁-Norig₂+1))
```

Ceci, en enregistrant de manière interne le domaine d'activité définissant le triangle : « $j \leq -i + Nlen1 + 1$ ».

Réductions Les opérations de réduction (**reduceadd, reducemul, reduceor, reduceand, reducemax, et reducemin**) offertes par HELPDraw sont les mêmes que celles définies en C-HELP, c'est pourquoi leur traduction est directe (sous-règle r_{14}).

3 Exemple de génération de programme

Nous avons choisi, pour illustrer la génération de code C-HELP, un exemple d'algorithme très simple : la multiplication de matrices (N*N) : $C = A * B$. Nous décrirons d'abord la conception géométrique de l'algorithme (*cf.* 3.1), nous traduirons ensuite cette conception dans un algorithme géométrique (*cf.* 3.2), nous donnerons enfin le code C-HELP généré auto-

matiquement (*cf.* 3.3).

3.1 Conception géométrique

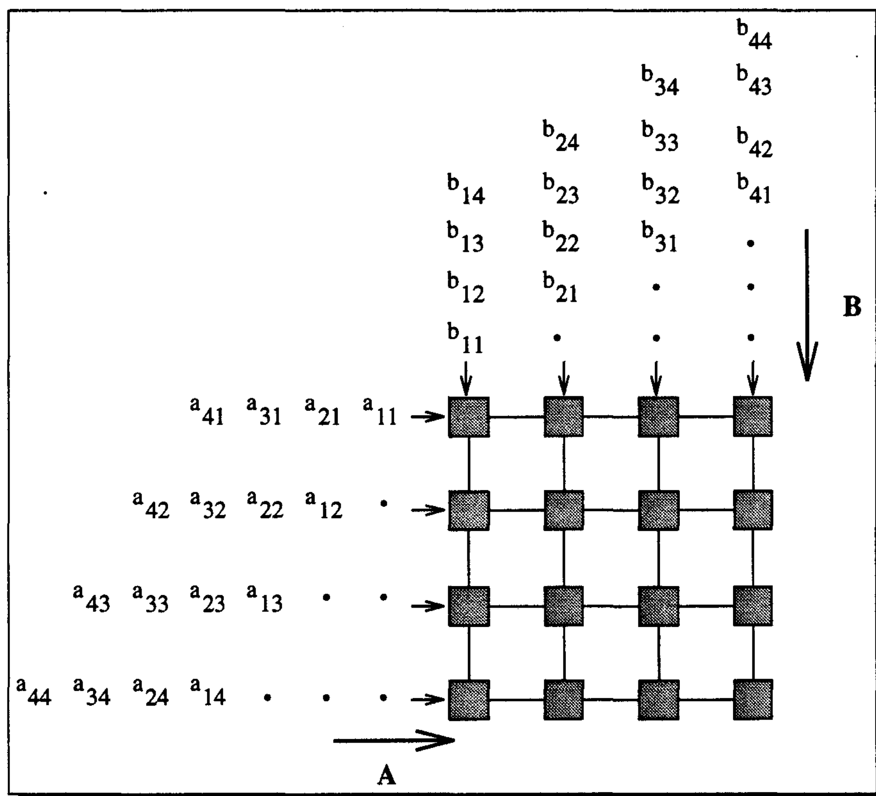


FIG. IV.13 - *Multiplication de matrices sur une grille 2-D*

L'idée est de faire glisser les deux parallélogrammes (représentant les matrices A et B) l'un contre l'autre en faisant les multiplications et les additions nécessaires (figure IV.13). L'idée de cet algorithme est inspirée de [BBES91] et [Lei92].

Pour simplifier la conception de cet algorithme, prenons un exemple de matrices (3×3), figure IV.14. L'algorithme est réalisé en deux étapes principales : (*i*) d'abord l'initialisation de la matrice C à partir de A et B placées sous formes de parallélogrammes dans la grille de processeurs, (*ii*) puis le calcul de C en glissant pas à pas les deux parallélogrammes A et B.

Première étape La première étape est d'obtenir le placement des éléments ayant un fond gris dans la figure IV.14 sur la grille de processeurs (3×3). Ceci est réalisé en faisant un décalage torique. Chaque ligne de la matrice A est décalée à gauche d'un pas égal au « numéro de la ligne moins un », et chaque colonne de B est décalée vers le haut d'un pas égal au « numéro de la colonne moins un » (*cf.* figure IV.15).

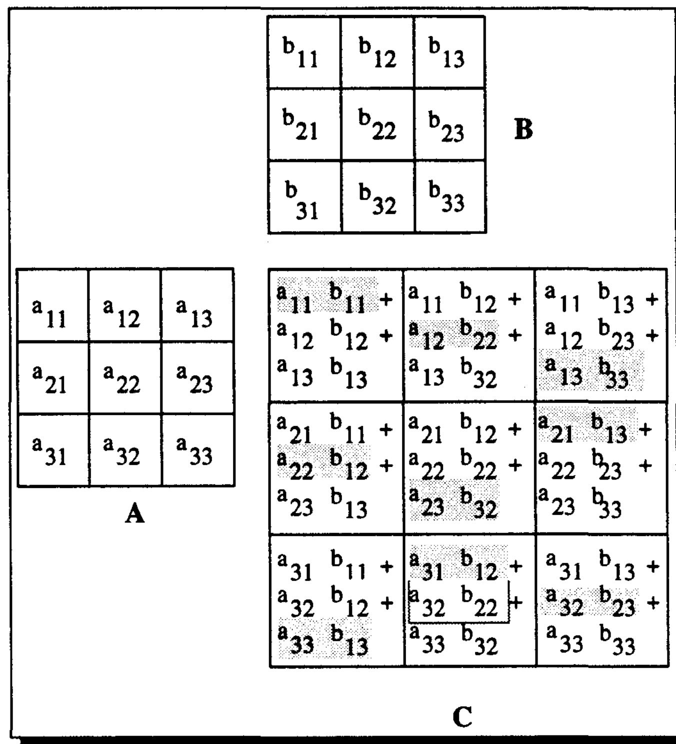


FIG. IV.14 - *Les valeurs c_{ij} de la matrice que nous souhaitons obtenir à la fin de la multiplication de A par B ($C = A * B$)*

Le choix des éléments est fait de telle sorte qu'il n'y ait pas un élément dans deux endroits en même temps. Ce qui permettra de faire leur multiplication en une seule fois.

Seconde étape Pour le reste de toutes les représentations de la figure IV.14 (éléments encadrés puis ceux non encadrés), nous ne faisons que décaler chaque ligne de A (resp. colonne de B) d'un pas vers la gauche (resp. vers le haut), puis faire la multiplication que nous ajouterons à chaque fois à la somme précédemment obtenue (*cf.* figure IV.16).

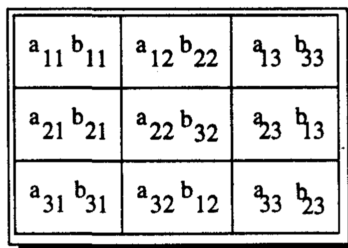


FIG. IV.15 - *Rangement des données après la première étape*

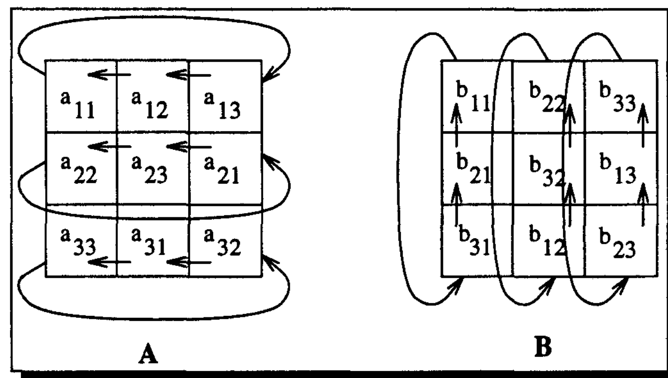


FIG. IV.16 - *A chaque étape, décalage de A vers la gauche et de B vers le haut*

3.2 L'algorithme géométrique

/ Multiplication de matrices */*

/ Initialisation */*

Soient les matrices A, B et C (N×N) sur une grille N×N

/ première étape */*
A = décalage_torique_vers_la_gauche de A (pas=n^o ligne -1)
B = décalage_torique_vers_le_haut de B (pas=n^o colonne -1)
C = $A * B$

/ seconde étape */*
pour $i = 1$ à $(N - 1)$
faire
    A = décalage_torique_vers_la_gauche de A (pas= 1)
    B = décalage_torique_vers_le_haut de B (pas= 1)
    */*cumul des produits*/*
    C = $C + A * B$
fait

3.3 Le code C-HELP généré automatiquement

```
/* Initialisation */
hspace grille [x=N, y=N] ;
DPO float grille [x=1;N,y=1;N] A ;
DPO float grille [x=1;N,y=1;N] B ;
DPO float grille [x=1;N,y=1;N] C ;


/* Première étape */
A = A.shifttor(x, - (crel(A,y)-1)) ;
B = B.shifttor(y, - (crel(B,x)-1)) ;
C = A * B ;


/* Seconde étape */
for(i=1; i<=N-1; i++) {
  A = A.shifttor(x,-1) ;
  B = B.shifttor(y,-1) ;
  C = C + A * B ;
}
```

Remarque Dans les opérations de décalage `shifttor` de la première étape, l'argument de déplacement est un tableau `crel(A,y)`. Ce type d'argument ne figure pas dans les spécification de C-HELP [Laz95], mais cela est pris en compte dans la version actuelle du compilateur.

4 Conclusion

Nous avons montré dans ce chapitre qu'il est possible aux utilisateurs de HELPDraw de développer un code data-parallèle C-HELP sans se soucier du langage lui même. HELPDraw utilise de simples règles pour exprimer les manipulations de l'utilisateur en un code C-HELP. Nous avons vu également l'abstraction qu'il fait d'une part pour effectuer certaines opérations (par exemple l'extraction directe d'un sous-objet) et d'autre part pour manipuler des objets peu réguliers (ici les diagonales et les triangles).

HELPDraw et C-HELP sont issus du même modèle de programmation data-parallèle: le modèle géométrique HELP. C'est pourquoi la traduction des manipulations sous HELPDraw en un code C-HELP a été quasiment directe. Sera t-elle aussi directe lorsqu'il faudra générer du code dans d'autres langages, en particulier HPF qui appartient à un modèle de programmation orthogonal au notre? C'est ce que nous allons voir dans le chapitre suivant, où nous étudierons notamment le passage un modèle basé sur le référentiel à un modèle basé sur les indices.

Chapitre V

Génération automatique de code HPF

HELPDraw est conçu afin de faciliter le développement de codes data parallèles et de rendre les machines parallèles accessibles à un grand nombre de scientifiques. Il est ainsi tout à fait naturel de générer un code dans le langage le plus utilisé par cette communauté de scientifiques (Fortran data-parallèle). HELPDraw propose le langage HPF « High Performance Fortran ». Les utilisateurs de HELPDraw bénéficient des avantages de HPF, en particulier la portabilité du code produit, sans se soucier du langage lui-même ou de l'architecture cible.

Nous proposons dans ce chapitre les mécanismes utilisés par HELPDraw pour générer automatiquement un code HPF. Nous présentons au début une description de HPF nous permettant d'introduire les nouveautés du langage par rapport à ses prédécesseurs. Cette description nous permettra d'étudier les différentes possibilités de génération de codes que HELPDraw pouvait adopter (*cf.* 2.1). Nous montrons ensuite la solution choisie pour HELPDraw. Celle-ci se traduit par la nécessité de passer de la notion du point dans notre modèle à la notion d'indices de HPF (*cf.* 2.2). Nous étudierons après les mécanismes d'optimisation mis en œuvre par HELPDraw permettant de regrouper l'ensemble des opérations géométriques constituant un opérande en une seule étape de communication. Ceci nécessite en particulier l'étude d'une fonction de description f permettant de retrouver les éléments sources correspondant aux éléments d'un quelconque opérande (*cf.* 2.4.1). La fonction f permettra à HELPDraw de ne générer qu'un seul code, explicité par un FORALL, pour toute l'expression data-parallèle. Nous complèterons ensuite la génération automatique des expressions par l'étude du niveau microscopique, en particulier l'interprétation des segments conformes par de simples opérations d'extractions (*cf.* 2.5). Pour mieux illustrer tout cela, nous présentons à la fin du chapitre le code à générer automatiquement pour un exemple data-parallèle concret (inversion de matrice de Gauss-Jordan), *cf.* 3. Nous tenons à préciser cependant que la production automatique du code HPF n'est pas entièrement implémentée.

1 Description du langage HPF

Depuis son apparition, il y a plus de trois décennies, Fortran a été le langage choisi pour la programmation scientifique sur les machines séquentielles. L'exploitation de la capacité maximale des nouvelles architectures, requiert cependant plus d'informations que ce qui est fourni par un programme ordinaire Fortran 77 (séquentiel) ou Fortran 90 (data-parallèle). Fournir ces informations est devenu l'objectif de HPF.

Les buts de HPF sont justement de définir des extensions au standard Fortran 90 pour supporter les trois points suivants, tels que définis dans le draft [For93] :

1. Une référence aux données plus générale que dans les assignations tableaux : pour réaliser ce but, HPF ajoute l'instruction ou la construction data-parallèle FORALL (*cf.* 1.7) ainsi que plusieurs fonctions intrinsèques nouvelles.

2. Une haute performance sur des machines aussi bien SIMD que MIMD : en d'autres termes, il faut assurer la portabilité d'un code à travers une variété de machines parallèles. Ceci en préservant en même temps le niveau d'efficacité d'un programme porté d'une machine à une autre ayant un nombre comparable de processeurs. Pour ce second but, HPF se base sur des caractéristiques de distribution de données avec quelques autres directives.

3. Et enfin un affinement de code ciblant des machines précises : ce but doit être atteint par des fonctions liées à la machine cible (extrinsèques) ; ces fonctions permettent l'accès aux particularités de la machine.

Pour augmenter les performances de la machine, le nombre de communications inter-processeurs doit être réduit. Pour cela HPF propose des directives[1] de placement qui autorisent le programmeur à gérer de manière fine l'allocation des tableaux sur les processeurs. Il y a deux niveaux de placement, *cf.* figure V.1. Les objets sont premièrement alignés les uns par rapport aux autres (directives ALIGN ou REALIGN) ; ensuite ce groupe d'objets est distribué sur un arrangement de processeurs abstraits (directives DISTRIBUTE ou REDISTRIBUTE). Le programmeur définit l'arrangement de processeurs qui correspond le mieux à la distribution de ses données : c'est une machine virtuelle. Cet arrangement de processeurs abstraits est déclaré par la directive PROCESSORS. Le passage ensuite des processeurs abstraits aux processeurs physiques dépendra de l'implémentation et de la machine cible.

1. Une directive est une extension d'un langage donné (Fortran en général) qui se présente sous la forme d'un commentaire ; elle est ainsi ignorée des compilateurs auxquels elle n'est pas destinée. Une directive ne porte pas de sémantique en elle-même, mais c'est une indication au compilateur afin que celui-ci puisse générer du code plus efficace. En HPF, une directive est spécifiée par exemple avec « !HPF$ ».

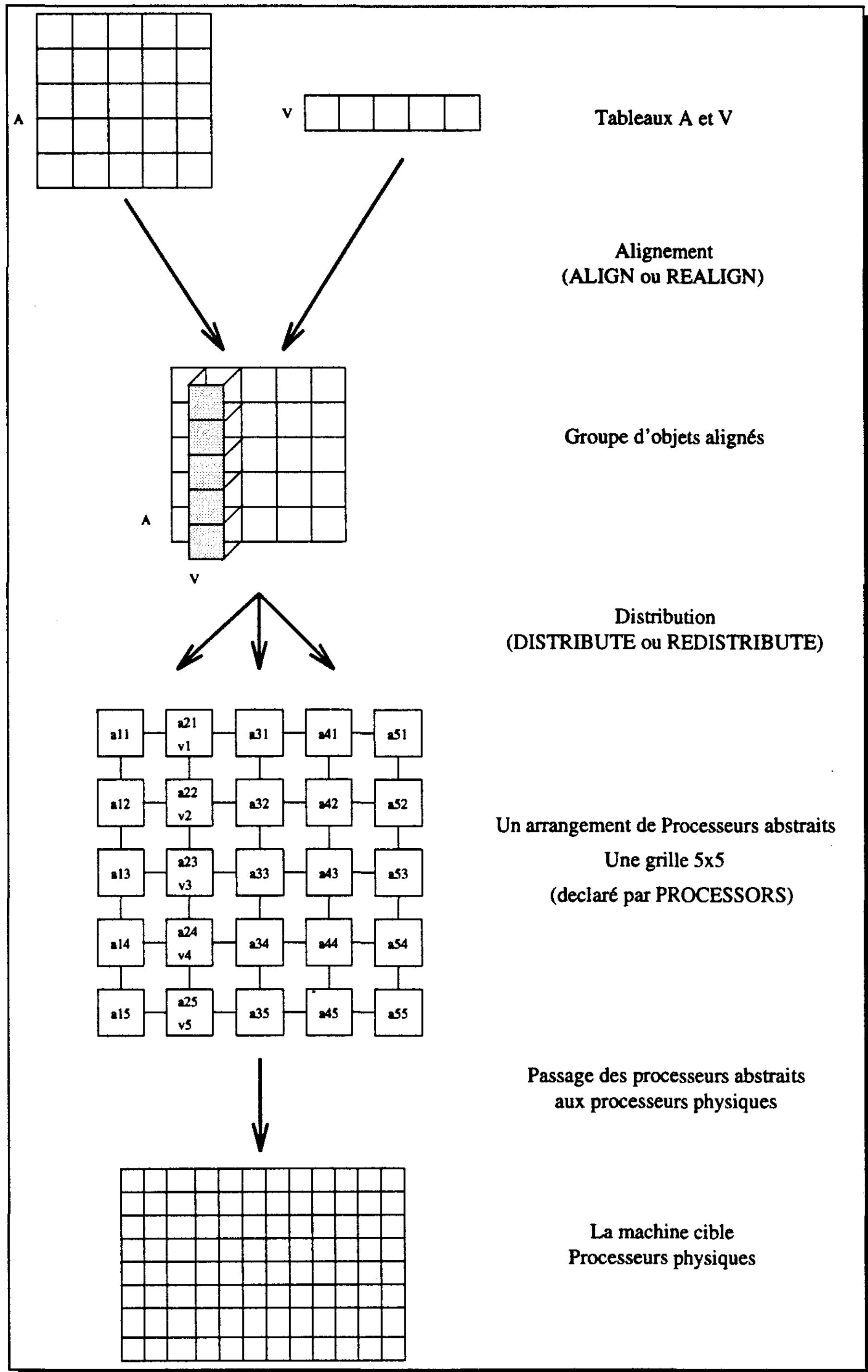


FIG. V.1 - *Les différents niveaux de placement des données en HPF*

Avant de décrire ces directives, nous présentons les tableaux et la directive TEMPLATE. Celle-ci permet de définir des référentiels par rapport auquels peuvent se faire des alignements. Nous renforçons à chaque fois ces descriptions par un minimum d'exemples de syntaxe afin de mieux illustrer leur utilisation. Pour plus de détails sur HPF nous nous reportons à [For93].

1.1 Les tableaux

Les langages Fortran data-parallèles manipulent les tableaux en tant qu'entités du langage. Ils incluent aussi des constructeurs permettant l'expression naturelle d'opérations sur des tableaux entiers ou des sections de tableaux. Ceci inclut aussi bien des assignations telles que « $A = B$ » que des opérations arithmétiques comme « $A + B$ » où A et B sont des tableaux conformes (ici la conformité signifie que les tableaux sont de même rang et de même taille sur chaque dimension).

En plus des tableaux entiers, les opérandes peuvent être des sections de tableaux. Ces dernières peuvent être spécifiées soit par des triplets : « borne inférieure, borne supérieure et pas », par exemple $A(2:20:2)$, soit par une liste d'index, exemple : $A(V)$ où V est une liste d'index (un tableau d'entiers). Lorsque les bornes ne sont pas précisées (exemple : $A(:,:)$, cela signifie que c'est toute la dimension du tableau qui est considérée).

1.2 La directive « TEMPLATE »

Nous avons besoin parfois de considérer un grand espace d'index sur lequel nous alignons plusieurs tableaux (égaux ou plus petits que cet espace), mais sans déclarer un tableau réel qui contienne cet espace d'index. HPF permet la déclaration d'un template qui peut être considéré comme un tableau sans type et dont les éléments n'ont pas de contenu (donc sans allocation mémoire). C'est simplement un espace abstrait de positions ou d'index qui peut être distribué et avec lequel des tableaux peuvent être alignés.

La directive TEMPLATE déclare un ou plusieurs templates. Elle spécifie pour chacun son nom, son rang (nombre de dimensions), et la taille de chaque dimension.

1.2.1 Syntaxe

```
!HPF$ TEMPLATE  nom_de_template [ (dimensions) ]
```

1.2.2 Exemples

```
!HPF$ TEMPLATE  A(N)
!HPF$ TEMPLATE  B(N,N), C(N,2*N)
```

La directive TEMPLATE peut être combinée avec d'autres directives, exemple :

```
!HPF$ TEMPLATE, DIMENSION(100,100) :: F,G
```

Les templates sont utiles, particulièrement lorsque nous voulons aligner plusieurs tableaux les uns par rapport aux autres, mais sans être obligé de déclarer un tableau global qui couvre tout l'espace d'index désiré.

1.3 Les directives d'alignement

La directive ALIGN est utilisée pour spécifier que certains objets de données doivent être placés de la même façon que d'autres objets. Des opérations entre des objets alignés sont plus efficaces que des opérations entre des objets non alignés (ceux alignés sont considérés comme étant placés sur les mêmes processeurs abstraits). La directive ALIGN est conçue afin qu'il soit facile de spécifier, en une seule fois, des placements explicites pour tous les éléments d'un tableau.

Par exemple le code (extrait de [Mar92]) :

```
      DIMENSION  U (5), V (5), A (5, 5)
!HPF$ ALIGN U (I) WITH A (I, 1)
!HPF$ ALIGN V (I) WITH A (2, I)
```

aligne les éléments de U avec ceux de la première ligne de A, et les éléments de V avec ceux de la seconde colonne de A (figure V.2).

Il est possible d'aligner des tableaux de tailles différentes. Le code suivant :

```
      DIMENSION  R (5), Q (20)
!HPF$ ALIGN R (I) WITH Q (I+5)
```

aligne les éléments de R sur ceux de Q tel que représenté à la figure V.3.

La directive REALIGN est similaire à la directive ALIGN à part que la première est exécutable. Un tableau (ou un template) peut être réaligné à n'importe quel moment, il suffit qu'il ait été déclaré dynamique (directive DYNAMIC, *cf.* §1.4).

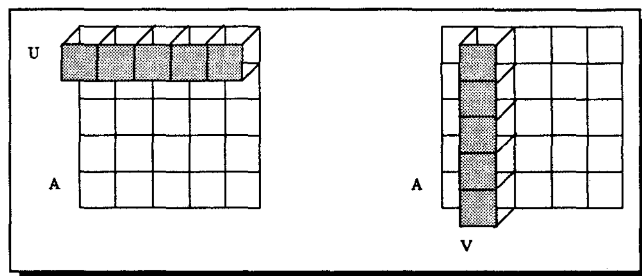


FIG. V.2 - *Alignements de projection*

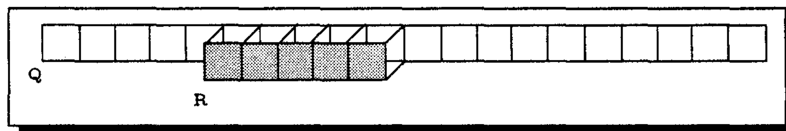


FIG. V.3 - *Alignement de déplacement*

1.3.1 Syntaxe

Sans donner la grammaire complète de ces directives, nous retiendrons la syntaxe simplifiée suivante qui permet d'aligner un ou plusieurs « *alignés* » avec un tableau ou un template :

```
!HPF$ [RE]ALIGN aligné [(list_srce_alignt)] &
!HPF$     WITH tableau_ou_template[(list_index)]
```

ou bien

```
!HPF$ [RE]ALIGN [(list_srce_alignt)] &
!HPF$     WITH tableau_ou_template[(list_index)] :: aligné[,aligné2[,...]]
```

1.3.2 Exemples d'alignement par rapport à des templates

Les tableaux peuvent être également alignés avec des templates. Voici comment on peut par exemple aligner 4 tableaux (N, N) avec les 4 coins d'un template *terre* de taille $(N + 1, N + 1)$:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) :: terre(N+1,N+1)
      real, DIMENSION(N,N) :: NordOuest,NordEst,SudOuest,SudEst
!HPF$ ALIGN NordOuest(I,J)  WITH  terre(I,J)
!HPF$ ALIGN NordEst(I,J)    WITH  terre(I,J+1)
!HPF$ ALIGN SudOuest(I,J)   WITH  terre(I+1,J)
!HPF$ ALIGN SudEst(I,J)     WITH  terre(I+1,J+1)
```

1.4 La directive « DYNAMIC »

La directive DYNAMIC spécifie qu'un objet peut être dynamiquement réaligné ou redistribué. Par exemple :

```
!HPF$ DYNAMIC  A,B,C,D,E
!HPF$ DYNAMIC :: A,B,C,D,E
```

La seconde forme permet de combiner d'autres directives avec la directive DYNAMIC.

1.5 La directive « PROCESSORS »

Afin d'aider l'expression du placement, HPF autorise la déclaration d'entités abstraites sur lesquelles sera effectué le placement des données. La directive PROCESSORS déclare un ou plusieurs arrangements de processeurs. Elle spécifie pour chacun son nom, son rang (nombre de dimensions), et la taille de chaque dimension.

Si deux arrangements de processeurs ont la même forme (shape), les éléments correspondants des deux arrangements sont supposés référencer le même processeur abstrait. De plus, lorsque des directives spécifient collectivement que deux objets sont placés à un instant donné sur le même processeur abstrait, le but est que ces deux objets soient placés sur le même processeur physique.

Les fonctions intrinsèques NUMBER_OF_PROCESSORS et PROCESSORS_SHAPE peuvent être utilisées pour s'informer sur le nombre total des processeurs physiques actuels utilisés dans l'exécution du programme. Cette information peut donc être utilisée afin de calculer les tailles appropriées pour la déclaration des arrangements de processeurs.

1.5.1 Syntaxe

```
!HPF$ PROCESSORS  nom_de_l'arrangement [ (dimensions) ]
```

1.5.2 Exemples

```
!HPF$ PROCESSORS  P(N)
!HPF$ PROCESSORS  Q(NUMBER_OF_PROCESSORS()),&
!HPF$             R(8, NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS  Bizarre(1972:1997, -20:17)
```

La première directive déclare un arrangement linéaire de processeurs P de longueur N. La deuxième déclare deux arrangements Q et R ; Q est linéaire d'une longueur égale au nombre de processeurs physiques de la machine, et R est un arrangement qui configure l'ensemble des processeurs physiques de la machine en une grille de 8 lignes. La troisième directive montre la possibilité de donner des index aux processeurs abstraits.

1.6 Les directives de distribution

Dans cette partie, nous étudions les différentes possibilités de distribuer un tableau ou un template sur l'ensemble des processeurs abstraits de la machine virtuelle. HPF offre au programmeur la possibilité de spécifier le placement qu'il souhaite pour ses données en terme de découpage en couches : par bloc ou cyclique.

Distribution par Bloc

La directive DISTRIBUTE spécifie le placement d'objets de données sur des processeurs abstraits. Par exemple pour une machine hypothétique de 8 processeurs et un tableau de 52 éléments :

```
!HPF$ PROCESSORS Phuit(8)
      REAL A(52)
```

la distribution du tableau par *bloc* (ce qui signifie ici des *blocs* de 7 éléments) :

```
!HPF$ DISTRIBUTE A(BLOCK) ONTO Phuit
```

donne le placement suivant des éléments du tableau sur les processeurs abstraits :

P1	P2	P3	P4	P5	P6	P7	P8
1	8	15	22	29	36	43	50
2	9	16	23	30	37	44	51
3	10	17	24	31	38	45	52
4	11	18	25	32	39	46	
5	12	19	26	33	40	47	
6	13	20	27	34	41	48	
7	14	21	28	35	42	49	

La taille du bloc peut être spécifiée explicitement ; la distribution précédente signifie la même chose que `sc block(7)`. La distribution du même tableau par bloc de 8 éléments :

```
!HPF$ DSTRIBUTE A (BLOCK(8))
```

spécifie que des groupes d'exactement 8 éléments devraient être placés sur des processeurs abstraits successifs. Il doit y avoir au moins $\lceil 52/8 \rceil = 7$ processeurs abstraits pour satisfaire la directive. Le septième processeur contiendra seulement 4 éléments, en l'occurrence A(49:52).

Distribution Cyclique

HPF permet aussi la distribution cyclique. La distribution *cyclique* du tableau A précédent :

```
!HPF$ DISTRIBUTE A(CYCLIC) ONTO Phuit
```

donne le placement suivant des éléments du tableau sur les processeurs abstraits suivant (les éléments successifs du tableau sont distribués d'une manière circulaire (round-robin) sur des processeurs abstrait successifs) :

P1	P2	P3	P4	P5	P6	P7	P8
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52				

Là aussi nous pouvons spécifier la taille du groupe; distribuer le tableau A avec `CYCLIC(3)` :

```
!HPF$ DISTRIBUTE A(CYCLIC(3)) ONTO Phuit
```

donnerait le placement suivant :

P1	P2	P3	P4	P5	P6	P7	P8
1	4	7	10	13	16	19	22
2	5	8	11	14	17	20	23
3	6	9	12	15	18	21	24
25	28	31	34	37	40	43	46
26	29	32	35	38	41	44	47
27	30	33	36	39	42	45	48
49	52						
50							
51							

1.7 L'instruction FORALL

Pour exprimer explicitement le traitement data-parallèle, une nouvelle instruction a été définie dans HPF : l'instruction FORALL [For93] [2]. Cette instruction (appelée aussi instruction d'assignation de tableaux d'*éléments* [ALS91]) est utilisée pour spécifier une affectation d'un tableau en terme d'éléments individuels ou groupes de sections. L'assignation pourrait être masquée par une expression logique scalaire. Le corps de l'instruction FORALL doit être une seule instruction d'assignation dont le membre gauche spécifie un tableau d'éléments ou une section de tableau [3]. La sémantique du FORALL est l'assignation de chacun de ces éléments ou de ces sections (un pour chaque combinaison de valeurs d'index pour laquelle l'expression de masque est vraie) avec tous ceux du membre droit qui sont d'abord évalués (donc avant qu'aucun élément du membre gauche ne soit assigné). L'assignation FORALL est sémantiquement équivalente à une paire de DO_LOOP consécutives ; la première évalue la partie droite dans un temporaire et la deuxième assigne le temporaire au membre gauche. Voici un exemple direct de l'instruction FORALL :

```
FORALL ( i = 1:N, j = 1:M:2 )   A ( i,j ) = i * B ( j )
```

Le FORALL se distingue par rapport aux autres instructions d'assignation par le fait qu'il soit plus suggestif quant aux opérations locales sur chaque élément d'un tableau et aussi par sa généralité qui doit permettre la spécification d'une classe plus large de sections de tableaux [For93].

2. En fait le FORALL a été proposé lors de l'élaboration du standard Fortran 90 sans qu'il soit adopté. Mais il existait dans d'autres langages avant HPF, par exemple MPF de DEC, CMF de TMC, ...

3. Il existe également la construction FORALL qui peut inclure dans son corps plusieurs instructions, mais nous ne la traitons pas ici.

2 Traduction de la sémantique du modèle géométrique HELP en HPF

Nous montrons dans ce qui suit les mécanismes dont se dote HELPDraw pour générer automatiquement à partir des manipulations interactives du programmeur un code HPF. Notre objectif est de trouver les moyens permettant à HELPDraw de générer un code aussi proche que possible d'un code écrit à la main. Avant d'étudier la méthode que nous avons adopté (*cf.* 2.2), nous présentons une autre alternative qui n'est pas réalisable actuellement principalement à cause des restrictions imposées dans le subset HPF [For93].

2.1 La première alternative écartée

L'idée de base concernant la première alternative consiste à préserver la traduction explicite des communications en les distinguant des opérations de calcul. Dans cette solution, nous considérons que l'hyper-espace peut se traduire par un template, puisqu'il n'est rien d'autre qu'un espace d'index virtuel par rapport auquel sont positionnés les DPO. La traduction ensuite des opérations géométriques revient à changer l'alignement des tableaux (représentant les DPO) par rapport au template (représentant l'hyper-espace). Ce pourquoi nous aurions traduit ces opérations par des directives de réalignement. Avec ces directives, nous sommes sensés assurer que lors d'un calcul les éléments ont le même alignement. Il apparaît cependant deux problèmes : d'une part ces directives ne permettent pas d'exprimer toutes les opérations géométriques du modèle, et d'autre part elles ne sont pas encore incluses dans le subset HPF.

Théoriquement nous sommes capables de traduire des opérations géométriques comme le MOVE. Par exemple le déplacement d'un objet A vers la position (x1,y1,z1), se traduit par :

```
!HPF$ REALIGN  A  WITH my_hspace(x1:, y1:, z1:)
```

mais il est impossible de traduire des opérations tel que l'EXPAND sans rajouter un FORALL ou un équivalent.

L'autre problème vient du fait que le subset actuel de HPF n'inclut pas les directives dynamiques (DYNAMIC, REALIGN,...). Et même si elles existent, nous ne sommes pas sûrs que le compilateur puisse détecter l'alignement et éviter donc de générer des communications. En effet, HPF peut être amené à générer des communications pour l'évaluation d'une expression calculatoire. Par exemple le FORALL procède à la redistribution des itérations comme mécanisme de compilation.

Ces raisons nous ont amené à écarter cette solution et d'ignorer par conséquent, dans la

méthode de génération automatique adoptée pour HELPDraw, la séparation communications-calculs du modèle. Cette seconde solution doit se réaliser à travers le passage de la notion de point du modèle géométrique HELP vers la notion d'indice sur laquelle se base HPF. Il faut trouver les moyens permettant de convertir la vision géométrique qu'a le programmeur en une description de tableaux par les indices. L'intérêt ici de HELPDraw est de garder transparent au programmeur cette gestion d'indices.

2.2 Le principe du passage du point à l'indice

Pour décrire le principe de la solution retenue, nous avons préféré procéder sur deux niveaux. Nous expliquerons d'abord comment représenter en HPF un objet temporaire résultant d'une suite d'opérations géométriques. Nous verrons ensuite comment traduire une expression data-parallèle complète, qui manipule plusieurs temporaires.

2.2.1 Représentation d'un objet temporaire

En HELP, l'application d'une suite d'opérations macroscopiques (`suite_macro`) à un DPO source (exemple : `dpo`) produit un autre objet temporaire `Tmp` tel que :

```
Tmp = dpo.suite_macro
```

Dans cette seconde approche où il y a plutôt notion d'indices, on ne parle plus d'objet mais de tableau décrit par des indices. On ne parle pas non plus d'une suite d'opérations géométriques, mais d'une description par une manipulation d'indices. La figure V.4 montre les deux visions : HELP et HPF.

Dans la vision HPF,

- l'objet source `dpo` est représenté par un tableau décrit par des indices : `dpo(i,j,k)` où i, j, et k permettent de parcourir le domaine de définition de `dpo` ($D_{initial}$) ;
- le temporaire final lui aussi est un tableau `Tmp` décrit par d'autres indices, soient I, J, K représentant le domaine de définition de `Tmp` (D_{final}) ;
- la suite d'opérations macroscopiques est elle aussi traduite par une nouvelle description d'indices. N'oublions pas que l'ensemble des éléments de `Tmp` se trouvent dans l'objet source `dpo`, mais dans un ordre différent. Il suffit donc de trouver une fonction f permettant d'associer à chaque élément (I,J,K) de `Tmp` sa source f(I,J,K) dans le DPO initial `dpo` (*cf.* figure V.5). Cette fonction représente en fait l'ensemble des opérations

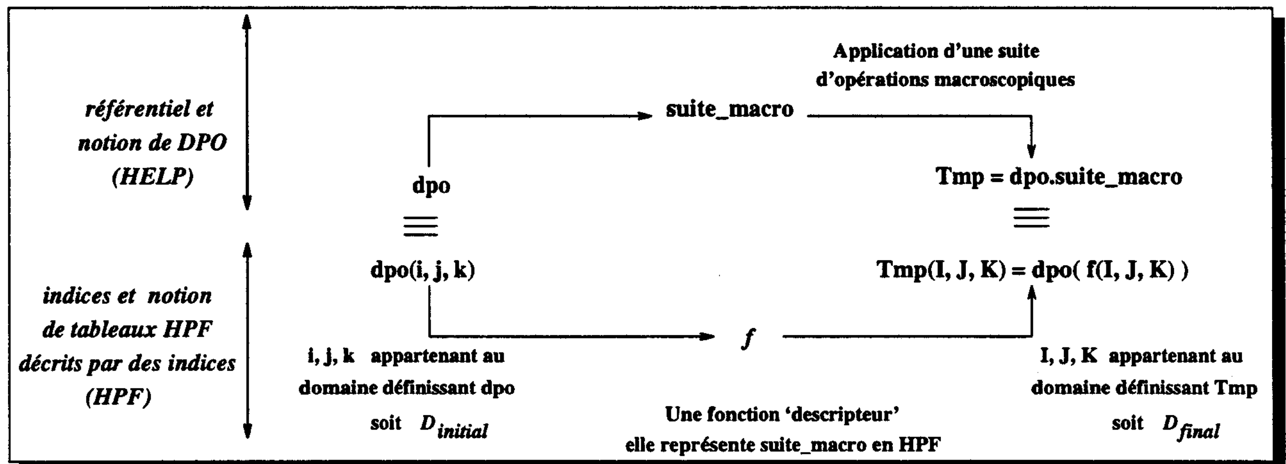


FIG. V.4 - *La nécessité d'une fonction de description f pour représenter une suite d'opérations macroscopiques appliquées à un* DPO.

géométriques appliquées à dpo. Pour être plus précis, cette fonction s'écrit en fait :

$$(f_x(\mathtt{I,J,K}),\ f_y(\mathtt{I,J,K}),\ f_z(\mathtt{I,J,K}))$$

Chaque f_i est une expression arithmétique qui s'écrit en fonction d'un des indices I, J, ou K, par exemple

$$\begin{cases} f_x &= I+1 \\ f_y &= J \\ f_z &= K+2 \end{cases}$$

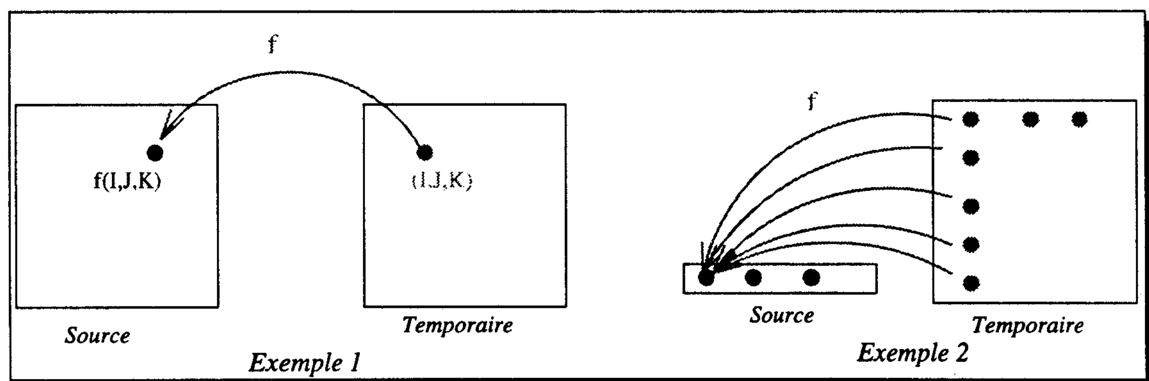


FIG. V.5 - *Exemples de correspondance entre un élément (I,J,K) d'un temporaire obtenu à la suite d'opérations géométriques et sa source f(I,J,K) dans le* DPO *source*

N'oublions pas que ces f_i forment le descripteur qui doit être appliqué au DPO source. Ainsi, lorsque ce dernier est par exemple bi-dimensionnel, seules f_x et f_y sont à déterminer.

2.2.2 Traduction d'une expression data-parallèle

Nous avons vu comment décrire individuellement chaque opérande ou argument d'une expression data-parallèle. Si on prend l'ensemble des opérandes d'une expression data-parallèle, ils s'écrivent tous de la même façon en HPF. Chacun s'écrit :

dpo_source(f_x(I,J,K), f_y(I,J,K), f_z(I,J,K)) $\quad$ I, J, K $\in D_{final}$

la fonction f est à déterminer pour chaque opérande.

Mais on peut constater par ailleurs que tous ces opérandes ont pour un segment conforme le même domaine de définitions : D_{final}. (Évident puisqu'ils sont censés être conformes : même formes.) Seul un cas peut nous faire défaut : un argument gauche d'une affectation peut englober D_{final}. Mais comme nous le verrons plus loin (*cf.* 2.5.2), il est très facile de le ramener au même domaine commun à tous les opérandes.

Maintenant si tous les opérandes ont le même domaine de définition, on peut facilement traduire toute expression data-parallèle par un FORALL. Si le domaine D_{final} est $(Dlen1, Dlen2, Dlen3)$, une expression data-parallèle par exemple :

Res = dpo1.suite_macro1 *op* dpo2.suite_macro2...

se traduira dans un FORALL :

```
FORALL (I=1:Dlen1, J=1:Dlen2, K=1:Dlen3)
      Res(fx, fy, fz) = dpo1(fx', fy', fz') op dpo2(fx'', fy'', fz'') ...
```

Le domaine D_{final} est connu à partir de l'interprétation visuelle. Il nous reste seulement à déterminer la fonction de description f pour chaque opérande.

À noter quelques cas particuliers :

- Lorsqu'un opérande manipulé dans une expression n'est pas un objet temporaire (c'est à dire un DPO variable, exemple : dpo1), la fonction f n'est rien d'autre que la fonction identité. L'opérande est : dpo(I,J,K).
- Le domaine de définition final peut être un triangle. Dans ce cas il faut rajouter un masque à l'entête du FORALL.

Nous décrirons dans ce qui suit comment HELPDraw définit les objets du modèle géométrique (hyper-espaces et DPO), *cf.* 2.3. Nous montrerons ensuite comment il réussit à retrouver la fonction f pour n'importe quelle suite d'opérations géométriques (*cf.* 2.4). Puis, à travers l'étude des opération microscopiques, nous verrons le développement d'une expression data-parallèle complète (*cf.* 2.5).

2.3 Déclarations des éléments du modèle géométrique

Les hyper-espaces auraient pu être traduits par des templates par rapport auquels seront alignés les DPO. Mais ces DPO n'auront qu'un alignement statique (directive `align`) ; tel que nous l'avons précédemment expliqué, ils ne peuvent pas être réalignés. Nous pouvons distinguer de ce fait deux cas : dans le premier nous traduisons les hyper-espaces par des templates, nous aurons un alignement statique des objets, mais qui ne sera pas pris en compte lors du traitement. Dans le deuxième cas nous ne traduirons pas les hyper-espaces (pas de référentiel), et il n'y aura donc pas d'alignements à faire.

Dans ce document nous avons retenu le dernier cas : pas de traduction d'hyper-espace, puisque de toute façon cela n'influe pas sur la traduction des instructions data-parallèles. Il faut savoir par ailleurs que l'adoption du premier cas (templates + alignements), ne pose aucune difficulté.

Dans cette partie, nous étudierons le code HPF généré automatiquement par HELPDraw pour déclarer les objets data-parallèles manipulés par le programmeur.

2.3.1 Les objets rectangulaires

Les DPO sont représentés par des tableaux en HPF. Le code produit automatiquement par HELPDraw, pour un DPO ($len1, len2, len3$) quelque soit son origine dans l'hyper-espace, est le suivant :

```
C DPO float my_hspace[x=orig1;len1, y=orig2;len2, z=orig3;len3] A
      real A(len1, len2, len3)
```

2.3.2 Les objets non rectangulaires

Si la forme géométrique du DPO n'est pas rectangulaire, la déclaration en HPF correspond au plus petit tableau englobant. Pour le triangle de la figure V.6, HELPDraw génère

automatiquement la déclaration :

```
real my_triangle(len1, len1)
```

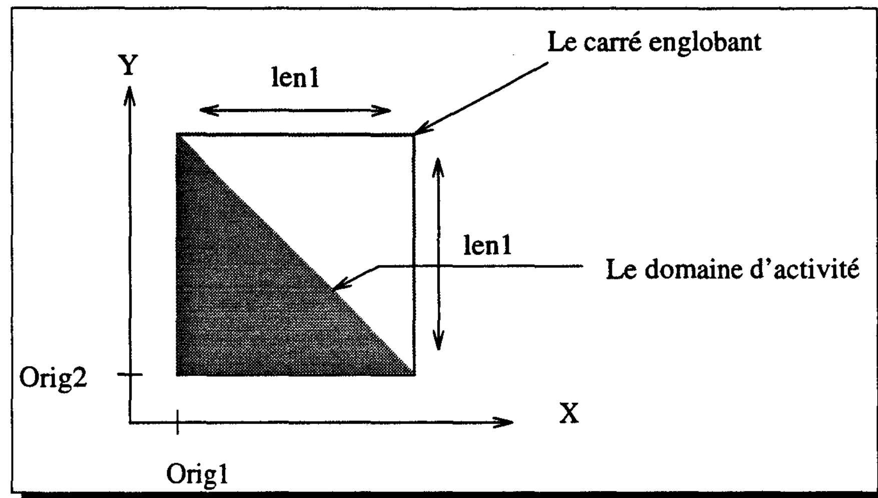


FIG. V.6 - *Un exemple de forme géométrique : le triangle*

Le tableau englobant est complètement transparent à l'utilisateur qui continue à manipuler le triangle en tant que tel. Lorsque des opérations sont appliquées à ce type de formes géométriques, HELPDraw génèrera, toujours de manière transparente, le masque (spécifié par un FORALL) correspondant au domaine d'activité défini par l'objet. Le masque pour le triangle de la figure V.6 est défini par : « $j \leq -i + len1 + 1$ », où i et j sont les indices sur les deux dimensions x et y.

La diagonale ne nécessite aucun masque, elle est traitée comme une simple ligne. HELPDraw la déclare et la gère comme un simple tableau mono-dimensionnel. Par exemple :

```
C      dpo float my_hspace2 [x=20, y=10, d=1;len ] my_diag
       real my_diag (len)
```

2.3.3 Les dpo dynamiques

Les DPO dynamiques dans notre modèle peuvent changer de rang et le programmeur peut changer aussi bien leur nombre de dimensions que la taille de chaque dimension. Un DPO $(2 * N, 3 * N)$ peut devenir un DPO mono-dimensionnel de taille (N). En HPF, nous avons la

possibilité de changer la taille sur chaque dimension, mais pas le nombre de dimensions. Nous utilisons les primitives d'allocation dynamique « **allocate, deallocate** ». Elles permettent d'allouer ou de libérer l'espace réservé d'un tableau. Pour pouvoir les utiliser, il faut que le tableau soit déclaré en précisant le nombre de dimensions (primitive **dimension**), par exemple :

```
      real, allocatable, dimension(:,:) :: A
C     première allocation
      allocate (A(N,1))
      ...
C     deuxième allocation
      deallocate(A, stat=errr)
      allocate(A(N,2))
```

Pour éviter cette contrainte au programmeur, HELPDraw déclare un tableau avec un nombre de dimensions égal à celui de l'hyper-espace, puis n'utilise que les dimensions nécessaires. Si l'hyper-espace est 3-D, HELPDraw génère le code suivant pour déclarer un DPO A de taille initiale (NxN) :

```
C DPO float my_hspace[x=N, y=N, z=1] A
      real, allocatable, dimension(:,:,:) :: A
C     première allocation
      allocate (A(N,N,1))
```

Si ensuite le programmeur change cet objet en un DPO mono-dimensionnel de taille (P), HELPDraw génère le code suivant :

```
C     deuxième allocation
      deallocate(A, stat=errr)
      allocate(A(P,1,1))
```

2.4 Le niveau macroscopique

Dans cette partie nous expliquerons d'abord le principe du calcul de f, en distinguant les différentes étapes de ce calcul (*cf.* 2.4.1). Certaines informations sont nécessaires au calcul de f, nous donnerons dans la sous-section 2.4.2 les outils permettant de les maintenir et de les gérer. La déterminaison de la fonction f est dépendante de la plupart des opérations macroscopiques, c'est pourquoi nous étudierons dans la sous-section 2.4.3 l'influence de chaque opération. Nous donnerons à la fin la forme générale de la fonction (*cf.* 2.4.4) que nous illustrerons par un

exemple détaillé (*cf.* 2.4.5).

2.4.1 Les étapes du calcul de la fonction de description f

Nous devons déterminer pour chaque dimension du domaine de définition du DPO source, la fonction f_i (f_x pour la première dimension, f_y pour la seconde, etc.). Le calcul (ou plutôt la construction d'une fonction f_i) se fait en deux étapes. Dans la première nous déterminons l'expression arithmétique correspondant à f_i : $f_i = expr(Indice_i)$. Cette expression est fonction d'un indice $Indice_i$ jusqu'alors inconnu. Dans la seconde étape, nous déterminons justement quel est cet indice I, J, ou bien K (les indices que nous utilisons dans le FORALL traduisant l'expression data-parallèle). Nous avons préféré laisser la déterminaison de $Indice_i$ jusqu'à la dernière étape du calcul de f_i, parce qu'autrement nous aurions été obligé de le recalculer après chaque opération de rotation, *cf.* explications ci-après.

Étape 1

Cette étape est la plus importante. Notre méthode consiste à suivre une par une l'application des opérations macroscopiques. À chaque opération nous réévaluons la fonction f_i, qui vaut initialement : $f_i = Indice_i$. Il faut savoir que chaque type d'opération est un cas particulier, c'est pourquoi nous devons trouver pour chacun le traitement adéquat. Ces traitements seront décrits dans les sous-sections 2.4.3.1 à 2.4.3.6.

Une opération peut apporter deux types d'informations. Le premier concerne des informations que nous utilisons tout de suite dans la réévaluation de f_i. Le second par contre est nécessaire soit pour les opérations suivantes soit pour la seconde étape (permettant de déterminer l'indice $Indice_i$).

Nous n'abordons pas tout de suite la description des informations qui entrent dans la réévaluation immédiate de f_i (le premier type d'informations). Nous les découvrirons au fur et à mesure dans la suite pour chaque type d'opération macroscopique. Nous donnons par contre les deux informations que nous devons enregistrer pour la suite des opérations (le second type d'informations) :

1. La disposition des indices (i, j, et k) référençant les éléments du DPO source. Cette information est nécessaire pour déterminer $Indice_i$, nous y reviendrons ci-après (*cf.* Étape 2).
2. La dimension (x, y, ou z) sur laquelle s'est faite une réplication, ainsi que la longueur répliquée. Cette information doit être enregistrée dans le cas où il y aurait une opération d'extraction après. Ce cas sera décrit plus en détail avec les extractions (*cf.* 2.4.3.5).

Étape 2

Dans cette étape nous devons déterminer l'indice $Indice_i$ qui entre dans l'expression de la fonction f_i. C'est un traitement très simple. Nous partons du fait que l'objet source est décrit par les indices i, j, et k: `dpo(i,j,k)`. Si on applique une opération de rotation (exemple: `rotate(x,y)`), on obtient un temporaire `Tmp` dont les éléments (j,k,i) sont les mêmes que les éléments (i,j,k) de l'objet source:

```
Tmp(j, i, k) = dpo(i, j, k)
```

Si de nouveau on applique une rotation `rotate(y,z)`, on aura:

```
Tmp(j, k, i) = dpo(i, j, k)
```

Ici les indices i,j,k décrivent le domaine de définition de l'objet source (`dpo`).

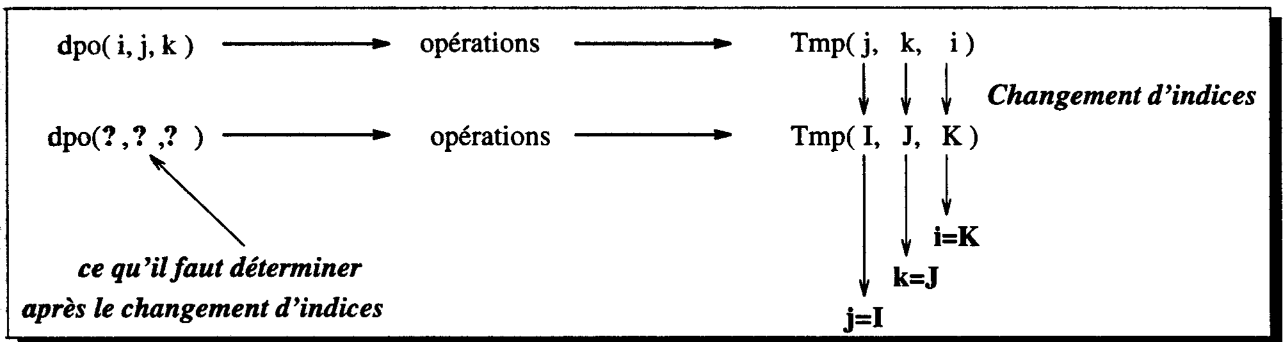


FIG. V.7 - *Changement d'indices*

Cette vision nous permet de savoir où vont se retrouver les éléments `dpo(i,j,k)` à la suite des opération macroscopiques. Or notre objectif est exactement la lecture opposée: nous devons déterminer d'où viennent les éléments (I,J,K) de `Tmp`. C'est un simple *changement d'indices*. Comme nous le montre la figure V.7, en remplaçant j (resp. k et i) par I (resp. J et K), on peut directement retrouver l'ordre des indices décrivant le DPO source:

`dpo(i,j,k)` devient `dpo(K,I,J)`.

Ce résultat nous permet de savoir que $Indice_x = K$, $Indice_y = I$, $Indice_z = J$; d'où le résultat:

$$\begin{cases} f_x = expr(K) & \text{-- } f_x \text{ s'exprime en fonction de l'indice } k \\ f_y = expr(I) \\ f_z = expr(J) \end{cases}$$

Ce changement d'indices ne s'opère qu'à la fin des opérations. C'est pourquoi nous gardons

l'information donnant la disposition des indices $(i, j,$ et $k)$. Ces indices sont maintenus par HELPDraw jusqu'à la dernière opération, mais il faut savoir que seules les opérations de rotation changent cette disposition (*cf.* 2.4.3.2).

2.4.2 Les outils de maintien des informations nécessaires au calcul de f

Les deux informations que nous avons citées : d'une part la disposition des indices et d'autre part la longueur et la dimension de la réplication, sont enregistrées dans une table interne où chaque ligne correspond à une opération. Voici un modèle de ce type de table :

	x	y	z
DPO-source	i	j	k
opération	i	j	k

x	y	z
Slen1	Slen2	Slen3
-	-	-

Chaque dimension (x, y, ou z) de l'hyper-espace est représentée par deux colonnes : une pour l'indice et l'autre pour la longueur de l'objet sur cette dimension. La première ligne de la table correspond au DPO source dont les éléments sont parcourus par les indices i, j et k. Pour toute opération appliquée correspondra ensuite une ligne où on note la disposition des indices (première table) ainsi que les longueurs du nouvel objet temporaire obtenu à la suite de l'opération.

Ces enregistrements sont effectués jusqu'à la fin de la première étape du calcul des f_i. La figure V.8 rappelle les points à suivre.

Pour des raisons de clarté en ce qui concerne le changement d'indices intervenant à la seconde étape du calcul de f, nous montrons à chaque fois ce changement par un passage d'une table d'informations à une autre table représentant les nouveaux indices. Si la dernière ligne de la table est (j, k, i), HELPDraw opère la transformation suivante :

	x	y	z
DPO-source	i	j	k
opération	j	k	i

$$\xrightarrow[\substack{J = k \\ K = i}]{I = j}$$

	x	y	z
DPO-source	K	I	J
opérande	I	J	K

La table à droite nous permet de savoir que l'élément (I, J, K) de l'opérande lui correspond une source qui est de la forme :

$$\begin{cases} f_x &= expr(K) \\ f_y &= expr(I) \\ f_z &= expr(J) \end{cases}$$

À partir de là, on sait que f_x par exemple s'exprime en fonction de l'indice K.

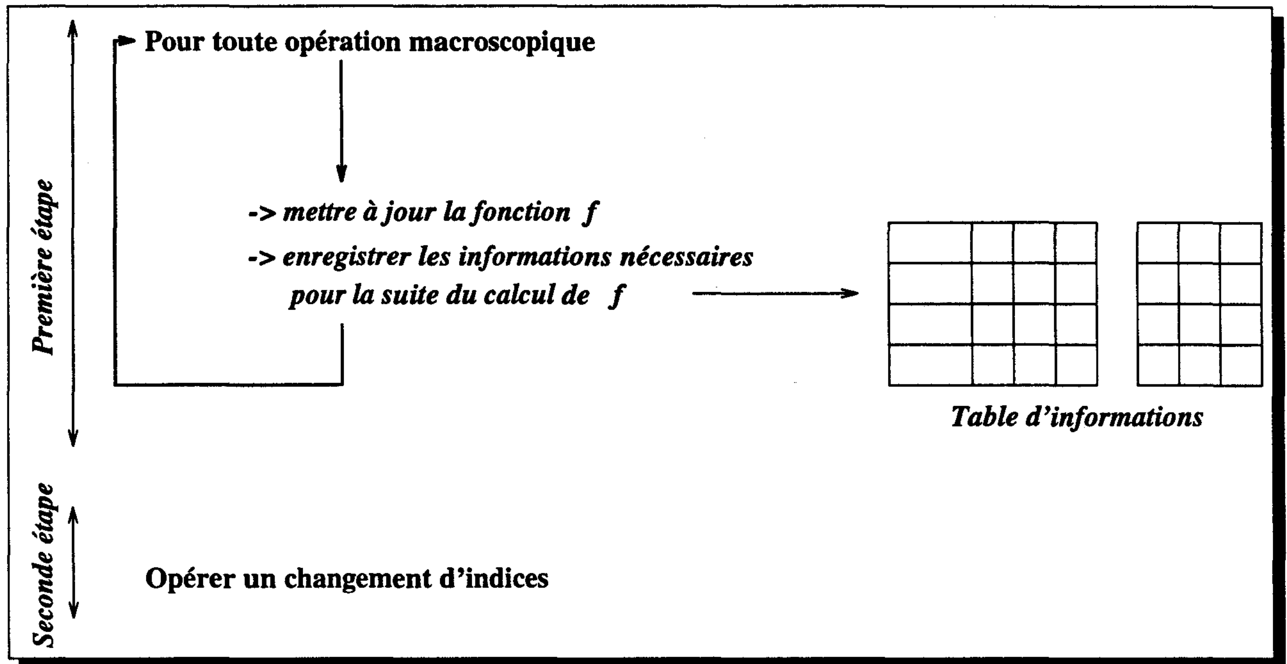


FIG. V.8 - *Les étapes de calcul de f*

Notations

Pour la suite des explications, nous utiliserons les notations suivantes. Pour une table d'informations :

	x	y	z
DPO-source	i	j	k
opération1	indice(x)	indice(y)	indice(z)
opération2	indice(x)	indice(y)	indice(z)

x	y	z
$Nlen_x$	$Nlen_y$	$Nlen_z$
$Nlen_x$	$Nlen_y$	$Nlen_z$

$indice(col)$ est l'indice qui se trouve sur la colonne x, y, ou z. Il modifie l'élément $f_{indice(col)}$, où $f_{indice(col)} est$:

$$\begin{cases} f_x & si\ indice(col) = i \\ f_y & si\ indice(col) = j \\ f_z & si\ indice(col) = k \end{cases}$$

2.4.3 Calcul de f pour chaque opération macroscopique

Nous étudions dans cette partie l'influence de chaque opération macroscopique sur le calcul de f. Nous verrons en particulier :

- les informations à enregistrer à la suite d'une opération (ces informations peuvent être nécessaires dans les opérations suivantes) ;
- réévaluation de f : on recalcule f pour prendre en compte l'opération appliquée.

Pour plus d'illustration, on regardera pour certaines opérations ce que donnerait le changement d'indices, si l'opération macroscopique effectuée était seule : (`dpo.macro`).

2.4.3.1 Déplacements

Informations à enregistrer

L'opération de déplacement ne nécessite aucune nouvelle information à enregistrer. Seule l'origine absolue du temporaire change à la suite d'une telle opération. f est ici la fonction identité :

	x	y	z
DPO-source	i	j	k
move(x1,y1,z1)	i	j	k

x	y	z
Slen1	Slen2	Slen3
Slen1	Slen2	Slen3

La longueur $Slen_i$ représente la longueur du DPO source sur la dimension i.

Réévaluation de f

Aucune réévaluation n'est à faire : $f_i = f_i$.

Exemple de changement d'indices

Si la dernière ligne de la table représente la dernière opération géométrique à effectuer, HELPDraw après changement d'indices obtiendra la table suivante :

	x	y	z
DPO-source	i	j	k
move(x1,y1,z1)	i	j	k

$$\xrightarrow{I=i} \quad J=j \quad K=k$$

	x	y	z
DPO-source	I	J	K
opérande	I	J	K

Les expressions décrivant l'opérande restent les mêmes. Dans cet exemple, où il y a seulement une opération de déplacement, elles sont :

$$\begin{cases} f_x &= I \\ f_y &= J \\ f_z &= K \end{cases}$$

Exemple de code généré Le code généré automatiquement pour l'expression data-parallèle « `Res = dpo1 + dpo2.move(100,100,1)` », est :

```
FORALL(I=1:Dlen1, J=1:Dlen2, K=1:Dlen2)
   Res(I,J,K) = dpo1(I,J,K) + dpo2(I,J,K)
```

2.4.3.2 Rotations

Informations à enregistrer

Nous traitons les deux opérations **ROTATE** et **EXCHANGE** de la même façon, puisque la seule différence est l'origine absolue du temporaire. Or cette origine, comme dans le cas de l'opération de déplacement, n'influe pas sur le calcul de f. Ainsi, lorsqu'une opération de rotation est appliquée, HELPDraw notera seulement la nouvelle disposition des indices. Il interchangera par rapport à la dernière ligne de la table, aussi bien les indices que les longueurs correspondant aux deux dimensions de la rotation.

Voici un exemple de table mise à jour à la suite d'une opération de rotation

« `B.rotate(x,z)` » appliquée au DPO tri-dimensionnel B :

	x	y	z
DPO-source	i	j	k
rotate(x,z)	k	j	i

x	y	z
Slen1	Slen2	Slen3
Slen3	Slen2	Slen1

Réévaluation de f

Aucune réévaluation n'est nécessaire pour des opérations de rotation : $f_i \vdash f_i$.

Exemple de changement d'indices

Après le changement d'indices, HELPDraw déduira pour chaque élément composant le triplet en fonction de quel indice il s'exprime. Dans cet exemple, le changement d'indices donne :

	x	y	z
DPO-source	i	j	k
rotate(x,z)	k	j	i

$$\xrightarrow[\substack{J = j \\ K = i}]{I = k}$$

	x	y	z
DPO-source	K	J	I
opérande	I	J	K

Dans ce cas, les éléments de f s'expriment de la façon suivante :

$$\begin{cases} f_x &= expr(K) \\ f_y &= expr(J) \\ f_z &= expr(I) \end{cases}$$

f_x par exemple s'exprime en fonction de l'indice k. En particulier pour l'opérande « `B.rotate(x,z)` », HELPDraw génère : `B(K,J,I)`.

Exemple de code généré Pour une expression :

```
Res = dpo1 + B.rotate(x,z)
```

HELPDraw génère automatiquement le code suivant :

```
FORALL(I=1:Dlen1, J=1:Dlen2, K=1:Dlen2)
  Res(I,J,K) = dpo1(I,J,K) + B(K,J,I)
```

Triangles et diagonales

La rotation d'une diagonale ou d'un triangle est exprimée de la même façon, puisque ils sont traités respectivement comme un tableau mono-dimensionnel ou comme un carré. En ce qui concerne le triangle, l'expression du masque fait partie du domaine d'activité qui est connu au niveau de l'interprétation visuelle. L'opération de rotation n'a donc pas besoin de calculer le masque. Pour l'expression :

```
Res = t1 + t2.rotate_triangle(x,y)
```

HELPDraw génère le code suivant (t2 est représenté dans la figure V.9) :

```
FORALL(I=1:Dlen1, J=1:Dlen1, J.LE.I)
   Res(I,J) = t1(I,J) + t2(J,I)
```

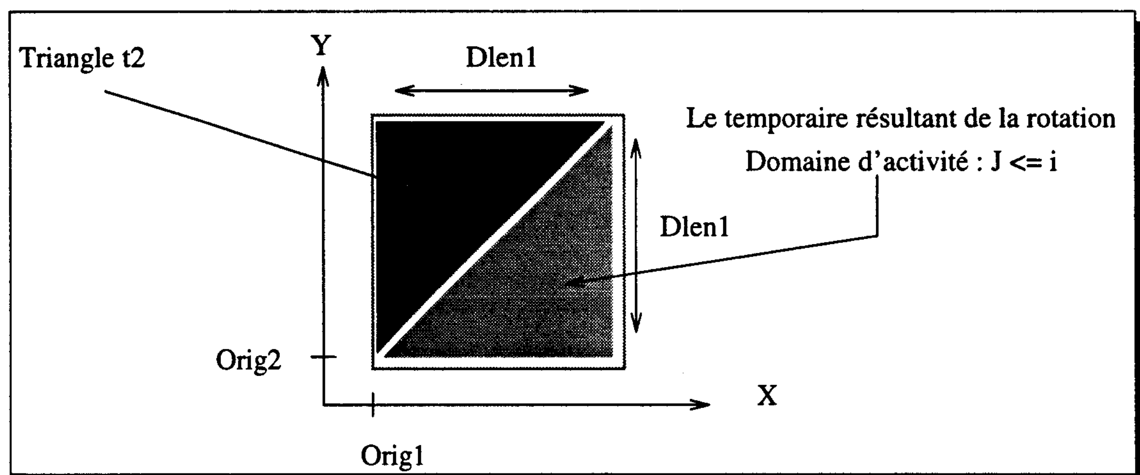


FIG. V.9 - *Un exemple de rotation d'un triangle*

Pour l'expression manipulant une diagonale :

```
Res = Vect + my_diag.rotate_toline(x)
```

HELPDraw génère le code suivant :

```
FORALL(I=1:Dlen1)
   Res(I) = Vect(I) + my_diag(I)
```

La diagonale est traitée de la même façon avant et après la rotation.

2.4.3.3 Réplications

Informations à enregistrer

Nous étudions ici le cas général de réplication : **expand(dim,d)**. L'information que doit enregistrer HELPDraw consiste simplement à marquer la longueur de l'objet (len1, len2, ou len3) sur laquelle s'est effectuée la réplication. Nous verrons plus tard que ces informations sont essentielles pour les opérations d'extraction. Si l'expansion se fait sur l'axe « x » par exemple, la table sera du type :

	x	y	z
DPO-source	i	j	k
expand(x,d)	i	j	k

x	y	z
Slen1	Slen2	Slen3
Slen1*	Slen2	Slen3

En clair, nous gardons les mêmes informations que la ligne précédente, nous rajoutons seulement « * » pour se souvenir qu'il y a eu réplication sur $len_x = Slen1$ ($Slen_i$ est la i^e longueur du DPO source). Cette information nous permet de dire que l'élément (i,j,k) du DPO source correspond aux éléments (i+α*Slen1,j,k) du temporaire résultant de l'opération, où α varie de 0 à ⌈Dlen1/Slen1⌉-1 (⌈Dlen1/Slen1⌉ représente le nombre de copies) :

$$\text{dpo}(i,j,k) \longrightarrow \text{se retrouve à} \longrightarrow \texttt{Tmp}(i+\alpha * Slen1, j, k) \qquad i,j,k \in D_{dpo}$$

La figure V.10 illustre cette correspondance.

Lorsque cette information est interprétée dans l'autre sens, puisque c'est ce qui nous intéresse, elle devient : « la source de l'élément (I,J,K) de l'opérande est l'élément (Mod(I-1,Slen1)+1,J,K) du DPO source[4] » :

$$\texttt{Tmp}(I,J,K) \longrightarrow \text{vient de} \longrightarrow \text{dpo}(Mod(I-1, Slen1)+1, J, K) \qquad I,J,K \in D_{Tmp}$$

Réévaluation de f

Les opérations de réplication modifient la fonction de description f. Avant l'expansion nous écrivions $f_{indice(col)} = Indice_{col}$, or maintenant nous devons écrire, dans le cas d'une

4. Mod(A,B) = A modulo B.

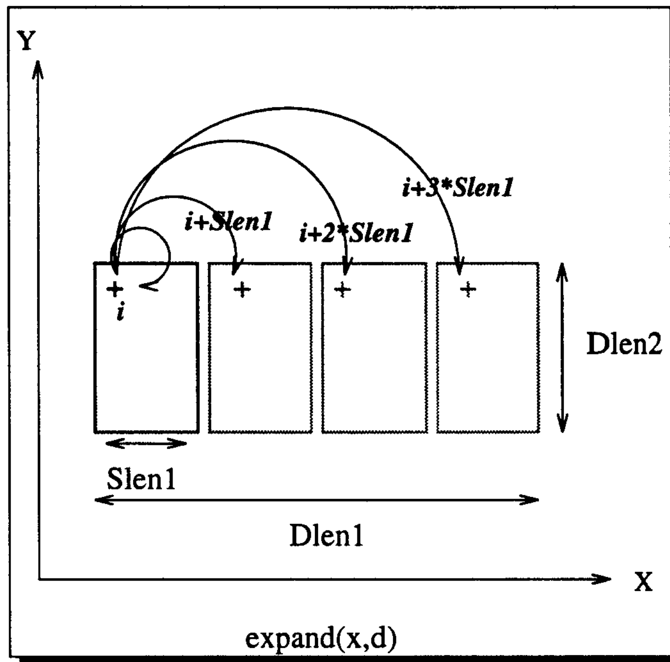


FIG. V.10 - *Un exemple de réplication : l'élément (i, j) du* DPO *source se retrouve aux éléments $(i + \alpha * Slen1, j)$ de l'objet résultat*

expansion sur une dimension *col* :

$$f_{indice(col)} = mod(Indice_{col} - 1, Nlen_{col}) + 1$$

Exemple de changement d'indices

Si nous opérons le changement d'indices pour l'exemple de la figure V.10 « DPO-source.expand(x,d) », nous obtiendrons la table :

	x	y	z
DPO-source	i	j	k
expand(x,d)	i	j	k

$$\xrightarrow{I = i} \quad J = j \quad K = k$$

	x	y	z
DPO-source	I	J	K
opérande	I	J	K

Une fois l'ordre des indices (I, J, K) est retrouvé, l'expression des éléments décrivant l'opérande devient :

$$\begin{cases} f_x = Mod(I - 1, Slen1) + 1 \\ f_y = J \\ f_z = K \end{cases}$$

Exemple de code généré Lorsque cette opérande est utilisée dans une expression data-parallèle, HELPDraw génère un code du type :

```
FORALL(I=1:Dlen1, J=1:Dlen2, K=1:Dlen3)
   Res(I,J) = dpol(I,J) + C(Mod(I-1,Slen1)+1, J, K)
```

2.4.3.4 Décalages

Informations à enregistrer

Les opérations de décalages ne nécessitent aucune nouvelle information à enregistrer. Seule la référence aux éléments sources doit être recalculée pour prendre en compte le décalage.

Réévaluation de f

En ce qui concerne l'opération de décalage circulaire « `circular_shift` », tout élément i de l'objet source va se retrouver à la position $mod(i + d - 1, Slen) + 1$. Lorsque cette information est interprétée dans l'autre sens, elle devient : « la source de l'élément `I` de l'opérande résultat est l'élément $mod(I - d - 1, Slen) + 1$ de l'objet source», *cf.* figure V.11. La fonction f s'écrit donc, dans le cas d'un décalage circulaire de longueur `d` :

$$f_{indice(col)} = \mathrm{mod}(\mathrm{Indice}_{col} - \mathrm{d} - 1, \mathrm{Nlen}_{col}) + 1$$

De même pour l'opération miroir « `mirror` » qui permet d'interchanger des éléments opposés, les éléments `I` de l'opérande résultat ont comme source les éléments `Slen+1-I`, *cf.* figure V.11. D'où la fonction de description f s'écrit, à la suite d'une opération miroir :

$$f_{indice(col)} = \mathrm{Nlen}_{col} + 1 - \mathrm{I}$$

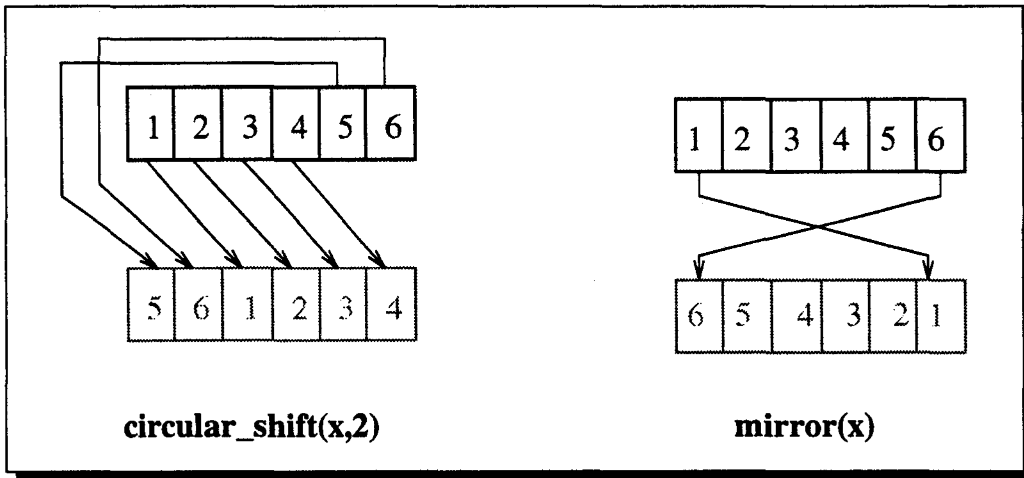


FIG. V.11 - *Exemples d'opérations de décalages*

2.4.3.5 Extractions

Informations à enregistrer

En ce qui concerne l'EXTRACT, la disposition des indices ne change pas ; HELPDraw génère une ligne dont le contenu concernant les indices est le même que celui de la ligne précédente. Il doit cependant introduire les nouvelles longueurs correspondant à la partie extraite. Ceci afin de pouvoir retrouver à chaque fois, la nouvelle zone des éléments sources (c'est à dire le nouveau domaine de définition du temporaire résultat) :

	x	y	z
DPO-source	i	j	k
extract(Norig,Nlen)	i	j	k

x	y	z
Slen1	Slen2	Slen3
Nlen1	Nlen2	Nlen3

En plus des informations à enregistrer, cette opération nécessite la réévaluation de la fonction f.

Réévaluation de f

A priori tous les éléments de l'objet source sont référencés. Mais lorsqu'une opération d'extraction est appliquée (exemple : `extract_subdpo(Norig,Nlen)`), seule une partie de ces éléments reste concernée. La fonction de description doit donc tenir compte de ces nouvelles données (Norig et Nlen[5]) pour ne faire référence qu'aux éléments sources représentant la

5. $Norig$ (resp. $Nlen$) représente l'ensemble des coordonnées $(Norig_1, Norig_2, Norig_3)$ (resp. $(Nlen_1, Nlen_2, Nlen_3)$)

partie extraite.

Si avant l'opération d'extraction, la fonction f décrivait la dimension totale d'un DPO source, par exemple :

$$f_{indice(col)} = \mathrm{Indice}_{col}$$

il suffirait alors d'exprimer le décalage par rapport à l'origine du DPO source. $f_{indice(col)}$ devient :

$$f_{indice(col)} = \mathrm{Norig}_{col} - 1 + \mathrm{Indice}_{col}$$

Or, comme nous l'avons déjà décrit dans l'opération précédente, s'il y a eu une expansion avant l'`extract`, nous n'aurions pas écrit $f_{indice(col)} = \mathrm{Indice}_{col}$, mais[6] :

$$f_{indice(col)} = (\mathrm{Indice}_{col} - 1)[\mathrm{len}_{col}] + 1$$

Ceci afin que la référence aux éléments sources reste toujours entre le début de l'objet et *len*. Nous continuons le même raisonnement lorsque nous avons après une expansion, un `extract`. La fonction $f_{indice(col)}$ devient dans ce cas :

$$f_{indice(col)} = (\ (\mathrm{Norig}_{col} - 1)[\mathrm{len}_{col}] + (\mathrm{Indice}_{col} - 1)\ )[\mathrm{len}_{col}] + 1$$

Pour savoir s'il y a eu expansion avant, HELPDraw regarde dans la table d'informations s'il y a un astérisque « * » devant len_{col} de la ligne précédente.

Lorsque le programmeur applique une autre opération géométrique le calcul se fera par rapport à la nouvelle zone précisée dans la table d'informations. S'il applique par exemple une autre opération d'extraction (`extract_subdpo(Norig',Nlen')`), $f_{indice(col)}$ deviendra :

$$f_{indice(col)} = (\ (\mathrm{Norig}_{col} - 1)[\mathrm{len}_{col}] + (\mathrm{Norig}'_{col}) + (\mathrm{Indice}_{col} - 1)\ )[\mathrm{len}_{col}] + 1$$

Extraction de triangles

L'extraction d'un triangle revient à extraire le carré englobant et à mémoriser le nouveau domaine d'activité définissant le triangle. Or, comme nous venons de décrire l'extraction d'un carré, nous n'insisterons pas plus sur l'extraction de triangles.

6. Nous utilisons parfois, pour simplifier l'expression, la notation A[B] au lieu de mod(A,B).

Extraction de diagonales

Rappelons que la diagonale est représentée par un tableau mono-dimensionnel. Son extraction se déroule de la même manière que l'extraction d'un autre objet mono-dimensionnel régulier, notamment en ce qui concerne les informations à enregistrer et les formules à appliquer. La seule différence réside dans le calcul final des fonction f_i. Pour un objet régulier, nous aurions obtenu à la fin des $f_{indice(col)}$ exprimées en fonction de $Indice_{col}$. Mais en ce qui concerne la diagonale, dès qu'il y a une extraction, les indices $Indice_{col1}$ et $Indice_{col2}$ représentant le plan `[col1,col2]` où est définie la diagonale, deviennent un seul :

$$Indice_{col1} = Indice_{col2} = Indice_{col}$$

$Indice_{col}$ est déterminé par la convention suivante : pour un plan [xy] (respectivement [yz] et [xz]), la diagonale est considérée allouée parallèlement à l'axe « x » (respectivement « y » et « z »). Si la diagonale est allouée sur le plan `[xy]`, $Indice_{col}$ sera $Indice_x$.

Exemple Pour l'extraction d'une diagonale de longueur $Nlen$ à partir de l'origine $Norig$ d'une matrice Mat : `Mat.extract_diag([xy],Norig,Nlen)`, où $Norig = (2,1)$, l'évaluation de f_i se fait comme suit :

	x	y
Mat	i	j
extract_diag([xy],Norig,Nlen)	i	j

x	y
len1	len2
Nlen	1

Comme il y a eu extraction de diagonale, on sait tout de suite que les indices $Indice_x$ et $Indice_y$ vont être remplacés à la fin par $Indice_x$. Avant ce remplacement les f_i sont :

$$\begin{cases} f_x &= Norig_x - 1 + Indice_x \\ f_y &= Norig_y - 1 + Indice_y \end{cases}$$

L'application numérique donne ($Norig_x = 2,\ Norig_y = 1$) :

$$\begin{cases} f_x &= 1 + Indice_x \\ f_y &= Indice_y \end{cases}$$

Ensuite, le changement d'indices nous donne $Indice_x = I$ et $Indice_y = J$. Or, comme il y a eu extraction de diagonale, les deux indices I et J sont remplacés par l'indice I. Les f_i s'écriront enfin :

$$\begin{cases} f_x &= 1 + I \\ f_y &= I \end{cases}$$

Si cet opérande apparaît par exemple dans une expression :

$$\text{ResDiag} = \text{diag1} + \text{Mat.extract_diag([xy], Norig, Nlen1)}$$

HELPDraw génère ($Norig = (2,1)$) :

```
FORALL(I=1:Nlen1)
   Res(I) = Vect(I) + Mat(1+I,I)
```

2.4.3.6 Réductions

Les opérations de réduction ne peuvent pas être traitées comme les autres opérations macroscopiques. À la différence de celles-ci, les opérations de réduction nécessitent à la fois des communications et des calculs, ce qui est incompatible avec la fonction de description f. En effet, cette dernière permet de retrouver une référence directe des éléments sources qui existent effectivement dans l'objet source. À la suite d'une réduction, un élément de l'objet résultat ne lui correspond plus un élément source, mais le résultat d'un traitement sur plusieurs éléments de l'objet source.

Ainsi, dès la rencontre d'une opération de réduction, par exemple :

`S.suiteOpGéom1.réduction.suiteOpGéom2`

HELPDraw arrête le calcul de la fonction de description f, il crée un temporaire « *Tmp* », puis il génère le code du FORALL correspondant à « `Tmp = S.suiteOpGéom1` » :

```
Forall (I=1:TmpLen1, J=1:TmpLen2, K=1:TmpLen3)
   Tmp(I,J,K)  =  S(fx, fy, fz)
```

Il génère ensuite le code de la réduction : `Tmp.reduce`, en utilisant les fonctions intrinsèques de réduction de HPF : SUM, PRODUCT, IANY, IALL, MAXVAL, MINVAL. Pour une réduction somme d'un DPO 3-D selon la dimension « x » (`reduceadd(x)`), HELPDraw génère le code :

```
Tmp(1, :, : )  =  sum(Tmp, dim=1)
```

En réalité avant de générer le code correspondant à la réduction, HELPDraw attend de voir si d'autres opérations de réduction vont être appliquées successivement à la première. Dans ce cas il génère un seul code pour l'ensemble des réductions. Par exemple pour les deux opérations de réduction successives :

```
Tmp.reduceadd(x).reducemul(y)
```

HELPDraw génère :

```
Tmp(1, 1, : )  =  product( sum(Tmp, dim=1), dim=2 )
```

Enfin HELPDraw reprend de nouveau l'évaluation de la fonction correspondant à la suite d'opérations géométriques restantes : `suiteOpGéom2`, en considérant comme objet source le temporaire résultant des réductions.

Dans cette section, nous avons seulement expliqué le principe d'évaluation de la fonction f pour chaque opération macroscopique prise à part. La section suivante traite le cas général où il peut y avoir toutes les combinaisons d'opérations géométriques dans un même opérande. La fonction f se construira par l'application d'un ensemble de règles de réécritures.

2.4.4 Formule générale de la fonction de description f

Étant donné que le programmeur peut appliquer n'importe quelle suite d'opérations géométriques, nous donnons ici l'expression générale de la fonction f permettant de décrire un opérande. Cette formule doit exprimer toutes les opérations géométriques.

À la fin, cet opérande s'écrira :

$$\texttt{opérande}(f_x, f_y, f_z)$$

Nous réalisons le calcul des éléments de l'opérande en deux étapes. La première consiste en l'application d'un ensemble de règles de réécritures permettant de retrouver l'expression de chaque f_i en fonction d'un indice (Indice_i) :

$$\begin{cases} f_x &= expr(Indice_x) \\ f_y &= expr(Indice_y) \\ f_z &= expr(Indice_z) \end{cases}$$

La seconde étape consiste à remplacer chaque indice Indice_i par sa vraie valeur I, J,

ou K. Ces valeurs seront retrouvées à la fin des opérations géométriques en opérant une transformation sur la table d'informations (le changement d'indices).

Étape 1

L'expression de chaque f_i est retrouvée en appliquant pas à pas les règles de réécritures énoncées ci-dessous (*cf.* table V.1). Ces règles sont déclenchées au fur et à mesure que le programmeur développe ses opérations. Elles mettent à jour à la suite de chaque opération, l'expression f_i. Elles se basent essentiellement sur la table d'informations.

TAB. V.1 - *Les règles de réécriture contextuelles pour :* $f_{indice(col)}$

- Le descripteur f_i est un mot δ qu'on va construire progressivement en appliquant pour chaque opération macroscopique effectuée, la règle R_i ($i = 1$ à 5) qui lui correspond.
- Le descripteur de tout objet est initialisé à : $\delta = S + 1$
- $\forall\, \delta,\ \exists\, \alpha,\ \beta \mid \delta = \alpha S \beta$ $\quad \alpha\ \beta$ uniques
  À part la règle R_5, toutes les autres substituent S de $\alpha S \beta$ par un nouveau mot :
  $\alpha S \beta \vdash \alpha$ *nouveau_mot* β

Règle		Substitution	Condition
R_1 [*expand(col)*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ \mathbf{mod}(S,\ \mathbf{Nlen}_{col})\ \beta$	
R_2 [*circular_shift(col,d)*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ \mathbf{mod}(S\mathbf{-d},\ \mathbf{Nlen}_{col})\ \beta$	
R_3 [*mirror(col)*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ \mathbf{Nlen}_{col}\mathbf{+1} - S\ \beta$	
R_4 [*extract*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ \mathbf{mod}(\mathbf{Norig}_{col}\mathbf{-1},\ \mathbf{Nlen}_{col})^{a} + S\ \ \beta$	**Si** $Nlen_{col}$*
		$\vdash \alpha\ \mathbf{Norig}_{col} - 1 + S\ \beta$	**Si** $Nlen_{col}$
R_5 [*autre-macro*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ S\ \beta$	
R_6 [*fin*]	: $\delta = \alpha S \beta$	$\vdash \alpha\ \mathbf{Indice}_{col}\ \mathbf{-\ 1}\ \beta$	

[a] Nlen_{col} de la ligne précédente dans la table d'informations.

On considère que chaque f_i est un mot δ initialisé à « $\delta = S + 1$ » (S non terminal). δ peut être décomposé à n'importe quel moment en $\delta = \alpha S \beta$ (α, β sont des mots). À la suite de chaque opération appliquée par le programmeur, HELPDraw d'abord met à jour la table d'informations comme nous l'avons décrit précédemment. Il applique ensuite la règle (R_i) correspondant à l'opération pour substituer « S » par un nouveau mot représentant l'opération. Il est important de remarquer que la règle R_4 substitue « S » de deux façons différentes selon qu'il y a eu une réplication avant ou non. Ce processus est repris jusqu'à la fin des opérations qui se termine par l'application de la dernière règle (R_6). L'expression résultante est écrite en fonction de Indice_{col} qui sera déterminé dans l'étape suivante.

À noter que toutes les dimensions de l'hyper-espace doivent être représentées dans la

table d'informations. On ne calcule par contre que $f_{indice(col)}$ où *col* entre dans la définition de l'objet source. Si un DPO mono-dimensionnel est alloué parallèlement à l'axe « y », seule f_y sera calculée.

Étape 2

À la fin des opérations, nous opérons une transformation sur la table interne des informations. Nous obtiendrons une table de la forme :

	x	y	z
DPO-source	$Indice_x$	$Indice_y$	$Indice_z$
opérande	I	J	K

Cette table nous permet de connaître l'ordre des indices (I, J, et K) exprimant les éléments de l'opérande. Selon la transformation opérée, $Indice_i$ peut être : I, J, ou K.

Nous avons montré la génération de code automatique correspondant au cas général. La méthode que nous avons adoptée permet à HELPDraw de générer un seul FORALL pour toute l'expression data-parallèle. L'exemple suivant illustre mieux le maintien des informations puis le calcul et l'application de la fonction f afin de générer le code correspondant à une expression data-parallèle.

2.4.5 Exemple de génération automatique d'une expression data-parallèle

Nous proposons d'étudier la génération automatique du code correspondant à l'expression data-parallèle suivante :

```
Res = E + A.exchange(x,y).extract(Norig,Nlen).expand(x).
          extract(Norig',Nlen').exchange(x,y).expand(x)
```

Notre objectif est de montrer comment se construit le code correspondant à l'opérande :

```
A.exchange(x,y).extract(Norig,Nlen).expand(x).
 extract(Norig',Nlen').exchange(x,y).expand(x)
```

Le code complet de l'expression sera montré à la fin de cette section. La figure V.12 montre les différentes opérations géométriques appliquées au DPO A.

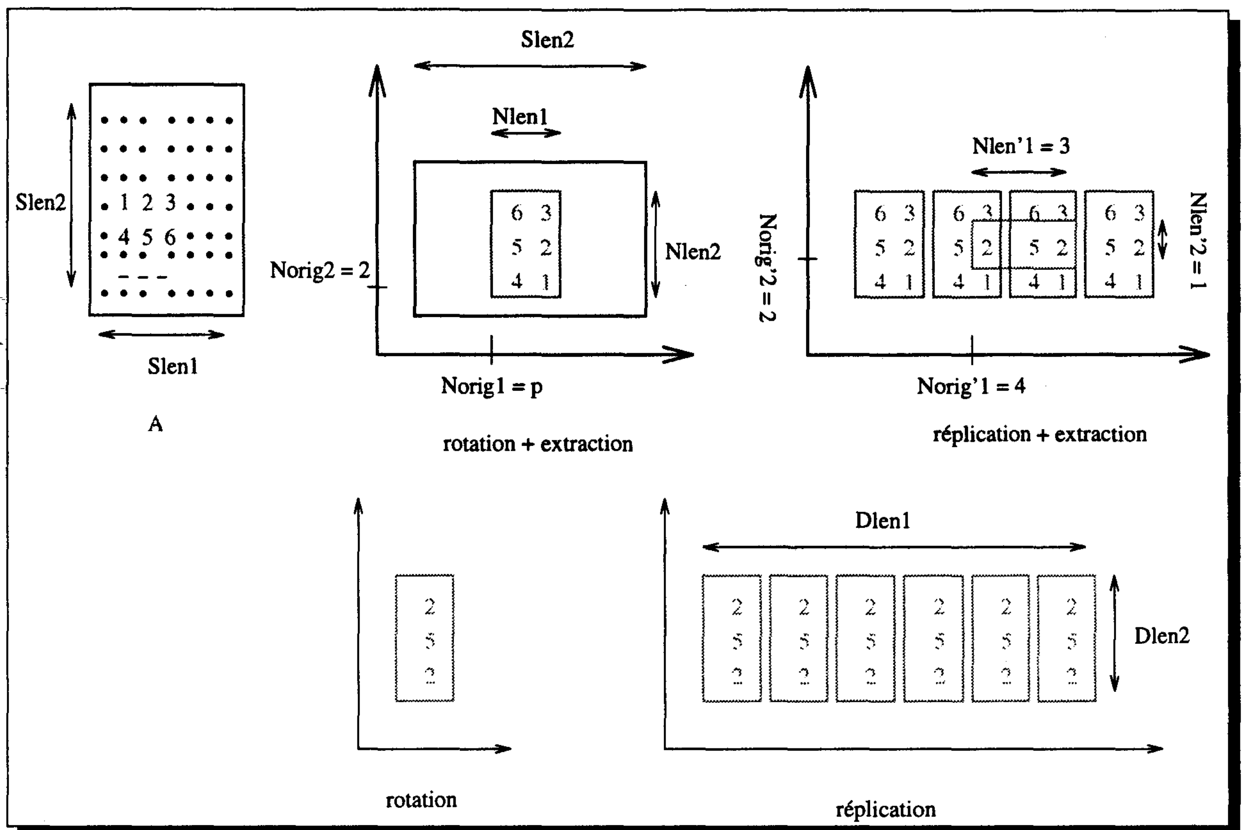


FIG. V.12 - *Un exemple d'opérande*

Nous allons construire, au fur et à mesure de l'application des opérations géométriques, les descripteurs f_x et f_y décrivant l'opérande $A(f_x, f_y)$. Jusqu'à la fin des opérations, nous n'aurons que la forme de ces descripteurs : il nous manquera l'indice I ou J qui doit être utilisé dans un tel ou tel descripteur. Ainsi nous notons dans notre construction $Indice_x$ et $Indice_y$ pour représenter respectivement les indices que nous allons retrouver à la fin pour : f_x et f_y.

Initialisation

Nous commençons d'abord par une table initialisée comme suit :

		x	y
(1)	A	i	j

x	y
Slen1	Slen2

La ligne (1) correspond au DPO source A lui-même : aucune opération n'est encore appliquée. Les éléments de l'opérande s'écrivent pour l'instant :

$$\begin{cases} f_x = S+1 \\ f_y = S+1 \end{cases}$$

Opération « exchange(x, y) »

Tel que nous l'avons décrite plus haut, cette opération change la disposition des indices, ici ceux correspondant aux colonnes « x » et « y ». Cette opération interchange également les longueurs définissant les éléments sources. Nous obtenons ainsi la ligne (2) :

		x	y
(1)	A	i	j
(2)	exchange(x,y)	j	i

x	y
Slen1	Slen2
Slen2	Slen1

Les opérations de rotation ne changent pas l'expression des f_i. Aucune règle ne correspond à cette opération. D'où :

$$\begin{cases} f_x \; reste \; f_x \\ f_y \; reste \; f_y \end{cases}$$

Opération « extract(Norig, Nlen) »

À la suite de cette opération, il faut mettre à jour la table. La disposition des indices est la même, mais en ce qui concerne les longueurs nous introduisons les nouvelles correspondant à la partie extraite. Les opérations postérieures se feront par rapport à cette nouvelle partie. La table devient :

		x	y
(1)	A	i	j
(2)	exchange(x,y)	j	i
(3)	extract(Norig,Nlen)	j	i

x	y
Slen1	Slen2
Slen2	Slen1
Nlen1	Nlen2

Jusqu'ici tous les éléments du DPO source étaient référencés dans l'opérande : de l'origine $(1,1)$ jusqu'à $(Slen1, Slen2)$. Or, après l'extraction seule une partie du DPO source reste concernée. C'est pourquoi il faut exprimer ce nouvel espace d'éléments sources dans les éléments f_i de l'opérande. C'est pourquoi nous appliquons la règle R_4 (2e alternative) :

$$\begin{cases} f_y = Norig_1 - 1 + S + 1 & (R_4) \\ f_x = Norig_2 - 1 + S + 1 & (R_4) \end{cases}$$

BU LILLE

Opération « expand(x) »

Au niveau de la table, la disposition des indices et des longueurs ne change pas. Nous rajoutons cependant un astérisque pour la longueur de la colonne « x » concernée par l'expansion. La table mise à jour devient :

		x	y
(1)	A	i	j
(2)	exchange(x,y)	j	i
(3)	extract(Norig,Nlen)	j	i
(4)	expand(x)	j	i

x	y
Slen1	Slen2
Slen2	Slen1
Nlen1	Nlen2
Nlen1*	Nlen2

L'opération de réplication a créé plusieurs copies d'un même objet. Selon la formule, cette opération nécessite l'ajout d'un modulo au niveau de la dimension concernée (la colonne « x »). On ajoutera donc un modulo pour l'expression de f_y. Le modulo portera sur la longueur de la colonne « x », ici $Nlen_x = Nlen1$. L'expression de f_x reste inchangée. L'application de la règle R_1 pour f_y, nous donne :

$$\begin{cases} f_y = Norig_1 - 1 + mod(S, Nlen1) + 1 & (R_1) \\ f_x = f_x \end{cases}$$

Opération « extract(Norig', Nlen') »

Nous avons déjà vu un exemple d'extraction. Nous notons seulement la prise en compte de l'astérisque ($Nlen1^*$) dans la mise à jour de f_y. Nous appliquons donc la règle R_4 (1re alternative) pour f_y. Pour f_x par contre nous appliquons R_4 (2e alternative).

La table après mise à jour devient :

		x	y
(1)	A	i	j
(2)	exchange(x,y)	j	i
(3)	extract(Norig,Nlen)	j	i
(4)	expand(x)	j	i
(5)	extract(Norig',Nlen')	j	i

x	y
Slen1	Slen2
Slen2	Slen1
Nlen1	Nlen2
Nlen1*	Nlen2
Nlen'1	Nlen'2

La double application de la règle R_4, nous donne :

$$\begin{cases} f_y = Norig_1 - 1 + mod(mod(Norig_1' - 1, Nlen1) + S, Nlen1) + 1 & (R_4,\ 1^{ere} alternative) \\ f_x = Norig_2 - 1 + Norig_2' - 1 + S + 1 & (R_4,\ 2^{e} alternative) \end{cases}$$

Opération « exchange(x,y) »

Les opérations de rotation ne modifient pas ici l'expression des f_i décrivant l'opérande. L'interchangement des indices sera pris en compte à la fin, lors du changement d'indices (passage à la table transformée). Au niveau de la table, la disposition des indices et des longueurs est modifiée :

		x	y	x	y
(1)	A	i	j	Slen1	Slen2
(2)	exchange(x,y)	j	i	Slen2	Slen1
(3)	extract(Norig,Nlen)	j	i	Nlen1	Nlen2
(4)	expand(x)	j	i	Nlen1*	Nlen2
(5)	extract(Norig',Nlen')	j	i	Nlen'1	Nlen'2
(6)	exchange(x,y)	i	j	Nlen'2	Nlen'1

Opération « expand(x) »

Enfin la table, à la suite de la dernière opération géométrique, est la suivante :

		x	y	x	y
(1)	A	i	j	Slen1	Slen2
(2)	exchange(x,y)	j	i	Slen2	Slen1
(3)	extract(Norig,Nlen)	j	i	Nlen1	Nlen2
(4)	expand(x)	j	i	Nlen1*	Nlen2
(5)	extract(Norig',Nlen')	j	i	Nlen'1	Nlen'2
(6)	exchange(x,y)	i	j	Nlen'2	Nlen'1
(7)	expand(x)	i	j	Nlen'2*	Nlen'1

L'expansion sur l'axe « x » nécessite l'ajout d'un modulo à l'expression $f_{indice(x)} = f_x$. Le modulo portera sur la longueur $Nlen_x = Nlen'2$. L'expression de f_y par contre ne changera pas. Nous appliquons donc seulement la règle R_1 pour f_x :

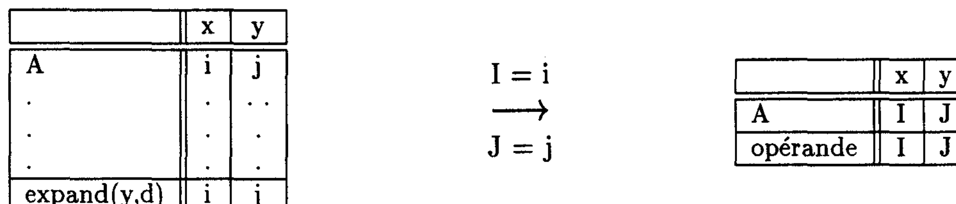
$$\begin{cases} f_x = Norig_2 - 1 + Norig'_2 - 1 + mod(S, Nlen'2) + 1 & (R_1) \\ f_y = f_y \end{cases}$$

Vu que c'est la dernière opération, nous appliquons la règle R_6. Les f_i deviennent :

$$\begin{cases} f_x = Norig_2 - 1 + Norig'_2 - 1 + mod(Indice_x - 1, Nlen'2) + 1 & (R_6) \\ f_y = Norig_1 - 1 + mod(mod(Norig'_1 - 1, Nlen1) + Indice_y - 1, Nlen1) + 1 & (R_6) \end{cases}$$

Changement d'indices

La dernière ligne de la table précédente, nous permet de retrouver les indices nécessaires à l'expression des descripteurs de l'opérande. Nous opérons pour cela la transformation suivante :



	x	y
A	i	j
.	.	. .
.	.	.
.	.	.
expand(y,d)	i	j

$$\mathrm{I} = \mathrm{i} \quad \longrightarrow \quad \mathrm{J} = \mathrm{j}$$

	x	y
A	I	J
opérande	I	J

Ainsi les expressions f_x et f_y utiliserons respectivement les indices I et J puisque $\text{Indice}_x = \text{I}$ et $\text{Indice}_y = \text{J}$. En remplaçant les origines et les longueurs par les valeurs numériques données comme exemple (*cf.* figure V.12, page 198),

$$\begin{cases} Norig1 = p & Norig2 = 2 \\ Nlen1 = 2 & Nlen2 = 3 \\ Norig'1 = 4 & Norig'2 = 2 \\ Nlen'1 = 3 & Nlen'2 = 1 \end{cases}$$

l'opérande sera :

```
A(3, p + mod(J,2))
```

L'indice I n'apparaît pas dans le premier élément de l'opérande, car le modulo « (I-1)[Nlen'2] » se fait sur une longueur de valeur Nlen'2 = 1.

La génération automatique traduisant toute l'expression data-parallèle est la suivante :

```
C   Res = E + A.exchange(x,z).extract(Norig,Nlen).expand(x).
C                extract(Norig',Nlen').expand(x)
    FORALL(I=1:Dlen1, J=1:Dlen2)
       Res(I,J) = E(I,J) + A(3, p + mod(J,2))
```

2.4.6 Discussion

Dans cette exemple, nous voyons bien que pour l'expression data-parallèle entière, HELPDraw ne génère qu'un seul FORALL. L'utilisateur n'a pas à s'inquiéter du nombre d'opé-

rations géométriques qu'il pourrait faire afin de bien positionner son DPO et assurer la conformité, puisque ces opérations sont traduites par une seule opération de communication. Ceci sans doute conforte la souplesse de travail qu'offre HELPDraw pour le développement d'un programme en HPF.

2.5 Le niveau microscopique

En calculant la fonction de description f, nous avons supposé que les opérations géométriques s'appliquaient sur un DPO source S:

$$S.suite_opGom$$

Le résultat que nous obtenons pour exprimer l'opérande est de la forme:

$$S(f_x, f_y, f_z)$$

Or, nous savons que HELPDraw permet de développer un opérande du type *(expr–microscopique).suite_opGéom*, par exemple :

$$(A + B).macro$$

ou

$$(on(C)\ A + B).macro$$

Dans ce cas les opérations géométriques s'appliquent sur un DPO source qui est donc une expression microscopique. Quel est alors le code généré par HELPDraw pour traduire de tels opérandes? ou précisément que devient le calcul de la fonction de description f?

2.5.1 Application d'opérations géométriques à une sous-expression microscopique

En ce qui concerne le premier cas $(A + B).macro$, on sait que le résultat de l'expression microscopique est un temporaire Tmp. On peut écrire alors: $Tmp.macro$. La fonction de description f s'évaluera comme avant en produisant un opérande:

$$Tmp(f_x, f_y, f_z)$$

L'opérande représente en fait :

$$(A + B)\ (f_x, f_y, f_z)$$

Or, puisque le temporaire Tmp est conforme aux arguments de la sous-expression microscopique (ici A et B), la référence (f_x, f_y, f_z) pointe également sur les éléments de A et de B interagissant dans la sous-expression. Nous pourrons écrire ainsi :

$(A + B)(f_x, f_y, f_z)$ est équivalent à $A(f_x, f_y, f_z) + B(f_x, f_y, f_z)$

Ceci est d'ailleurs tout à fait logique car

$(A + B).macro$ est résultat équivalent à $A.macro + B.macro$

Pour montrer comment HELPDraw prend en compte ce type de sous-expression, rappelons que dès qu'un argument est utilisé dans une expression, HELPDraw lui associe une fonction de description f. Le calcul de cette fonction évolue au fur et à mesure de l'application des opérations. Ce calcul ne s'arrête que lorsque toute l'expression est construite. Ainsi, lorsqu'une une opération macroscopique est appliquée à une sous-expression microscopique, elle est appliquée à chaque DPO argument de la sous-expression. D'où, l'application par exemple de l'opération `macro` sur la sous-expression `(A+B)` déclenche la réévaluation des fonctions de descriptions de A et de B.

2.5.2 Les constructions data-parallèles

Les constructions data-parallèles définies dans le modèle géométrique HELP sont le `on` et le `where`. La première construction permet de préciser le domaine de conformité : `on(dpo)` restreint le traitement aux points délimités par la forme de `dpo`. On peut traduire cette construction en HPF de deux manières équivalentes : par les assignations de tableaux, ou par un FORALL. L'instruction suivante par exemple, où les DPO A, B, et C sont bi-dimensionnels avec C de taille (len1,len2), *cf.* figure V.13 :

```
on(C)  A = B + C
```

peut se traduire dans la première solution par (`len`i sont les longueurs de C) :

```
A[onOrigR_A1:onOrigR_A1+len1-1, onOrigR_A2:onOrigR_A2+len2-1] =
B[onOrigR_B1:onOrigR_B1+len1-1, onOrigR_B2:onOrigR_B2+len2-1] + C[:,:]
```

où onOrigR_A (respectivement onOrigR_B) est l'origine du DPO C relative à A (respectivement B) :

$$onOrigR_Ai = origCi - origAi + 1.$$

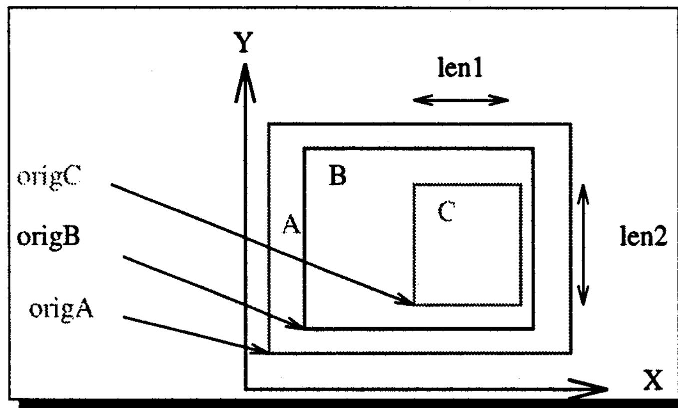


FIG. V.13 - *Exemple d'interaction de* DPO *dans un domaine contraint*

Le constructeur **on** peut se traduire également par un **FORALL** :

```
FORALL(i=1:len1, j=1:len2)
  A(i+onOrigR_A1, j+onOrigR_A2) = B(i+onOrigR_B1, j+onOrigR_B2) +C(i,j)
```

Pour garder la cohérence avec la génération de code que nous avons vue plus haut, HELPDraw utilise la deuxième solution : le **FORALL**. D'autres considérations s'imposent néanmoins : d'une part le **on**, comme le **where** d'ailleurs que nous verrons après, peut apparaître en membre droit d'une affectation, par exemple :

```
Res = (on(C) A+B).macro ...
```

et d'autre part, il peut s'appliquer à un objet issu d'une réplication, au quel cas il ne suffit pas d'ajouter un déplacement aux indices mais il faut recalculer la plage des éléments sources.

Or, nous constatons que le **on** en HPF où il n'y a pas de notion de référentiel mais plutôt d'indices, n'est rien d'autre qu'une extraction d'un sous-objet. Le **on(dpo)** peut être remplacé par un **extract** appliqué à chaque argument de la sous-expression. Les caractéristiques du sous-objet à extraire sont celles du DPO argument du **on**. La sous-expression :

```
on(C) A+B
```

est équivalent à :

```
A.extract_subdpo(origC,lenC) + B.extract_subdpo(origC,lenC)
```

C'est de cette façon que HELPDraw traduit le **on** : il le fait par un extract distribué sur

les arguments de la sous-expression. Dès que le constructeur **on** est désactivé, HELPDraw considère qu'une opération géométrique « **extract** » est appliquée à cet argument. Cette opération déclenche ainsi la réévaluation de la fonction de description de chaque argument de la sous-expression (du segment spécifié par le **on** et des niveaux inférieurs). La sous-expression suivante :

```
(on(C) (A+B).macro1 -C).macro2 - D.macro3
```

peut être traduite en son équivalent :

```
((A.macro1.extract(C).macro2      +
  B.macro1.extract(C).macro2)    -
  C.macro2)     -
  D.macro3
```

À noter que lorsqu'il s'agit de diagonale (**on(diag)**), HELPDraw considère une extraction de diagonale : **extract_diag**. De même pour les DPO triangles, il utilise **extract_triangle**, mais il ajoute le masque définissant le triangle à l'entête du **FORALL**.

La deuxième construction data-parallèle **where(cond_DP)** peut se traduire elle aussi de deux manières équivalentes : soit par un **FORALL** dont le masque est la *cond_DP* (deux **FORALL** lorsqu'il y a **else_where**), soit simplement par le **where** de HPF qui est souvent équivalent à celui du modèle. En effet une légère différence existe ; à l'inverse de HPF, sous HELPDraw la condition du **where** peut faire référence à des coordonnées de points, par exemple : tous les éléments du DPO sauf ceux de la ligne p « **where((A!=0)&&(DIM1!=p))** ».

HELPDraw traduit le **where** par l'ajout du masque représentant la condition data-parallèle, par exemple :

```
FORALL(i=1:Dlen1, J=1:Dlen2, (A(I,J).NE.0).AND.(J.NE.p))
```

Pour l'expression :

```
Res = (on(C) where(A>0) A+B).macro ...
```

HELPDraw génère :

```
FORALL (I=1:Dlen1, J=1:Dlen2, K=1:Dlen3, A(f_x^1,f_y^1,f_z^1).gt.0)
      Res(I,J,K) = A(f_x,f_y,f_z) + B(f_x,f_y,f_z)
ENDFORALL
```

où `A(`f_x^1, f_y^1, f_z^1`)` est la traduction de `A.extract(origC,lenC)`
et `A(`f_x, f_y, f_z`)` est la traduction de `A.extract(origC,lenC).macro`.

2.5.3 Les règles du niveau microscopique

Nous savons qu'à la fin d'une instruction, HELPDraw génère un FORALL de la forme (règle R_7) :

```
FORALL (I=1:Dlen1, J=1:Dlen2, K=1:Dlen3, masque)
     Res(fx,fy,fz) = arg1(fx,fy,fz) op arg2(fx,fy,fz) ...
```

où le nombre de dimensions et la taille de chacune (Dlen*i*) sont déterminé au niveau de l'interprétation visuelle lors de la dernière opération d'affectation. Le masque dans l'entête du FORALL est construit au fur et à mesure de l'application des opérations. Ce masque est rajouté dans deux cas : lorsque le programmeur désactive un **where** (règle R_4), et/ou lorsque une sous-expression manipule des triangles (sous-règle r_3). À chaque fois qu'un nouveau DPO est utilisé dans l'expression, HELPDraw lui associe un code (**nom_dpo(**f_x, f_y, f_z**)**). Le descripteur (f_x, f_y, f_z) de l'opérande est réalisé au fur et à mesure de l'application des opérations géométriques et lorsqu'il y a une désactivation d'un domaine contraint.

La désactivation d'un **on** (règle R_3) fait appel à une fonction g qui est traduite, selon le domaine du **on** (sous-règles r_i, $i = 1$ à 3), comme une opération d'extraction effectuée sur tous les arguments du même segment : **extract_subdpo** si la forme de l'objet argument du **on** est régulière (sous-règle r_1), **extract_diag** lorsqu'il s'agit d'une diagonale (sous-règle r_2), et **extract_triangle** pour un triangle (sous-règle r_3).

À noter que l'interprétation d'un **on** par une extraction, fait « fondre » la hiérarchie des segments conformes. Il n'y aura qu'un seul segment pour toute l'expression. Chaque segment conforme se fondra dans le niveau supérieur. Par exemple la sous-expression suivante où il y a deux niveaux de segments conformes :

```
on(dpo1) A+B - (on(dpo2) C-D)
```

peut se traduire par :

```
A.extract_subdpo(origdpo1,lendpo1) +
B.extract_subdpo(origdpo1,lendpo1) -
(C.extract_subdpo(origdpo2,lendpo2).extract_subdpo(origdpo1,lendpo1) -
```

TAB. V.2 - *Les règles utilisées pour la génération de code HPF concernant les opérations microscopiques*

- Tout DPO a deux propriétés : c et p
  - c : son code — $c(\text{arg}) \rightarrow$ le code associé à *arg*
  - p : sa priorité (*cf.* III.23) — $p(\text{arg}) \rightarrow$ la priorité associée à *arg*
- Tout DPO variable a ses propriétés initialisées à :
  $c = nom_dpo\ (f_x, f_y, f_z)$
  $p = p(nom_dpo)$
- Pour toute opération microscopique, calculer les propriétés du résultat en appliquant l'une des règles suivantes :

  Soit la fonction f qui renvoie l'une des deux valeurs selon la priorité de **op** :

$$f(\text{arg}, \text{op}) \longrightarrow (\ c(\text{arg})\ ) \quad \underline{\textbf{si}}\ p(\text{arg}) < p(\text{op})$$
$$\longrightarrow c(\text{arg}) \quad \underline{\textbf{sinon}}$$

R_1	[*opBinaire*[a], *arg1*, *arg2*]	: $c =$	f(arg1,opBinaire) **opBinaire** f(arg2,opBinaire)
		$p =$	p(opBinaire)
R_2	[*opUnaire*, *arg*]	: $c =$	***opUnaire*** f(arg, opUnaire)
		$p =$	p(opUnaire)
R_3	[*désactiver-on*, *domaine*, *arg*]	: $c =$	g(domaine) c(arg)
		$p =$	p(on)
	r_1 $g(domaine = régulier)$	$\longrightarrow$	*appliquer* ***extract_subdpo***[b]
	r_2 $g(domaine = diag)$	$\longrightarrow$	*appliquer* ***extract_diag***
	r_3 $g(domaine = triangle)$	$\longrightarrow$	*appliquer* ***extract_triangle*** *et* *ajouter* masque_triangle *au* **FORALL**
R_4	[*désactiver-where*, *domaine*, *arg*]	: $c =$	*ajouter* masque-where *au* **FORALL**
		$p =$	p(where)
R_5	[*désactiver-segt-non-contraint*, *arg*]	: $c =$	c(arg)
		$p =$	p(on)
R_6	[*affectation.sans.on*[c], *arg1*, *arg2*]	: $c =$	c(arg1) = c(arg2) g(*domaine-arg2*)
		$p =$	p(=)
R_7	[*fin*]	: $c =$	**Forall**(I,J,K ∈ Domaine, Masque) c(arg)

[a] *opBinaire* différent de celui de la règle R_6.
[b] Les extractions déclenchées par les sous-règles r_1, r_2 et r_3 sont appliquées en même temps à tous les arguments du même segment (même en membre gauche d'une affectation) ainsi que ceux des niveaux inférieurs.
[c] affectation sans « on » au niveau du segment principal.

```
    D.extract_subdpo(origdpo2,lendpo2).extract_subdpo(origdpo1,lendpo1)
   )
```

`origdpo`*i* et `lendpo`*i* représentent respectivement les coordonnées de l'origine et les longueurs du DPO dpo*i*.

Les opérations arithmétiques et logiques sont appliquées comme nous l'avons vu au niveau de C-HELP ou la construction des historiques en HELPDraw (règle R_1, R_2). Lorsque par contre l'opérateur binaire est une affectation où il n'y a pas de constructeur **on** à gauche de cette affectation, HELPDraw applique la règle R_6 qui permet de préciser au niveau des fonctions de description, le domaine d'activité exacte de tous les arguments de l'expression (appel de la fonction g).

Maintenant que nous avons vu les expressions data-parallèles ainsi que la traduction des constructions data-parallèles, nous allons illustrer la génération automatique par un exemple concret : inversion de matrice.

3 Exemple de génération de programmes : Inversion de matrice

L'exemple suivant « inversion de matrice de Gauss-Jordan » illustre au mieux les mécanismes de traduction permettant de générer un code HPF fidèle à un programme écrit à la main..

Nous voulons inverser une matrice carrée A de taille (NxN). Soit une matrice M de taille (2NxN) divisée en deux parties ; la première partie M[1..N] lui est affectée A et la seconde M[N+1..2N] la matrice identité Id (M(2NxN)=$[A|I_d]$). Cet algorithme de Gauss-Jordan se fait en deux étapes principales : diagonalisation de la partie A de M (M[1..N]), puis division de chaque $ligne_i$ de M par l'élément i de la diagonale.

Diagonalisation de A Pour annuler les éléments (dans la partie A) d'une colonne p, nous appliquons d'abord pour toute ligne r (r≠p) :

$$ligne_r = ligne_r * \frac{d_p}{m_{rp}} - ligne_p$$

L'élément $\frac{d_p}{m_{rp}}$ est le pivot. Ceci annule à chaque élément d_p, les autres éléments de la colonne p

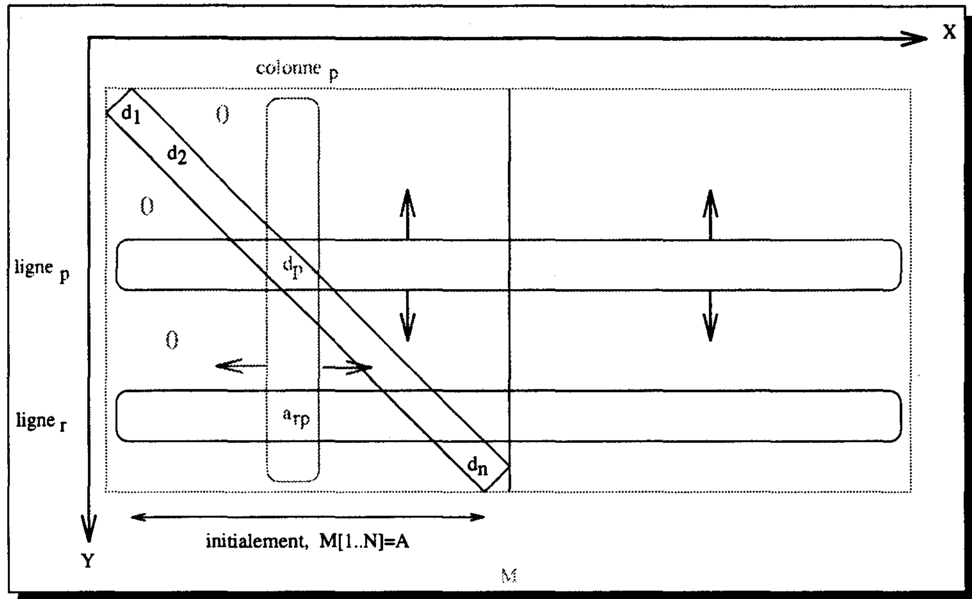


FIG. V.14 - *Inversion: diffusion des lignes et colonnes*

(d_p est le p^{eme} élément de la 1ere diagonale). Afin d'avoir les éléments a_{rp} (première équation) et toute $ligne_p$ le long des lignes r (seconde équation) et sachant que les communications sont explicites, nous diffusons la colonne pivot p sur les autres colonnes et la ligne pivot p sur les autres lignes (figure V.14). Les éléments d_p sont référencés comme des scalaires.

Division Après la diagonalisation, chaque ligne doit être divisée par l'élément d_r de la diagonale d

$$ligne_r = ligne_r / d_r \ ,$$

car il faut avoir une matrice $[I_d A^{-1}]$. Il suffit de diffuser les éléments de la diagonale sur toutes les colonnes, pour cela : nous effectuons une rotation (rotate) de la diagonale pour avoir une colonne (x=1), puis la diffuser (figure V.15).

À la fin, la partie droite de M (la partie N+1..2N) représentera la matrice inversée A^{-1}.

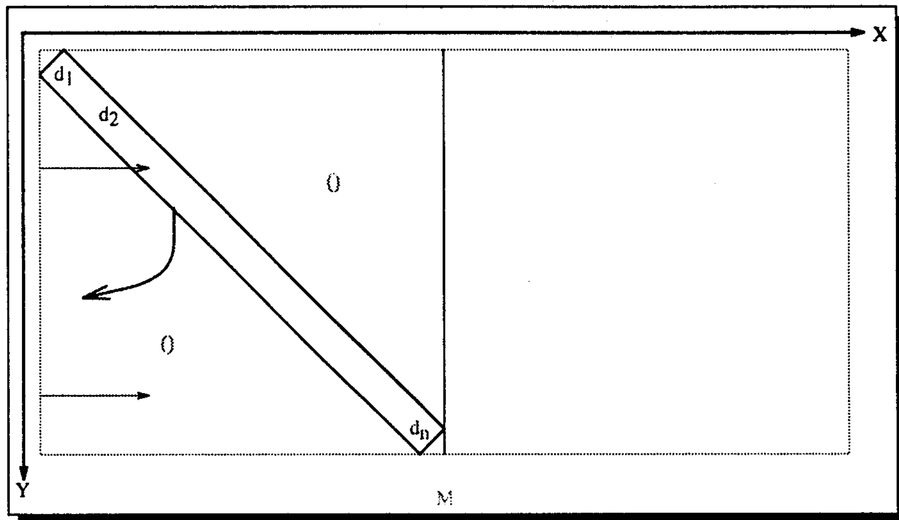


FIG. V.15 - *Rotation (diagonale vers colonne), puis diffusion*

Le code géométrique

```
/* Partie Spécification */
hspace my_hspace [x=2*N, y=N] ;
DPO float my_hspace[x=2*N, y=N] M ;
DPO float my_hspace[x=N, y=N] A;
DPO float my_hspace[x=N+1:2*N, y=N] Id ;
DPO float my_hspace[N+1:2*N, y=N] A_inv;


/* Partie Traitement */

/* Initialisation */
/* on suppose que A est déjà initialisé */
M = A ;
M = Id ;

/* Diagonalisation */
for (p=1; p<=N; p++) {
  where (Dim1 != p)
     M = M *
         (M(p,p) /
           M.extract_subdpo(p,1,1,N).move(1,1).expand(x))  -
         M.extract_subdpo(1,p,2*N,1).move(1,1).expand(y);
}

/* Division */
M = M /
    M.extract_diag(1,1,N).rotate_toline(y).expand(x) ;

/* Résultat */
on (A_inv) A_inv = M;
```

Le code HPF

```
C ---Partie Spécification---
C  DPO float my_hspace[x=2*N,y=N] M;
      real M(2*N,N)
C  DPO float my_hspace[x=N,y=N] A;
      real A(N,N)
C  DPO float my_hspace[x=N+1:2*N,y=N] Id;
      real Id(N,N)
C  DPO float my_hspace[x=N+1:2*N,y=N] A_inv;
      real A_inv(N,N)

C ---Partie Traitement---
C---Initialisations---
C M = [A | Id]
      M(1:N,:) = A
      M(N+1:2*N,:) = Id

C---Diagonalisation
      DO p = 1,N
         forall (I=1:2*N, J=1:N, (J.ne.p))
            M(I,J) = M(I,J)*(M(p,p)/M(p,J)) - M(I,p)
         endforall
      END DO

C---Division
     forall(I=1:2*N,J=1:N)  M = M(I,J)/M(J,J)

C---Résultat
C---On(A_inv) A_inv = M
      forall (I=1:N, J=1:N)  A_inv(I,J) = M(I+N,J)
```

4 Conclusion

Nous avons vu dans le chapitre III l'avantage principal qu'offre HELPDraw. L'utilisateur exprime sa pensée data-parallèle sans être contraint par une quelconque syntaxe ou architecture particulière. Dans ce chapitre, nous retrouvons d'autres avantages offerts par HELPDraw. Le premier concerne la portabilité du code qui est automatiquement généré dans un langage « candidat standard » : le langage High Performance Fortran. L'utilisateur bénéficie des avantages de HPF sans être confronté aux problèmes dus à la programmation par les indices. Le deuxième avantage de HELPDraw réside dans les mécanismes d'optimisation qu'il met en œuvre pour la génération de code. Il retrouve pour chaque opérande la fonction de description permettant de retrouver directement les éléments sources. Il combine l'ensemble des opérations de communication d'un opérande, puis toute l'expression data-parallèle en un seul FORALL généralement en une seule phase. Le programmeur n'a pas besoin de trouver la suite

d'opérations géométriques la plus optimale qui lui permette de bien positionner un objet ou assurer une conformité. Ceci facilite encore plus le développement de programmes sous HELPDraw.

L'intérêt principal de l'utilisation des hyper-espaces est de séparer les communications des traitements. L'évaluation d'une expression ne doit générer aucune communication implicite. Or, comme nous venons de le voir en HPF, cet intérêt est totalement perdu puisque nous étions obligés, dans la solution retenue, de générer un code HPF sans tenir compte des alignements des DPO par rapport à l'hyper-espace. À l'inverse de la génération de code C-HELP, lors de la génération de HPF l'hyper-espace devient seulement un espace de travail. La question qui s'impose par conséquent, est : « y a-t-il un autre moyen de programmation visuelle (autre que de se baser sur le modèle HELP) pour le parallélisme de données? ». Pour répondre à cette question dores et déjà des travaux étudiant d'autres techniques visuelles sont entrepris dans notre équipe.

Enfin HELPDraw peut s'adapter à d'autres langages que C-HELP et HPF, en particulier HPC (High Performance C) [VBF94] dont les concepts sont étroitement similaires à ceux de HPF, et encore mieux en ce qui concerne la syntaxe C qui est plus flexible.

Chapitre VI

Illustration des différentes étapes de développement d'un programme sous HELPDraw

Pour mieux illustrer le développement sous HELPDraw, nous avons choisi de développer un petit programme. Nous donnons toutes les étapes par lesquelles il faut passer, de la conception géométrique de l'algorithme jusqu'à la génération de code.

Nous devions choisir entre plusieurs exemples de programmes, mais nous avons sélectionné la multiplication de matrices sur une grille 3-D. Ce choix est motivé par deux raisons : d'une part pour la taille de l'exemple qui est relativement court ce qui rend plus ou moins simples nos explications, et d'autre part pour le contenu de l'algorithme car il nécessite plusieurs opérations géométriques appliquées consécutivement.

Les étapes que nous allons abordés sont les suivantes :

- la conception géométrique de l'algorithme,
- réalisation de l'algorithme sous HELPDraw,
- et enfin voir les deux codes produits C-HELP et HPF.

1 Conception géométrique

Généralement dans un code data-parallèle, la multiplication de matrices se traduit par une boucle itérative de décalages, de multiplications et d'additions (*cf.* chapitre IV : 3). Dans cet

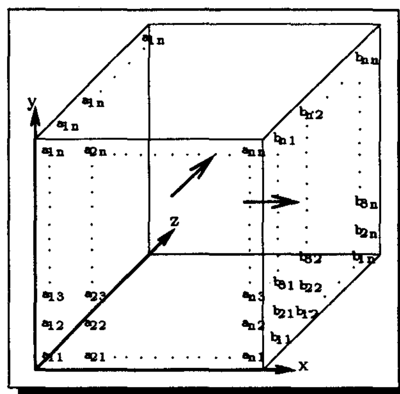


FIG. VI.1 - *A∗B sur une grille 3D : A répliquée selon l'axe 'z' et B (après rotation) répliquée selon l'axe 'x'*

exemple par contre, nous voulons réaliser la multiplication en évitant d'écrire une boucle. L'idée ici est de pouvoir exécuter l'opération de multiplication en une seule passe (sans boucle) ; donc un nombre d'instructions indépendant de la taille de la matrice. Implicitement la boucle est remplacée par la troisième dimension d'un hyper-espace 3D. Il faut arriver à avoir donc à un moment donné toutes les paires d'éléments nécessaires à la multiplication (multiplication de chaque ligne de la première matrice et les autres colonnes de l'autre matrice). Géométriquement c'est très simple puisque cela revient à faire des réplications après avoir bien positionné la matrice à répliquer. Il faut ensuite appliquer l'autre opération nécessaire à la multiplication de matrices : l'addition des éléments émanant du produit ligne-colonne. Ceci peut être réalisé par une opération de réduction. Voici les étapes à suivre (les étapes 2 à 6 représentent le produit ligne-colonne, et l'étape 7 représente les additions) :

1. Au début, les trois matrices A, B et C sont placées sur le plan [z=1] de l'hyper-espace 3D.
2. La matrice A est répliquée le long de l'axe 'z' (figure VI.1).
   « **`réplication de A (axe=z)`** »
3. Nous ferons une rotation de la matrice B par rapport à l'axe 'x' (donc du plan [xy] vers le plan [xz]).
   « **`rotation de B (plan: xy→xz)`** »
4. Une autre rotation est appliquée au résultat de l'étape (3), cette fois par rapport à l'axe 'z' (donc du plan [xz] vers le plan [yz]).
   « **`rotation de Tmp(3)`**[1] **`(plan: xz→yz)`** »
5. Maintenant que la matrice B (ou en fait une copie de B) a été bien positionnée, elle est répliquée le long de l'axe 'x'.
   « **`réplication de Tmp(4) (axe=x)`** »

1. Tmp(i) : le temporaire résultant de l'étape i.

6. La multiplication peut être exécutée, puisque nous avons maintenant toutes les paires d'éléments nécessaires (un élément de A avec un autre de B) ; chaque paire sur un point de l'hyper-espace. Cette multiplication représente le produit de chaque ligne de A avec les colonnes de B.
   « `Tmp(2) * Tmp(5)` »

7. Pour réaliser l'opération d'addition, il faut appliquer au résultat de l'étape (6) une opération de réduction (reduceadd) le long de l'axe 'y'.
   « **`réduction de Tmp(6) (axe=y)`** »

8. Et enfin pour retrouver le résultat sur le plan [z=1] (conforme à C), il faut appliquer une rotation au résultat de la réduction ; une rotation par rapport à l'axe 'x' (du plan [xz] vers le plan [xy]).
   « **`rotation de Tmp(7) (plan: xz→xy)`** »

9. Affecter le résultat de la précédente rotation à la matrice C.
   « `C = Tmp(8)` »

2 Réalisation sous HELPDraw

Étape 1 La définition de l'hyper-espace est réalisée à travers la boîte de dialogue de la figure VI.2. Nous avons appelé cet hyper-espace de taille (100×100×100), 'grille3D'. À l'intérieur nous allons y allouer les DPO A, B et C manipulés par l'algorithme. Dans un premier temps, nous définissons le DPO A sur le plan [z=1] de l'hyper-espace. Nous lui avons attribué la couleur cyan (*cf.* figure VI.3). Les déclarations correspondant à l'hyper-espace et au DPO A apparaissent dans le buffer, où le code ici est C-HELP.

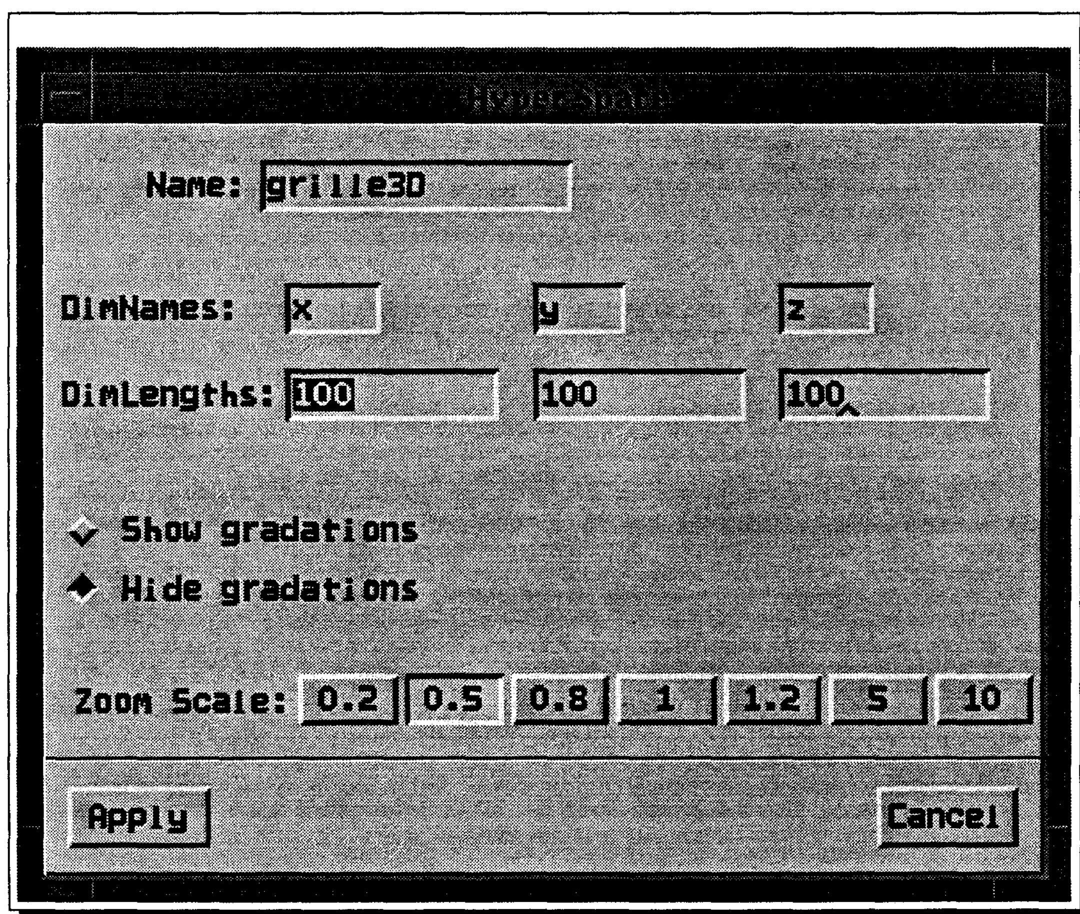


FIG. VI.2 - *Définition de l'hyper-espace 3-D*

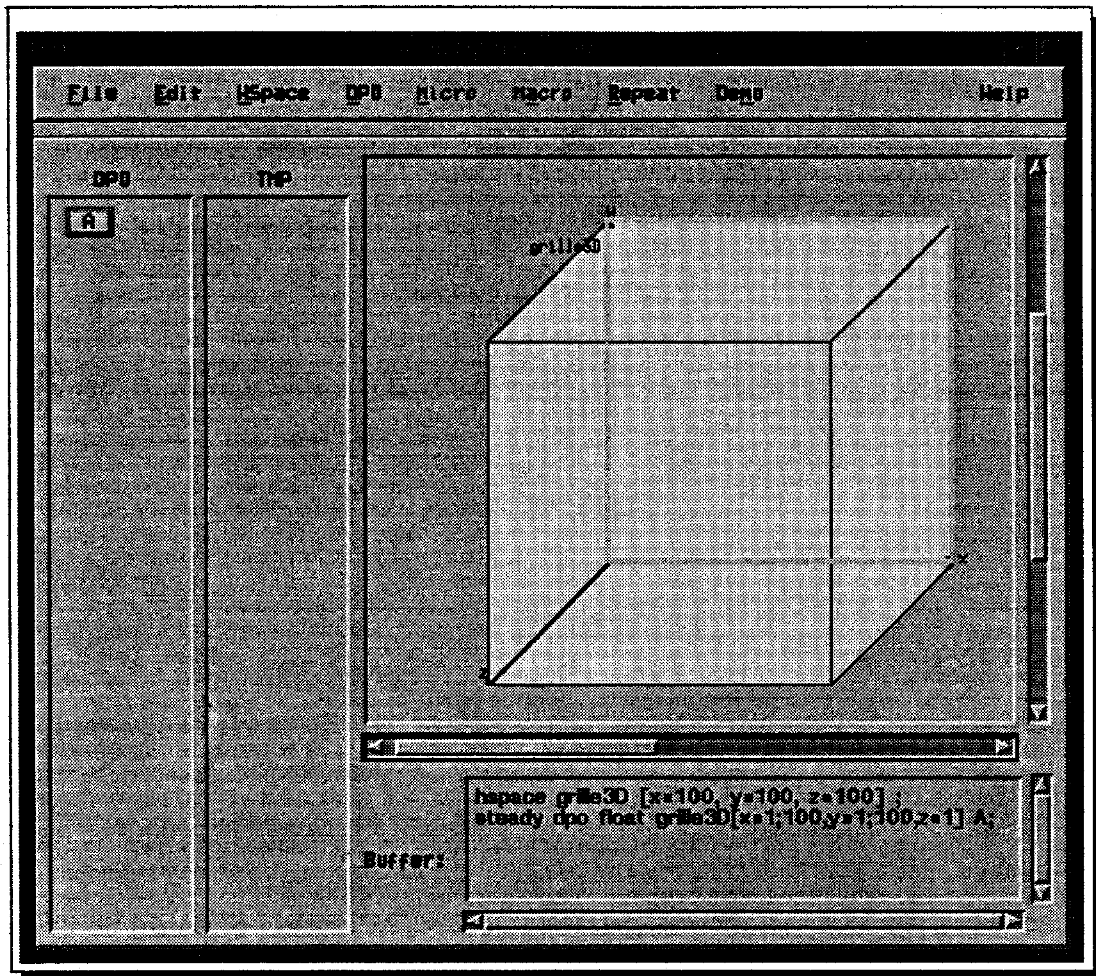


FIG. VI.3 - *Définition du* DPO *A sur le plan [z=1]*

Étape 2 Au DPO A, nous avons appliqué une opération d'expansion le long de l'axe 'z'. La figure VI.4 montre l'objet temporaire résultant ; il est de couleur verte. La boîte 'HISTORY' montre l'historique de ce temporaire.

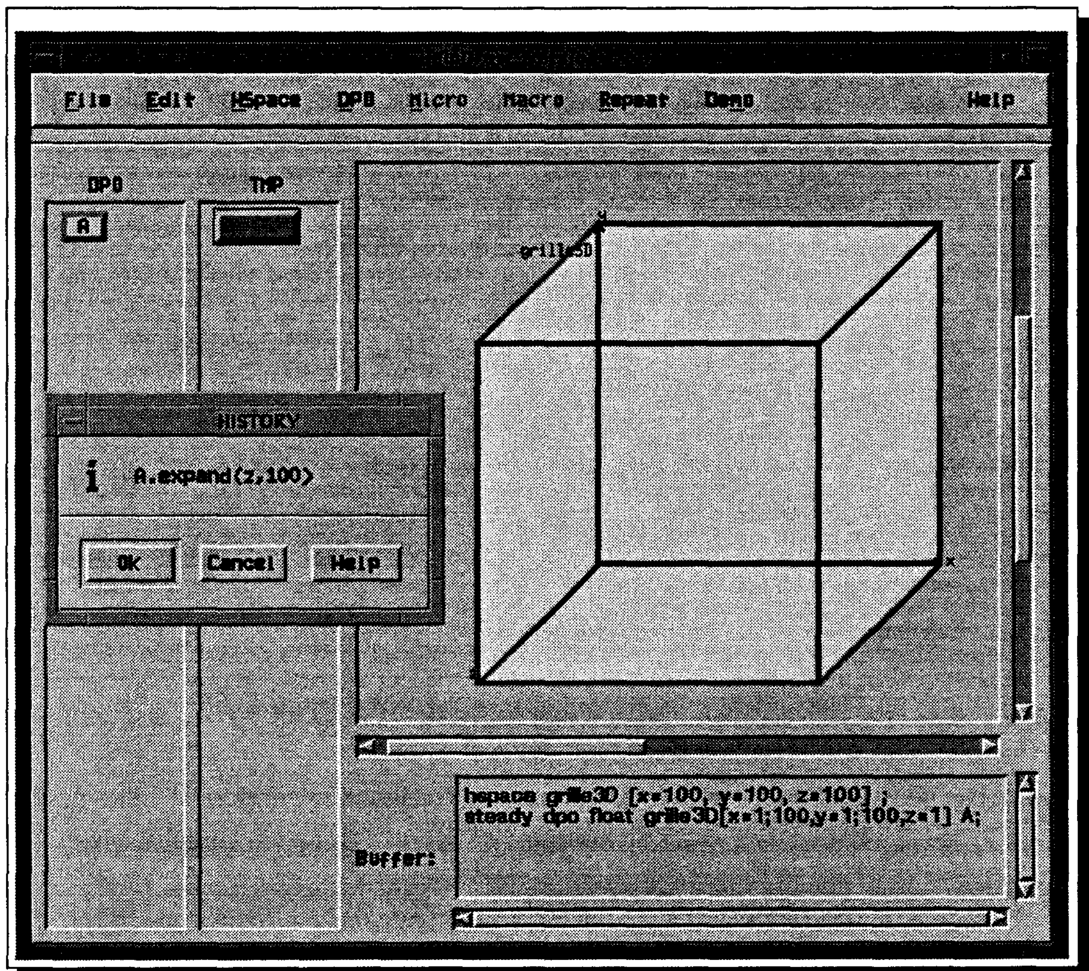


FIG. VI.4 - *Réplication du* DPO *A le long de l'axe 'z'*

Étapes 3,4,5 Nous décrirons les étapes 3, 4, et 5 à la fois. De la même manière qu'avant, nous définissons le second DPO B, il est de couleur marron. Nous lui appliquons deux opérations de rotation consécutives, la première par rapport à l'axe 'x' et la seconde par rapport à l'axe 'z'. (En fait la seconde rotation est appliquée au temporaire résultant de la première rotation.) Nous effectuons ensuite une opération de réplication le long de l'axe 'x'. Nous obtenons le temporaire rouge de la figure VI.5. On voit également dans cette figure l'historique correspondant au temporaire (`B.exchange(z,y).exchange(x,y).expand(x,100)`).

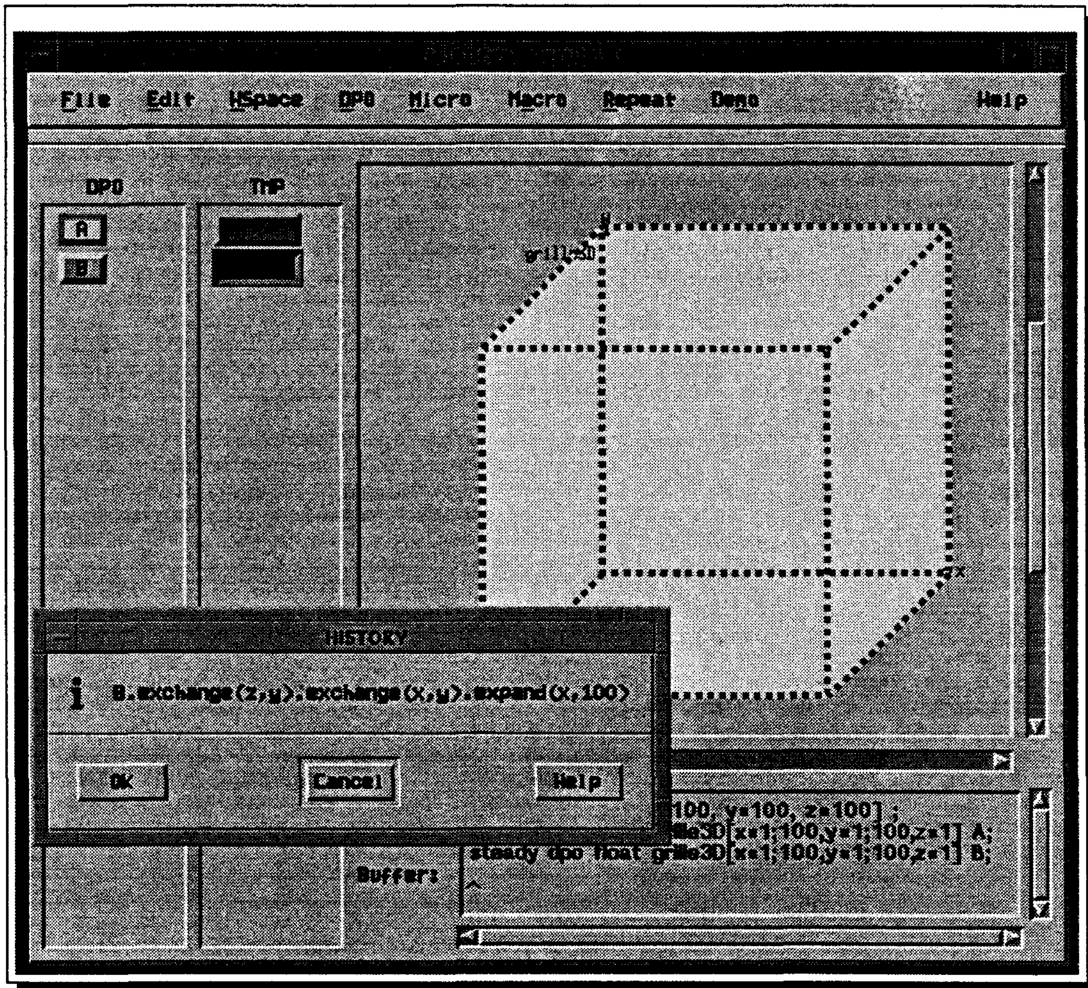


FIG. VI.5 - *Le temporaire résultant des opérations de rotation et de réplication appliquées au* DPO *B*

Étape 6 On peut appliquer maintenant l'opération de multiplication aux deux temporaires vert et rouge. Cette opération microscopique peut être évoquée directement à travers le menu intitulé 'Micro', mais juste pour montrer l'utilisation des expressions encapsulées nous avons utilisé l'expression « `dpo1*dpo2` ». La figure VI.6 montre l'expression après avoir fait la correspondance entre les arguments formels et les temporaires (rouge et vert). Les arguments `dpo1` et `dpo2` prennent respectivement la couleur du temporaire attaché, donc verte pour le premier et rouge pour le second.

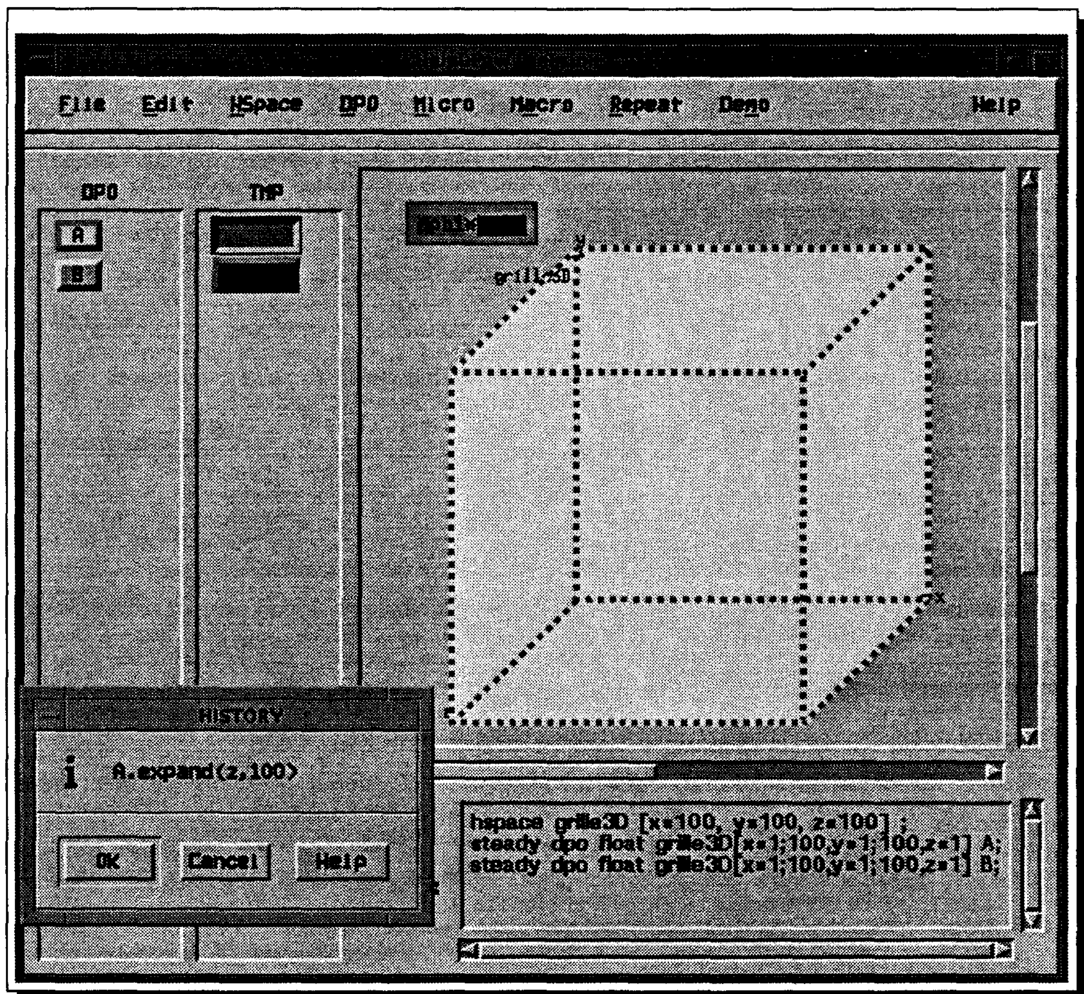


FIG. VI.6 - *Réalisation de la multiplication par l'intermédiaire d'une expression encapsulée*

Étapes 7,8 Il n'est pas nécessaire de détailler les opérations géométriques restantes (réduction et rotation). Elles sont réalisées de la même manière que les opérations géométriques précédentes. La figure VI.7 montre le temporaire 'vert' créé à la suite de ces opérations. Nous voyons également l'historique lui correspondant :

```
(A.expand(z,100)*B.exchange(z,y).exchange(x,y).expand(x,100)).
                     reduceadd(y).exchange(z,y)
```

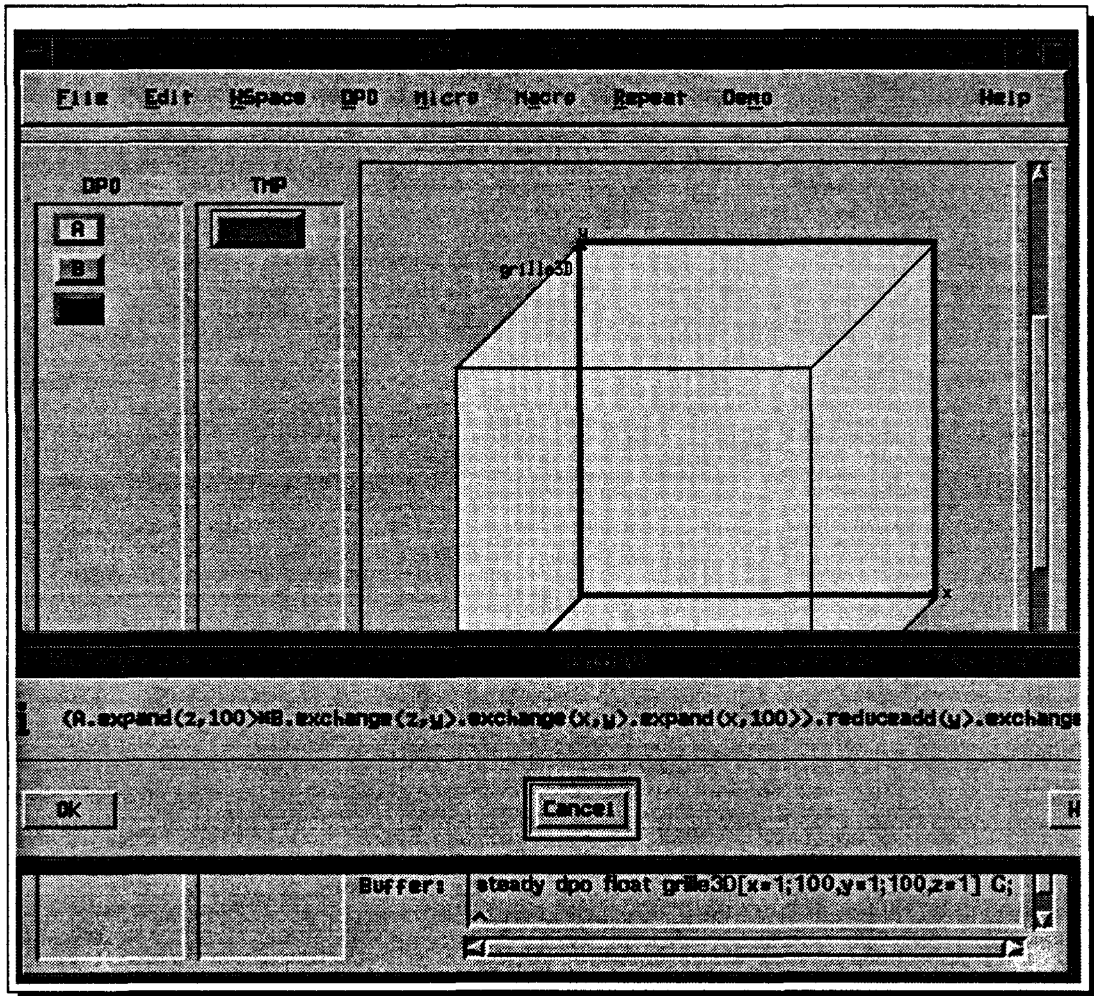


FIG. VI.7 - *Le résultat du produit de matrices*

Étape 9 Le temporaire 'vert' représente toute l'expression calculant le produit de matrices. Il suffit maintenant d'affecter ce temporaire au DPO C (rouge), pour avoir enfin le reste du code.

3 Génération de codes

Nous donnons dans cette section les deux types de codes générés pour l'algorithme que nous venons de développer : produit de matrices sur une grille 3-D.

3.1 Le code C-Help

```
/* Spécification */
  hspace grille3D [x=100, y=100, z=100];
  steady DPO float grille3D[x=1;100,y=1;100,z=1] A;
  steady DPO float grille3D[x=1;100,y=1;100,z=1] B;
  steady DPO float grille3D[x=1;100,y=1;100,z=1] C;

  /* Traitement */
  C = (A.expand(z,100) * B.exchangeabs(z,y).exchangeabs(x,y).
        expand(x,100)).reduceadd(y).exchange(z,y) ;
```

3.2 Le code HPF

```
      real A(100,100)
      real B(100,100)
      real C(100,100)
      real Tmp(100,100,100)

C     La suite d'opérations avant la réduction
      Forall(I=1:100, J=1:100, K=1:100)
       Tmp(I,J,K) = A(I,J) * B(J,K)
      Endforall

C     La réduction
      Tmp(:,1,:) = sum(Tmp, DIM=2)

C     Rotation, puis affectation du résultat
      Forall(I=1:100, J=1:100)
        C(I,J) = Tmp(I,1,J)
      Endforall
```

4 Résumé

Le but de ce chapitre était de résumer les différentes étapes que pourrait entreprendre un utilisateur de HELPDraw afin de mettre en œuvre un algorithme data-parallèle. Nous avons illustrer ces étapes pas à pas à travers un exemple simple : la multiplication de matrices.

Conclusion

1 Résumé des travaux

Nous avons montré dans cette thèse l'intérêt des environnements graphiques dans le domaine du parallélisme de manière générale. Nous avons mis en évidence la carence de ces environnements pour le parallélisme de données en particulier au niveau de la conception et de l'implémentation des programmes.

Pour contribuer à combler cette lacune, nous avons orienté nos travaux vers la programmation visuelle. Notre objectif est d'offrir un environnement visuel convivial permettant, à partir de simples interactions avec le programmeur, de produire automatiquement un code data-parallèle. Afin de mieux comprendre la programmation visuelle, nous avons décrit les différentes techniques de programmation visuelle tout en expliquant les avantages et les inconvénients.

Nous avons mis en œuvre HELPDraw : un environnement de programmation visuelle pour le parallélisme de données. Cet environnement a un double apport :

1. Les scientifiques peuvent avoir besoin d'écrire des programmes data-parallèles sans se soucier de la syntaxe d'un langage particulier ou de l'architecture de la machine cible. De par son aspect visuel, HELPDraw les assiste dans leurs conceptions data-parallèles et dans la traduction de leurs algorithmes en codes C-HELP et HPF qu'il génère automatiquement.

2. HELPDraw permet de mettre en valeur l'intérêt de la programmation géométrique. En se basant sur le modèle de programmation data-parallèle géométrique HELP (Hyper-Espace et Langage Parallèle), il montre la facilité de concevoir et de programmer data-parallèle en distinguant clairement les différentes manipulations géométriques appliquées aux objets data-parallèles.

2 Perspectives

HELPDraw ne génère que des parties de codes : déclarations, instructions, et constructions de contrôle. Le programmeur insère les codes produits là où il le souhaite à l'intérieur de son programme. Il peut insérer une instruction manipulant par exemple une variable « A » à un endroit du programme où la portée de la variable ne l'atteint pas (la déclaration de « A » a été insérée ailleurs). Ceci est dû au fait que HELPDraw n'a pas de contrôle sur le programme. Les instructions développées ne sont pas liées au contexte du programme. Cette gestion est laissée au programmeur. Une meilleure utilisation de HELPDraw nécessite la génération automatique et transparente d'un *programme complet*. Nous expliquons ci-après notre façon de voir cette évolution (*cf.* 2.1).

Nous nous intéressons aussi à la partie duale de HELPDraw : la visualisation de programmes data-parallèles. Un outil permettant de visualiser l'algorithmique correspondant à un code data-parallèle constituerait sûrement une bonne contribution à réduire les difficultés de développement. C'est dans cet esprit que nous proposerons une piste pouvant conduire à la mise en œuvre d'un tel outil, bien que notre proposition soit liée à l'environnement HELPDraw (*cf.* 2.2).

2.1 Un langage de programmation visuelle data-parallèle

Afin de pouvoir produire automatiquement un programme data-parallèles plus complet[2], nous proposons d'orienter HELPDraw vers un langage de programmation visuelle impératif structuré. Lors du développement d'un programme HELPDraw, nous distinguerons deux principaux niveaux d'abstraction. Le premier est global : il permet d'une part d'exprimer le squelette et la structure du programme et d'autre part de spécifier le contexte de développement des instructions de base du programme. Le second niveau concerne le développement de ces instructions (data-parallèles ou scalaires). HELPDraw pourrait utiliser pour chaque niveau une technique de spécification différente : « *programmation par HyperBoîtes* » pour le premier, et « *programmation par les exemples* » pour le deuxième.

La figure VI.8 montre le schéma général de ce que pourrait être le développement d'un programme HELPDraw. À partir d'une conception globale de la solution, l'utilisateur dessine le squelette de son programme. Cette construction cporrespond à une hiérarchie de boîtes, chacune représente une fonction ou construction de contrôle vides. Ceci est réalisé dans un éditeur graphique, appelé « éditeur de squelette ». Ces boîtes sont ensuite remplies avec des déclarations (uniquement pour les boîtes « fonctions »), et des instructions scalaires ou data

2. Nous précisons tout de suite que cette proposition n'est qu'une étape, bien que importante, pour arriver à la génération d'un programme complet. Il restera encore des points à étudier ultérieurement tels que les outils d'entrées/sorties (exemple : afficher le contenu d'une variable).

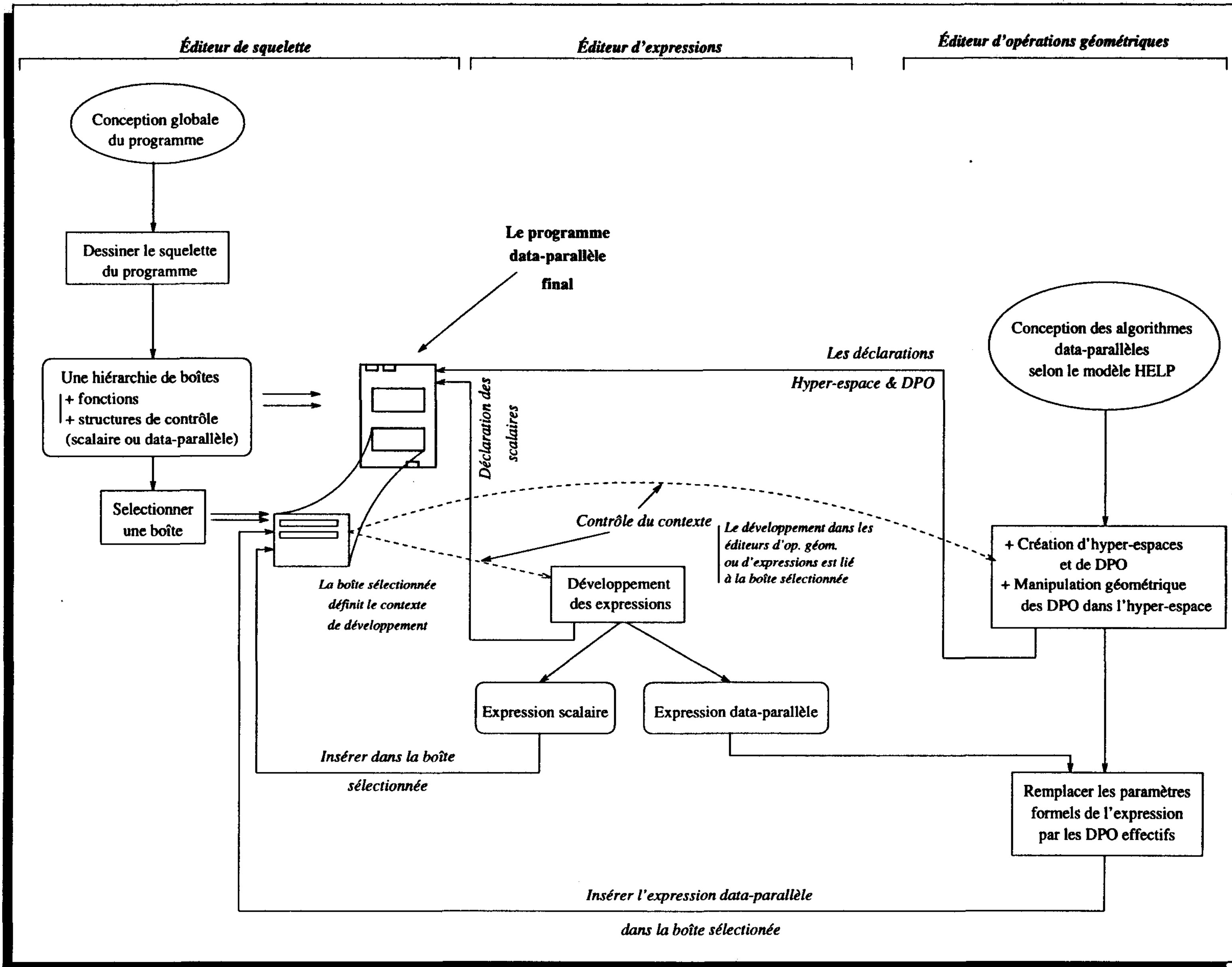


FIG. VI.8 - *Le schéma global de développement d'un programme* HELP*Draw*

parallèles. Le programmeur sélectionne d'abord la boîte à remplir, il passe ensuite au développement des instructions dans les éditeurs d'opérations géométriques et d'expressions. Ces deux éditeurs sont liés au contexte de la boîte sélectionnée. Il n'apparaîtra par exemple dans l'éditeur d'opérations géométriques que les DPO dont la portée atteint la boîte sélectionnée.

Nous ne reviendrons pas ici sur le développement des instructions qui a été décrit au chapitre III. Nous ne verrons que la programmation par *HyperBoîtes* pouvant être utilisée pour l'expression du squelette d'un programme.

2.1.1 Définition du squelette du programme

Un programme HELPDraw serait une hiérarchie de boîtes ; chacune représente une fonction, une structure de contrôle, ou un bloc d'instructions. Nous avons qualifié cette technique de *programmation par HyperBoîtes* parce que d'une part une boîte peut en contenir d'autres, et d'autre part il existe des liens permettant de naviguer entre les différentes structures du programme (exemple : passer directement de l'appel d'une fonction à la boîte où elle est définie). Ces liens, que nous appelons précisément :*HyperLiens*, sont similaires à ceux définis par P.J. Lyons [LSA93] (cf. chapitre II, sous-section 3.1.4, page 54). Cette technique de spécification, en plus de la possibilité de définir le squelette, nous permet de résoudre le problème de l'abstraction procédurale dont l'importance a été évoquée au chapitre II.

Nous n'avons pas l'intention de définir ici la grammaire visuelle du langage qui est similaire à celle des autres langages structurés. Nous décrivons par contre l'ensemble des boîtes pouvant être définies dans le langage de programmation visuelle HELPDraw.

HELPDraw doit offrir et gérer les boîtes suivantes permettant d'exprimer :

1. Le regroupement d'une séquence d'instructions : construction de séquence « *Seq* ».
2. La construction conditionnelle « *If_Then_Else* ».
3. Le constructeur de domaine contraint « *On* ».
4. Le constructeur de domaine masqué « *Where_Elsewhere* ».
5. Les constructions itératives :
   - la construction « *While* » ;
   - la construction « *Repeat* ».
6. La structure d'une « *Fonction* ».

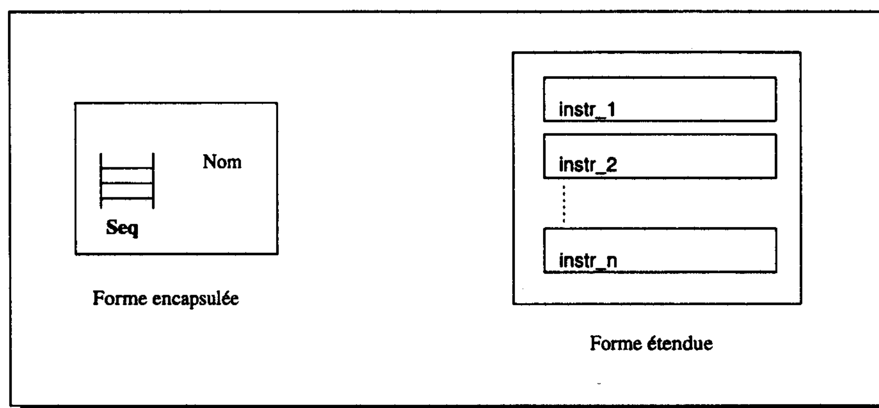


FIG. VI.9 - *La représentation graphique de la construction « Seq »*

2.1.1.1 La construction de séquence « Seq » La construction « *Seq* » représente un bloc d'instructions, tel qu'il peut être défini dans tous les langages structurés textuels (par exemple : « *Begin...End* » de Pascal ou Ada). Un élément de cette boîte peut être une instruction (scalaire ou data-parallèle) ou *une autre boîte.* La figure VI.9 montre les deux formes de la boîte (encapsulée et étendue). La première forme est un simple icône représentant la boîte. Le programmeur la place dans le squelette pour indiquer qu'il veut insérer une séquence d'instructions.

2.1.1.2 Les autres constructions de contrôles Les constructions de contrôles de 2 à 5 ont été décrites au chapitre III (section 3.1, page 132). Seule l'abstraction procédurale doit être rajoutée à ce niveau. Nous avons vu que HELPDraw insère successivement, dans une boîte donnée, les instructions développées par l'utilisateur. Celui-ci doit pouvoir en plus insérer des boîtes d'instructions.

2.1.1.3 La structure d'une « Fonction » La fonction est définie par trois parties: les arguments (les paramètres d'E/S), une partie spécification, et une partie traitement. La figure VI.10 montre ce que pourrait être la représentation graphique d'une boîte « fonction » :

1. Les arguments sont représentés par des cellules, les entrées en haut de la boîte et les sorties en bas. Ces cellules sont explicitement placées par le programmeur.
2. La partie spécification doit contenir l'ensemble des déclarations et définitions locales à la fonction : déclaration de variables ou de constantes.
3. La partie traitement est similaire aux corps des différentes constructions que nous avons vues. Le programmeur peut insérer une des boîtes précédemment citées (*Seq, While*,...) ou une seule instruction élémentaire.

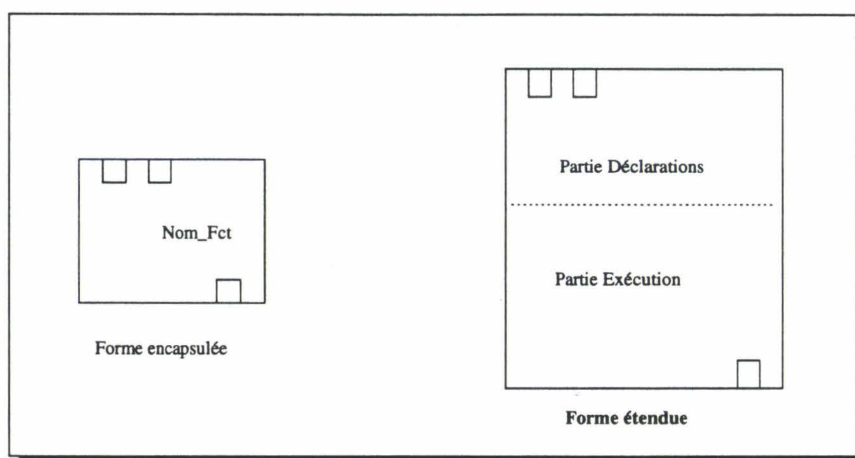


FIG. VI.10 - *La représentation graphique d'une structure « Fonction »*

2.1.2 Exemple de programme

Nous avons choisi l'exemple décrit ci-après (Ex1) pour illustrer la programmation par *HyperBoîtes* et en particulier le passage de paramètres.

La figure VI.11 montre le contenu de chaque boîte de la hiérarchie définissant le programme « Ex1 » suivant :

```
Program Ex1
{
    /* specification part */
    hs1,dpo1,dpo2

    Fct1(A)       /* A   E/S */
    {
        /* specification part */
        dpo3

        /* execution part */
        while(cond) {
            expr1: A = dpo1.opG1.opG2 + sinh(dpo2)
            expr2: dpo2 = dpo1 + 5
        }
        dpo2 = A.opG1 + dpo3
    }

    DPO Fct2(A)
    {
        /* specification part */
        Res
        ...
        /* execution part */
        if (cond)
          expr1: Res = Fct2(A)
        else
          expr2: dpo1 = A
        return Res
    }

    /* execution part */
    if (cond)
      expr1: dpo1 = dpo2 + Fct2(dpo1)
    else
      expr2: Fct1(dpo1)
}
```

Le programmeur définit d'abord une boîte fonction qu'il nomme « Ex1 » ; c'est la boîte racine correspondant au programme principal (*cf.* `Main_Program`). Une fois cette boîte ouverte (*cf.* **`Ex1 Expanded`**), le programmeur va définir, dans l'éditeur d'opérations géométriques, un hyper-espace « **hs1** » et à l'intérieur deux DPO : « **dpo1** » et « **dpo2** ». Les déclarations correspondant aux variables (ici les DPO) vont être automatiquement représentées, par des icônes, dans la partie spécification (paramètres locaux). L'icône prend la couleur du DPO correspondant. L'intérêt de la couleur sera montré ci-après au niveau du passage de paramètres.

Dans ce programme, on veut définir deux fonctions « `Fct1` » et « `Fct2` », le programmeur insère de ce fait les deux boîtes correspondantes. (Les boîtes ne sont pas forcément insérées en même temps.)

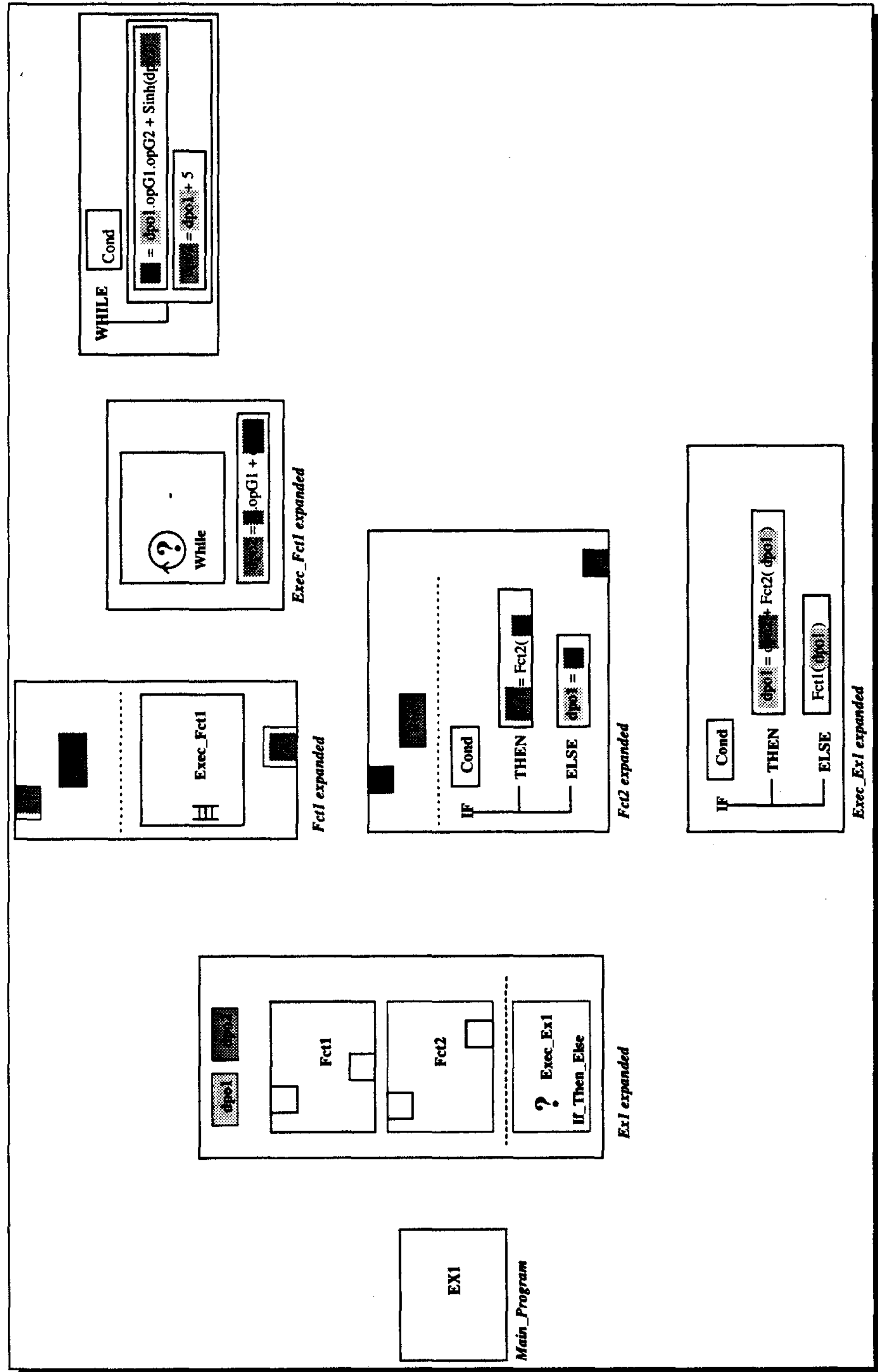


FIG. VI.11 - *Un exemple de programme défini par HyperBoîtes*

Pour spécifier que les fonctions ont des paramètres en entrée et en sortie, le programmeur doit explicitement placer des cellules respectivement à l'entrée et à la sortie de la boîte « Fonction ». Sachant que la définition d'une fonction se fait par l'intermédiaire d'exemples de valeurs (programmation démonstrationnelle), le programmeur déclare un objet (par exemple : A) dans l'hyper-espace et fait la correspondance avec la cellule en entrée. Lors du développement des instructions, il suffira d'un petit coup d'œil pour voir le lien entre les paramètres d'entrées/sorties et leurs références dans les instructions. Ce lien par la couleur permettra plus tard lors de l'appel d'une fonction de spécifier le passage de paramètres par une simple correspondance entre l'argument de la fonction et le DPO effectif (même si c'est un temporaire). La boîte « Fonction » est utilisée comme un composant dans l'éditeur d'expressions; son utilisation est similaire à celle des fonctions prédéfinies (ex: `cos`). La correspondance se fait au niveau de l'éditeur d'opérations géométriques.

Lorsque la même couleur apparaît dans les cellules en entrée et en sortie, le paramètre correspondant est en entrée/sortie (exemple: A dans la fonction `Fct1`).
Si la fonction doit retourner une valeur, la cellule placée en sortie est différente: elle ne contient pas de nom. Elle aura seulement la couleur de l'objet à retourner (exemple: la fonction `Fct2`).

Notons que les variables scalaires sont elles aussi passées par des couleurs, mais ont des cellules distinctes de celles des DPO.

2.2 Analyse et visualisation géométrique d'un programme data-parallèle

Les environnements d'analyse de programmes data-parallèles actuels assistent le programmeur dans le déverminage à travers la visualisation du code et/ou des données pendant l'exécution. Nous pouvons voir l'état d'un tableau avant et après l'exécution d'une instruction, mais pas ce que représente conceptuellement cette instruction. Ces environnements ne peuvent pas nous dire par exemple que tel résultat vient à la suite d'une rotation puis une réplication d'un tableau. Or cette démarche est très intéressante; elle nous permettrait d'une part de vérifier nos propres algorithmes data-parallèles et d'autre part de reprendre un code existant pour comprendre l'algorithmique sous-jacent (voire un code écrit par une tiers personne).

Il faut être par ailleurs conscient de l'impossibilité, même en recourant à l'intelligence artificielle, de retrouver automatiquement les manipulations géométriques codées dans un langage n'utilisant pas explicitement des instructions géométriques. Il est très difficile de conclure automatiquement que le code HPF suivant:

```
for (I = 1:N, J= 1:N)
 res(I,J) = A(J, mod(1+J,N)+1)
endfor
```

applique au tableau A un décalage circulaire d'un pas égal à 2 puis une rotation.

En ce qui nous concerne, nous proposons de visualiser l'algorithme d'un code existant produit par HELPDraw. Lors du développement d'un programme et donc à la génération de code source (exemple : dans HPF), HELPDraw génère aussi des « directives HELPDraw » qui explicitent les manipulations faites. Pour le code précédent par exemple, HELPDraw génèrerait en plus une directive explicitée par « !HELP » :

```
!HELP A.circular_shift(x,2).rotate(x,y)
      for (I = 1:N, J = 1:N)
         res(I, J) = A(J, mod(1+J,N) + 1)
      endfor
```

Par ces directives, nous nous ramenons à la visualisation d'un code écrit dans un langage géométrique.

Lors de la visualisation, HELPDraw analyse chaque directive commençant par « !HELP », puis montre (pas à pas) la conséquence graphique de chaque action ou déclaration. La différence entre la phase de développement et cette phase de visualisation est que dans la première HELPDraw reçoit les ordres du programmeur (par manipulation directe), alors que dans la deuxième il les retrouve dans un code. La partie visuelle par contre reste la même dans les deux cas.

Bibliographie

[AA82] T. Agerwalak and Arvind. Data flow systems: Guest editor's introduction. *IEEE Computer*, 15:10–13, 1982.

[AB89] A.L. Ambler and M.M. Burnett. Influence of visual technology of the evolution of language environments. *IEEE Computer*, October 1989.

[AB90] A.L. Ambler and M.M. Burnett. Visual forms of iteration that preserve single assignment. *Journal of Visual languages and Computing*, 1:159–181, 1990.

[AC90] A.Hough and J. Curry. Perspective views: A technique for enhancing parallel program visualization. In *Proc.of 1990 International Conference on Parallel Processing*, pages II 124–132, August 1990.

[ALS91] Eugene Albert, Joan D. Lukas, and Guy L. Steele Jr. Data parallel computers and the FORALL statement. *Journal of Parallel and Distributed Computing*, 13(2):185–192, October 1991.

[AMS89] W. Appelbe, C. McDowell, and K. Smith. Start/Pat: a Parallel Programming Toolkit. *IEEE Software*, pages 29–38, July 1989.

[AYWC86] T. Ae, M. Yamashita, and H. Matsumoto W.C. Cunha. Visual user interface of a programming system MOPS-2. In *IEEE Computer Society Workshop on Visual languages*, pages 44–53, Dallas, Texas, June 25–27 1986.

[BA94] Margaret M. Burnett and Allen L. Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual languages and Computing*, 4(5):29–60, 1994.

[Bau78] M.A. Bauer. *A basis for the acquisition of procedures*. PhD thesis, Department of Computer Science, University of Toronto, 1978. 310 pages.

[BB93] Thomas Bemmerl and Peter Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environment. *Journal of Parallel and Distributed Computing*, 18(2):118–128, June 1993.

[BB94] M.M. Burnett and M.J. Baker. A classification system for visual programming languages. Technical Report 93-60-14, Departmnet of Computer Science, Oregon State University, Corvallis, OR 97331, 1994. Revised June 16, 1994.

[BBES91] Ingo Barth, Thomas Bräunl, Stefan Engelhardt, and Frank Sembach. PARALLAXIS version 2 user manual. Technical Report 2/91, Fakultät Informatik, Universität Stuttgart, Germany, February 1991.

[BDG+91] A. Beguelin, J.S. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *ACM Supercomputing'91 Proc.*, pages 435–444, November 1991.

[BDLM92a] Akram Benalia, Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HELP for data-parallel scientific programming. In *Supercomputing '92 (poster session)*, Minneapolis, Minnesota, November 1992.

[BDLM92b] Akram Benalia, Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. HelpDraw: An interactive graphical environment for data parallel programming. In *CNI/MAT'92 International Workshop*, Nancy, France, December 1992.

[BDM93] Akram-Djellal Benalia, Jean-Luc Dekeyser, and Philippe Marquet. HelpDraw graphical environment: A step beyond data parallel programming languages. In M.J. Smith G. Salvendy, editor, *Human-Computer Interaction: Software and Hardware Interfaces. Proceedings of the Fifth International Conference on Human-Computer Interaction, (HCI'93), Volume 2*, pages 591–596, Orlando, Florida, August 8-13 1993. Elsevier Science Publishers B.V.

[BDM94] Akram Benalia, Jean-Luc Dekeyser, and Philippe Marquet. A Graphical Environment for HPF Code Development. In *Supercomputing '94 (poster session)*, Washington D.C. (USA), November 1994.

[Ben94a] Akram-Djellal Benalia. Un Environnement Graphique pour le Développement de Codes HPF. Rapport de Recherches ERA-153, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, May 1994.

[Ben94b] Akram-Djellal Benalia. Une interface Graphique pour le Développement de Codes HPF. In *RenPar'6 : 6e Rencontres francophones du parallélisme, présentation sur affiche*, Lyon, June 1994.

[Ben95] Akram-Djellal Benalia. HelpDraw : génération automatique de code HPF. In *Journées SEH 95 (Site Experimental en Hyperparallélisme)*, Paris, 24-26 Janvier 1995.

[BK91] J.L. Bentley and B.W. Kerninghan. A system for algorithm animation. *Computing Systems*, 4(1):5–30, 1991.

[Bla90] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of the IEEE Compcon Spring 1990*, pages 20–24, San Francisco, CA, February 1990. IEEE Society Press.

[BLDA93] H. Brams, M. Lobelle, G. Detroz, and A. April. G++: a graphical language to specify real-time parallel aplications. In *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pages 185–193, Gran Canaria, January 27-29 1993.

[BPP+93] D. Bruschi, P.Lenzi, E. Pozzetti, S. Gobbo, and G. Serazzi. A user-friendly environment for parallel programming. In *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pages 451–456, Gran Canaria, January 27-29 1993.

[Bro88] Marc H. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, 1988.

[Bro91] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proc. of IEEE Workshop on Visual Languages*, pages 4–9, Kopbe, Japan, October 1991. IEEE Computer Society Press, New York.

[CC86] O. Clarisse and S.-K. Chang. VICON: A visual icon manager. In *Visual Languages*, pages 151–190. Plenum Press, New York, 1986.

[CDZ93] W. Citrin, M. Doherty, and B. Zorn. Control constructs in a completely visual imperative language. Technical Report CU-CS-672-93, Dept. of Computer Science, Univ. of Colorado, Boulder, Sep. 1993.

[CDZ94] W. Citrin, M. Doherty, and B. Zorn. Design of a completely visual object-oriented programming language. In M. Burnett, A. Goldberg, and T. Lewis, editors, *Visual Object-Oriented Programming.* Prentice-Hall, New York, 1994. Not published yet.

[Cha87] Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, pages 29–39, January 1987.

[Che93] Doreen Y. Cheng. A survey of parallel programming languages and tools. Research Report RND-93-005, NASA Ames Research Center, Moffett Field, CA, March 1993.

[CTB86] N. Cunniff, R.P. Taylor, and J.B. Black. Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. In *Proc. SIGCHI'86: Human Factors in Computing Systems*, pages 175–182, Boston, MA, April 13–17 1986.

[DCS92] Chyi-Ren Dow, Shi-Kuo Chang, and Mary Lou Soffa. A visualization system for parallelizing programs. In *IEEE Supercomputing'91 Proceedings*, pages 194–203, Mineapolis, November 1992.

[Den75] J.B. Dennis. First version of data flow procedure language. Technical Report MIT/LCS/TM-61, Laboratory for computer Science, MIT, 1975.

[Dig92] Digital Equipment Corporation. *DECmpp System Overview — Manual*, January 1992. Doc. AA-PMAPA-TE.

[DK82] V.L. Davis and R.M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):175–182, 1982. hn.

[DLM94] Jean-Luc Dekeyser, Dominique Lazure, and Philippe Marquet. A geometrical data-parallel language. *ACM Sigplan Notices*, 29(4):31–40, April 1994.

[DSB92] J. Dongarra, D. Sorensen, and O. Brewer. Tools to aid in the design, implementation, and understanding of algorithms for parallel processors. In R. H. Perrott, editor, *Software for Parallel Computers*, chapter 13, pages 196–219. Chapman & Hall, London, 1992.

[EK94] Sherif El-Kassas. *Visual languages: Definitions and applications in system development environments.* PhD thesis, Eindhoven university of technology, April 29th 1994. 161 pages.

[Eng93] Marc Engels. *Design and Programming of Multiprocessors for Real-Time Digital Signal Processing.* PhD thesis, Catholic University of Leuven, Electrotechnical Departement, Afdeling East, Kardinaal 94 — B-3001 Heverlee, Belgium, April 1993.

[ERL92] H. El-Rewini and T.G. Lewis. *Introduction to parallel computing.* Prentice-Hall International Editions, 1992. Chapter 8.

[FLK+91] M. Friedell, M. LaPolla, S. Kochhar, S. Sistare, and J. Juda. Visualizing the behavior of massively parallel programs. In *ACM Supercomputing'91 Proc.*, pages 472–480, November 1991.

[For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Rice University, Houston, TX, May 1993.

[FS93] C. Farhat and H.D. Simon. TOP/DOMDEC: A software tool for mesh partitioning and parallel processing. Technical Report CU-CSSC-93-11, Department of Aerospace Engineering Sciences and Center for Space Structures and controls, University of Colorado at Boulder, Boulder, CO 80309-0429, USA, 1993.

[GA92] G.Viehstaedt and A.L. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing*, 3(3):273–298, 1992.

[GCCWP92] G.-C.Roman, K.C. Cox, C.D. Wilcox, and J.Y. Plun. Pavane: A system for declarative visualisation of concurrent computations. *Journal of Visual languages and Computing*, 3(2):161–193, 1992.

[GGJ+89] V.A. Guarna, J.D. Gannon, D. Jablonowski, A.D. Malony, and Y. Gaur. FAUST: an Integrated Environment for Parallel Programming. *IEEE Software*, 6(4):20–27, July 1989.

[GI85] R.B. Grafton and T. Ichikawa, editors. *IEEE Computer, Special Issue on Visual programming*, pages 6–94, 18, August 1985.

[GKM90] E.P. Glinert, M.E. Kopache, and D.W. McIntyre. Exploring the general-purpose visual alternative. *Journal of Visual languages and Computing*, 1(1), 1990.

[GM89] E.P. Glinert and D.W. McIntyre. The user's view of sunpict, an extensible visual environment for intermediate-scale procedural programming. In *Fourth Israel Conference on Computer Systems and Software Engineering*, pages 49–58, June 5-6 1989.

[GR90] E. J. Golin and S. P. Reiss. The specification of visual language syntax. *Journal of Visual languages and Computing*, 1(2):141–157, 1990.

[GRS91] Cécile Germain-Renaud and Jean-Paul Sansonnet. *Les ordinateurs massivement parallèles.* Armand Colin, 1991. Chapitre 1.

[GT84] E.P. Glinert and S.L. Tanimoto. Pict: An interactive graphical programming environment. *IEEE Computer*, 17:7–25, 1984.

[Hal84] D.C. Halbert. *Programming by Example.* 83 pages, University of California, Berkeley, 1984.

[Har88] D. Harel. On visual formailsms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hil92] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual languages and Computing*, 3(1):69–101, 1992.

[HIY+87] M. Hirakawa, S. Iwata, I. Yoshimoto, M. Tanaka, and T. Ichikawa. HI-VISUAL iconic programmaing. In *Proc. 1987 IEEE Workshop Visual Languages*, pages 305–314, 1987.

[HM94] J. Harvey and J. Morris. NL: A general purpose visual dataflow programming language. Technical report, Departement of Computer Science, University of Tasmania, GPO Box 252C, Hobart, Tasmania, Australia, 1994.

[HMM94] Steven T. Hackstadt, Allen D. Malony, and Bernd Mohr. Scalable performance vizualisation for data-parallel programs. In *Scalable High-Performance Computing Conference*, pages 342–349, Knoxville, Tennessee, May 1994. IEEE Society Press.

[HP91] Sue Utter Horing and C.M. Pancake. Graphical animation of parallel fortran programs. In *ACM Supercomputing'91*, pages 491–500, November 1991.

[IWC+88] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A visual programming environment. *SIGPLAN Notices*, 23(11):176–190, 1988. par env.

[JPD93] Journal of Parallel and Distributed Computing, 18(2), June 1993. Academic Press, Inc.

[KG90] M.E. Kopache and E.P. Glinert. C^2: A mixed textual/graphical environment for C. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, 1990.

[KMT91] K. Kennedy, Kathryn McKinley, and C. Tseng. Interactive Parallel Programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2:329–341, July 1991.

[KS91] Carol Kilpatrick and Karsten Schwan. ChaosMON–Application-Specific monitoring and display of performance information for parallel and distributed systems. *ACM Sigplan Notices*, 26(12), December 1991.

[KS93] Eillen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[KTC+90] D. Kozen, T. Teitelbaum, W. Chen, J. Field, W. Pugh, and B.V. Zanden. ALEX–an alexical programming language. In T. Ichikawa, editor, *Visual Languages and Applications*, pages 147–157. Plenum Press, New York, 1990.

[KvBS92] Dennis Koelma, Richard van Balen, and Arnold Smeulders. SCIL-VP: a multipupose visual programming environment. In *Proc. of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1188–1198, 1992.

[Laz95] Dominique Lazure. *Programmation Géométrique à Parallélisme de Données: Modèle, Langage et Compilation.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, February 1995. (en préparation).

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[Lod82] K.N. Lodding. Iconics- A visual Man-Machine Interface. In *Proc. Nat'l Computer Graphics Assoc*, volume 1, pages 221–233, NCGA, faifax, va, 1982.

[LR90] T.G. Lewis and W.G. Rudd. Architecture of the parallel programming support environment. In *CompCon Spring'90: Thirty-Fifth IEEE Computer Society International Conference*, pages 589–594, 1990.

[LSA93] P. Lyons, C. Simmons, and M. Apperley. Hyperpascal: A visual language to model idea space. In *Proc. 13th New Zealand Computer Society Conf.*, pages 492–508, New Zealand, August 1993.

[LSCA93] M.V. LaPolla, J.L. Sharnowski, B.H.C. Cheng, and K. Anderson. Data parallel program visualisation from formal specifications. *Journal of Parallel and Distributed Computing*, 18(2):252–257, June 1993.

[Mar92] Philippe Marquet. *Langages Explicites à Parallélisme de Données.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, February 1992.

[Mas91] MasPar Computer Corp. *MasPar Parallel Application Language (MPL) — User Guide, Software Version 2.0*, March 1991. Doc. 9302-0100, Rev. A4.

[McI92] D. W. McIntyre. *A Visual Method for Generating Iconic Programming Environments.* PhD thesis, Rensselaer Polytechnic Institute, Troy, N.Y., 1992.

[Mon94] Catherine Mongenet. Data compiling for systems of affine recurrence equations. In *Proceedings of the Franco/British N+N metting on data-parallel languages and compilers for portable [arallel computing.* Universit de Lille 1, April 1994.

[Mye86] Brad A. Myers. Visual Programming, Programming by Example, and program Visualisation; A Taxonomy. In *proc. SIGCHI'86: Human Factors in Computing Systems*, pages 59–66, Boston, MA, 1986. ACM Press.

[Mye88] Brad A. Myers. The State of the Art in Visual Programmin and Program Visualisation. Technical Report CMU-CS-88-114, Carnegie Mellon University Computer Science Department, 1988.

[Mye90a] B. A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, April 1990.

[Mye90b] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual languages and Computing*, 1(1):97–123, 1990.

[Mye92] Brad A. Myers. Demonstrational interfaces: A s tep beyond direct manipulation. *IEEE Computer*, pages 61–73, August 1992.

[MZ94] Gaurav Marwaha and Kang Zhang. Parallel program visualization for a message-passing system. In *Proceedings of 13th Annual IEEE International Phoenix Conference on Computers and Communications*, page 6 pages, 12-15 April 1994.

[Nau60] P. Naur. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[NCA93] V. Natarajan, D. Chiou, and B.S. Ang. Performance visualisation on Monsoon. *Journal of Parallel and Distributed Computing*, 18(2):169–180, June 1993.

[NK91] M.A. Najork and S. Kaplan. The Cube language. In *Proceedings of 91 IEEE workshop on visual Languages*, pages 218–224, Kobe, Japan, October 1991. IEEE Computer Society press Los Alamos.

[PBS93] B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualization. *Journal of Visual languages and Computing*, 4(3), September 1993.

[PGA+91] D. Peace, A. Ghafoor, I. Ahmad, D.L. Andrews, K. Foudil-Bey, T.E. Karpinski, M.A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, January 1991.

[PGH+89] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Paraphrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proc. of 1989 International conference on parallel processing*, pages II–39–II–48, St Charles, Illinois 1989, 1989.

[Pon86] M.-C. Pong. A graphical language for concurrent programming. In *IEEE Computer Society Workshop on Visual languages*, pages 26–33, Dallas, Texas, June 25–27 1986.

[Pri90] Blaine Alexander Price. A framework for the automatic animation of concurrent programs. Master's thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4, December 1990. 110 pages.

[PVG94] G-R Perrin, E. Violard, and S. Genaud. PEI: a theoretical framework for data parallel programming. In *Proceedings of the Franco/British N+N metting on data-parallel languages and compilers for portable [arallel computing.* Universit de Lille 1, April 1994. Publication 94-05 ICPS.

[Rae85] G. Raeder. A survey of current graphical programming techniques. *Computer*, 18(8):11–25, 1985.

[RASW90] J. Rasure, D. Argiro, T. Sauer, and C. Williams. A visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology*, 2:183–199, 1990.

[RC93] G. Roman and K.C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, December 1993.

[Rei84] Steven P. Reiss. Graphical program development with PECAN program development systems. *ACM Sigplan Notices*, 19(2):30–41, 1984.

[Roo94] Jean-Fran ois Roos. *Mise au point d'applications distribu es pour environnement de d veloppement bas sur une technologie objet.* 1281, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, February 1994.

[RW91] J.R. Rasure and C.S. Williams. An integrated data flow visual language and software development environment. *Journal of Visual languages and Computing*, 2(3):217–246, September 1991.

[RW93] Diane T. Rover and Charles T. Wright. Visualizing the peformance of SPMD and data-parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):129–146, June 1993.

[SBKB90] P.A. Suhler, J. Biswas, K.M. Korner, and J.C. Browne. TDFL: a task-level dataflow language. *Journal of Parallel and Distributed Computing*, 9(2):103–115, 1990.

[SBN89] D. Socha, M. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, 1989.

[Sed91] Robert Sedgewick. *Algorithmes en langage C*, chapter 9, Le tri rapide (Quicksort). InterEditions, Paris, 1991.

[SF90] P.D. Stotts and R. Furuta. Browsing parallel process networks. *Journal of Parallel and Distributed Computing*, 9(2):224–235, 1990.

[SG93] Sekhar R. Sarukkai and Dennis Gannon. SIEVE: A performance debugging environment for parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):147–168, June 1993.

[Shn83] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[SK94] Amitabh B. Sinha and Laxmikant V. Kal . A framework for intelligent performance feedback. Technical report, Department of Computer Science, University of Illinois, Urbana, IL 61801, 1994.

[Sny84] Lawrence Snyder. Parallel programming and the Poker programming environment. *Computer*, 17(7):27–36, 1984.

[SP92] J.T. Stasko and C. Patterson. Understanding nad characterizing software visualisation systems. In *Proc. of The IEEE 1992 Workshopon visual languages*, pages 3–10, Seattle, WA,IEEE Computer Society Press, new York, 1992.

[SRGB90] V.Y. Shen, C. Richter, M.L. Graf, and J.A. Brumfield. VERDI: A visual environment for designing distributed systems. *Journal of Parallel and Distributed Computing*, 9(2):128–137, 1990.

[TB86] T.H. Taylor and R.P. Burton. An icon-based graphical editor. *Computer Graphics World*, 9:77–82, 1986.

[Thi90] Thinking Machines Corporation, Cambridge, MA. *Getting Started in CM FORTRAN*, February 1990. Version 5.2-0.6.

[Thi92] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-5 – Technical Summary*, November 1992.

[VBF94] V. Van Dongen, C. Bonello, and C. Freehill. High Performance C, language specification, version 0.8.9. Technical Report CRIM-EPPP-94/04-12, CRIM, Montréal, April 1994.

[VP93] ric Violard and Guy-Ren Perrin. PEI: a single unifying model to design parallel programs. In *PARLE'93*, pages 500–516, 93.

[WCG+91] A. Wagner, S. Chanson, N. Goldstein, J. Jiang, H. Larsen, and H. Sreekantaswamy. TIPS: Transputer-based Interactive Parallelizing system. In P. Welch et al., editor, *Proc. World Transputer User Group Conf.*, pages 212–229, April 1991.

[WR90] C.S. Williams and J.R. Rasure. A visual language for image processing. In *IEEE Workshop on Visual Languages*, pages 86–91, Skokie, Illinois, 4–6 October 1990.

[ZF88] J.E. Ziegler and K.-P. Fahnrich. Direct manipulation. In M. Helander, editor, *Handbook of Human-Computer Interaction*, pages 123–132. Elsevier Science Publishers B.V., 1988. revised 1992.

[ZNQ93] X. Zhang, N.S. Nalluri, and X. Qin. MIN-Graph: A tool for monitoring and visualizing MIN-Based multiprocessor performance. *Journal of Parallel and Distributed Computing*, 18(2):231–241, June 1993.

Liste des tableaux

I.1 Classification des systèmes de développement de programmes parallèles . . . 9

II.1 Classification de langages de programmation visuelle 43

II.2 Classification des systèmes de visualisation de programmes 84

III.1 Les règles utilisées pour les opérations microscopiques 121

III.2 Les règles utilisées pour les opérations macroscopiques 125

IV.1 Les règles utilisées pour la génération de code C-HELP concernant les opérations microscopiques 150

IV.2 Les règles utilisées pour la génération de code C-HELP concernant les opérations macroscopiques 153

V.1 Les règles de réécriture contextuelles pour : $f_{indice(col)}$ 196

V.2 Les règles utilisées pour la génération de code HPF concernant les opérations microscopiques 208

Table des figures

I.1 Dépendance de contrôle 7

I.2 Schedule : un exemple de construction de programme sous Build 11

I.3 L'interface graphique de Parade : la fenêtre principale et une fenêtre de définition d'attributs pour le nœud 4 12

I.4 G++ : un exemple de programme temps-réel 14

I.5 Une configuration créée en utilisant Visputer 15

I.6 L'interface graphique VISUAL.PEI : exemple de multiplication de matrices . . 18

I.7 L'éditeur graphique de GRAPE 20

I.8 Le mode trace de l'interface graphique de HeNCE 23

I.9 Un exemple de ré-exécution d'un programme Schedule sous Trace facility . . 24

I.10 Animation d'un programme sous Visputer 25

I.11 Prism : un exemple de visualisation de surface 27

I.12 Une partie de l'interface graphique de Parafrase-2 29

II.1 Cantata : un espace de travail contenant un exemple d'application de traitement d'images 48

II.2 Cantata : un exemple de If_Then_Else et utilisation des espaces de travail hiérarchiques 49

II.3 Un exemple de programme VIPR « Hello World » (à gauche) et son équivalent en C (à droite) 52

II.4 VIPR : représentation d'une instruction conditionnelle 53

II.5 VIPR : Appel de procédure et retour 54

II.6 HyperPascal : un icône de sous-programme 56

II.7 HyperPascal : quatre icônes de sous-programmes assemblés dans un arbre de portée 57

II.8 HyperPascal : exemple de représentation d'un arbre d'actions 58

II.9 L'interface Peridot lors du développement d'un exemple de procédure 62

II.10 Peridot : un exemple d'inférence puis génération d'une liste d'itèmes 63

II.11 Peridot : l'aspect démonstrationnel 64

II.12 Pigsty : Un exemple de programme (Problème des cinq philosophes) 68

II.13 NL : le nœud If « Quadratique » 69

II.14 NL :(à gauche) Le nœud If « Quadratique » après expansion. (à droite) Le nœud If « Racines » après expansion 70

II.15 Forms/3 : un exemple de forme avec des cellules et une matrice 72

II.16 Forms/3 : définition d'une forme pour le produit ligne-colonne et d'une autre forme pour la multiplication de matrices 73

II.17 CUBE : définition du prédicat Factoriel 74

II.18 Un exemple de visualisation de codes : (A) statique, (B) dynamique 78

II.19 Un exemple de visualisation de données 79

II.20 Un exemple de visualisation d'algorithmes 79

II.21 Représentation directe 80

II.22 Représentation structurelle 82

II.23 Représentation synthétisée 83

III.1 Exemples de DPO dans un Hyper-Espace 3-D 94

III.2 Exemple d'hiérarchie de segments conformes 95

III.3 Exemples d'application du constructeur on 96

III.4 Application des opérations géométriques 98

III.5 Exemple d'opérations géométriques 99

III.6 Les trois formes de triangles acceptées dans un plan donné 102

III.7 Un exemple de conception selon le modèle HELP : M(i,j) = V(i)+V(j) 103

III.8 La chaîne de développement d'une instruction data-parallèle 105

III.9 Un exemple de définition d'un hyper-espace 106

III.10 Un exemple de déclaration de DPO réguliers 108

III.11 Les icônes permettant de choisir la forme du triangle 108

III.12 Exemples de DPO définis dans un hyper-espace 109

III.13 La liste des DPO : une partie de l'éditeur d'opérations géométriques 110

III.14 La partie principale de l'éditeur d'opérations géométriques 111

III.15 Application d'une opération géométrique à travers un menu 112

III.16 Application d'une opération d'expansion 113

III.17 Application d'une opération de réduction 114

III.18 La liste des DPO temporaires 115

III.19 Un exemple d'application d'une opération microscopique 116

III.20 Un exemple de développement d'une expression sous HELPDraw 117

III.21 Représentation des segments d'une expression 118

III.22 Définition et sélection de domaine 119

III.23 Gestion de priorités des opérateurs 120

III.24 L'éditeur de construction d'expressions 127

III.25 L'arbre d'un exemple d'affectation conditionnelle : « $res = (dpo1 - dpo2 <= tan(dpo)$ && $dpo3)$? $dpo1/dpo3$: 10 » 129

III.26 Exemple d'une liste d'expressions encapsulées 130

III.27 Un exemple d'icône-expression après correspondance 130

III.28 Génération de code (ici C-HELP) 132

III.29 L'éditeur de blocs d'instructions 133

III.30 La représentation graphique de la construction « On » 134

III.31 La représentation graphique de la construction « If_Then_Else » 134

III.32 La représentation graphique de la construction « Where_Elsewhere » 135

III.33 La représentation graphique de la construction « While » 136

III.34 La représentation graphique de la construction « Repeat » 136

IV.1 Exemples de projections d'un hyper-espace en C-HELP 141

IV.2 Un exemple de dimension secondaire 142

IV.3 Un exemple de déclarations d'objets en C-HELP 142

IV.4 Opérations de déplacements en C-HELP 144

IV.5 Opérations de réplications en C-HELP 145

IV.6 Un exemple d'extraction en C-HELP 145

IV.7 Déclaration de triangle en C-HELP 148

IV.8 Déplacement vers la position (x1, y1, z1) 152

IV.9 Exemples de rotations 154

IV.10 Rotation d'un triangle 155

IV.11 L'opération de réplication « **expand(x)** » de HELPDraw 156

IV.12 Exemples d'extractions de sous-objet 157

IV.13 Multiplication de matrices sur une grille 2-D 158

IV.14 Les valeurs c_{ij} de la matrice que nous souhaitons obtenir à la fin de la multiplication de A par B ($C = A * B$) 159

IV.15 Rangement des données après la première étape 159

IV.16 A chaque étape, décalage de A vers la gauche et de B vers le haut 160

V.1 Les différents niveaux de placement des données en HPF 165

V.2 Alignements de projection 168

V.3 Alignement de déplacement 168

V.4 La nécessité d'une fonction de description f pour représenter une suite d'opérations macroscopiques appliquées à un DPO. 175

V.5 Exemples de correspondance entre un élément (I,J,K) d'un temporaire obtenu à la suite d'opérations géométriques et sa source f(I,J,K) dans le DPO source . 175

V.6 Un exemple de forme géométrique: le triangle 178

V.7 Changement d'indices 181

V.8 Les étapes de calcul de f 183

V.9 Un exemple de rotation d'un triangle 187

V.10 Un exemple de réplication : l'élément (i, j) du DPO source se retrouve aux éléments $(i + \alpha * Slen1, j)$ de l'objet résultat 189

V.11 Exemples d'opérations de décalages 191

V.12 Un exemple d'opérande 198

V.13 Exemple d'interaction de DPO dans un domaine contraint 205

V.14 Inversion: diffusion des lignes et colonnes 210

V.15 Rotation (diagonale vers colonne), puis diffusion 211

VI.1 A∗B sur une grille 3D : A répliquée selon l'axe 'z' et B (après rotation) répliquée selon l'axe 'x' 216

VI.2 Définition de l'hyper-espace 3-D 218

VI.3 Définition du DPO A sur le plan [z=1] 219

VI.4 Réplication du DPO A le long de l'axe 'z' 220

VI.5 Le temporaire résultant des opérations de rotation et de réplication appliquées au DPO B 221

VI.6 Réalisation de la multiplication par l'intermédiaire d'une expression encapsulée 222

VI.7 Le résultat du produit de matrices 223

VI.8 Le schéma global de développement d'un programme HELPDraw 229

VI.9 La représentation graphique de la construction « Seq » 231

VI.10 La représentation graphique d'une structure « Fonction » 232

VI.11 Un exemple de programme défini par HyperBoîtes 234