TABLE DES MATIERES

Thèse de doctorat

[arrêté du 30 mars 1992]

Université Paris 8

Spécialité :

Informatique – Intelligence Artificielle

Présentée par :

Damien Ploix

Pour obtenir le titre de Docteur de l'Université Paris 8

Sujet de la thèse :

Elaboration, Réalisation et Evaluation d'un Environnement de Programmation Analogique

Soutenue le 4 janvier 1999 devant le jury composé de :

MM.	Patrick GREUSSAY	*président*
	Harald WERTZ	*directeur*
	Robert FRENCH	*rapporteur*
	Francis ROUSSEAUX	*rapporteur*
	Tristan CAZENAVE	
	Daniel GOOSSENS	
	Jean-Philippe VIENNE	

Je remercie Monsieur Patrick Greussay d'avoir accepté de présider le jury de cette thèse.

Je remercie vivement Messieurs Robert French et Francis Rousseaux d'avoir accepté de participer à mon jury de thèse et de rapporter sur mon travail.

Je remercie particulièrement Monsieur Harald Wertz, qui suit mon travail depuis ma maîtrise. Tout au long de ma recherche sa patience, ses conseils et ses encouragements m'ont été d'un support constant.

Je remercie Messieurs Tristan Cazenave, Daniel Goossens et Jean-Philippe Vienne de s'être intéressés à mon travail et d'avoir accepté de participer à mon jury.

Ce travail a été élaboré au Laboratoire d'Intelligence Artificielle de l'Université Paris 8 dont l'atmosphère rend la conception de méthodes et de logiciels avancés un véritable plaisir. Que tous ses membres en soient ici remerciés.

Je remercie Madame Françoise Balmas qui a, à travers de nombreuses discussions, grandement contribué à l'achèvement de mes recherches.

La version préliminaire de cette thèse à été relue avec attention par Maïté, je lui en suis grandement reconnaissant.

Résumé

Cette thèse présente une méthode originale de création et d'utilisation de représentations analogiques de programmes, notre environnement de programmation *Zeugma*, qui implémente cette méthode et son évaluation par la présentation de représentations analogiques de programmes.

Le système Zeugma permet à l'utilisateur de construire ses *propres* représentations de programmes, utilisant les analogies ou les métaphores qui lui sont familières, de les expérimenter sur des programmes qu'il a écrits ou qui lui sont inconnus, puis de les utiliser comme nouvelles composantes d'un environnement de programmation.

Notre méthode de construction de représentations analogiques de programmes considère la mise en relation de caractéristiques des programmes, décrivant des aspects de leur composition ou des aspects de leur comportement pendant l'exécution, avec des caractéristiques de telles représentations. Ainsi, des particularités syntaxiques, les flots de contrôle ou de données, les données manipulées ou le code exécuté peuvent determiner des aspects tels que la forme, le placement, la taille, ou encore la couleur d'une représentation graphique analogique. La sélection des caractéristiques utilisées s'effectuera en fonction du choix de son *utilisation* : par exemple la découverte de programmes inconnus, la recherche de parties nécessitant une optimisation ou encore l'animation d'algorithmes. Nous présenterons des exemples de représentations analogiques illustrant ces trois utilisations.

Zeugma est l'unique système de visualisation de programmes qui implémente une méthode de création de représentations analogiques permettant son utilisation autant comme outil de visualisation d'algorithmes et de programmes que comme environnement de programmation intégrant des représentations graphiques.

Enfin, nous donnerons le listing complet du système Zeugma, les définitions complètes des représentations analogiques présentées et le code source des modifications que nous avons effectuées sur le langage Xbvl afin d'implémenter notre système.

Table des matières

Introduction 2
I.2 METHODE DE CONSTRUCTION DE REPRESENTATIONS ANALOGIQUES DE PROGRAMMES 3
I.2.1 Le choix de l'utilisation de la représentation *5*
I.2.2 Le domaine de la représentation *7*
I.2.3 L'établissement des liens analogiques *8*
I.3 SCENARIO D'UTILISATION DE ZEUGMA 9
I.3.1 Présentation du système *9*
I.3.2 Construction d'une représentation analogique *10*
I.3.2.1 Choix des aspects de programmes 12
I.3.2.2 Définition d'aspect des programmes 13
I.3.2.3 Spécification du schéma générateur de représentations 14
I.3.2.4 Attachements dynamiques 16
I.3.2.5 Conclusion : propriétés de l'ORS une-fonction 17
I.3.3 Observation de la représentation analogique *18*
I.3.3.1 Consultation des liens 19
I.3.3.2 Réduction de la complexité 20
I.3.3.3 Contrôle des animations 22
I.3.3.4 Navigation analogique dans un historique 22
I.3.3.5 Impact de Zeugma 23
I.4 PLAN DE LECTURE 23
II Analogie, Métaphore et Visualisation de Programmes 26
II.1 PROLEGOMENES 26
II.1.1 Question de complexité *26*
II.1.1.1 En informatique 26
II.1.1.2 Dans l'art 28
II.1.1.3 Eléments de réponse 31
II.1.2 Représentation analogique de programmes *32*
II.1.2.1 Précision sur les termes 32
II.1.2.2 Critiques sur la visualisation de programmes et nécessité d'un modèle cognitif 34
II.2 METHODE DE CONSTRUCTION DES REPRESENTATIONS ANALOGIQUES DE PROGRAMMES 38
II.2.1 Introduction *38*
II.2.1.1 Modèle pour la compréhension des analogies 38
II.2.1.2 Modèle pour la génération d'analogie 43
II.2.2 Présentation de la méthode de construction de représentations analogiques de programmes *47*
II.2.2.1 Introduction : Précisions sur les domaines d'origine et cible 47
II.2.2.2 Présentation de la méthode 48
II.2.2.3 Le choix sémantique 49
II.2.2.4 Choix des aspects des programmes pris en compte 51
II.2.2.5 Choix de la représentation graphique 52
II.2.2.6 Spécification de liens analogiques entre points de vue sur les programmes et représentations graphiques 53
II.3 EXEMPLES DE REPRESENTATIONS ANALOGIQUES DE PROGRAMMES 55
II.3.1 Introduction *55*
II.3.2 Les programmes comme des plans de cités *55*
II.3.2.1 Motivation de l'analogie 56
II.3.2.2 Vision globale d'une cité 59
II.3.2.3 Les différents quartiers 61
II.3.2.4 Visite guidée de maisons 67
II.3.2.5 Un personnage se déplace et la cité s'anime 71
II.3.2.6 Conclusion 72
II.3.3 Les programmes comme des araignées en mouvement sur une toile *73*
II.3.3.1 Motivation de l'analogie 73
II.3.3.2 Présentation de la colonie d'araignées 76
II.3.3.3 Animation de la représentation analogique 80
II.3.3.4 Conclusion 85
II.3.4 Animation d'algorithmes avec Zeugma *86*
II.3.4.1 Animation analogique d'algorithmes 86
II.3.4.2 Description de la représentation analogique 87
II.3.4.3 Application au tri rapide 89
II.3.4.4 Application au tri par fusion 93

II.3.4.5 Application au tri par insertion 97
II.3.4.6 Conclusion 101
III Zeugma 104
III.1 ELEMENTS CONSTITUTIFS DU SYSTEME ZEUGMA 104
III.1.1 Introduction *104*
III.1.1.1 Problématiques 104
III.1.1.2 Choix techniques 106
III.1.2 Eléments constitutifs de Zeugma *110*
III.1.2.1 Les liens entre programmes Lisp et représentations analogiques 110
III.1.2.2 Aspects des programmes 111
III.1.2.3 Objets graphiques structurés 122
III.1.2.4 Liens programmes – représentations : caractéristiques des ORS 125
III.2 ZEUGMA COMME INTERFACE DE CREATION D'UNE REPRESENTATION ANALOGIQUE 127
III.2.1 Une interface pour la création de représentations analogiques de programmes *127*
III.2.1.1 Etapes de la construction d'une représentation analogique 127
III.2.1.2 Choix des domaines d'application de l'analogie 128
III.2.1.3 Les Objets de Relation Structurelle 132
III.2.1.4 Spécification du parcours des aspects des programmes 137
III.2.1.5 Spécification des attachements liés à l'exécution des programmes 144
III.2.1.6 Données sur les parcours 148
III.2.2 Inspection d'une analogie *153*
III.2.2.1 Navigation dans une représentation graphique 155
III.2.2.2 Informations sur les analogies et réduction dynamique de la granularité 156
III.2.2.3 Suivi et contrôle de la visualisation du comportement des programmes 160
IV Comparaison avec d'autres systèmes 169
IV.1 VISUALISATION DE PROGRAMMES 169
IV.1.1 Problématique des systèmes de visualisation de programmes *169*
IV.1.2 Le système TPM *170*
IV.1.2.1 Problématique de TPM 170
IV.1.2.2 Exemple d'interaction avec TPM 173
IV.1.2.3 Critique de TPM et comparaison avec Zeugma 175
IV.1.3 De Trip à IMAGE *176*
IV.1.3.1 Trip : des données abstraites aux données graphiques 177
IV.1.3.2 Traduction inverse : des données graphiques aux données abstraites 180
IV.1.3.3 Critique de TRIP2 et comparaison avec Zeugma 181
IV.2 VISUALISATION D'ALGORITHME 182
IV.2.1 Problématique des systèmes de visualisation d'algorithmes *182*
IV.2.2 Le système BALSA-II *183*
IV.2.2.1 La notion d' « Evénements Intéressants » 183
IV.2.2.2 Présentation de Balsa-II 185
IV.2.3 Le système Pavane *188*
IV.2.3.1 La notion de « Visualisation Déclarée » 188
IV.2.3.2 Présentation de PAVANE 189
IV.2.4 Critique de Balsa-II et de PAVANE et comparaison avec Zeugma *194*
IV.3 ENVIRONNEMENTS DE PROGRAMMATION 196
IV.3.1 Problématique des environnements de programmation *196*
IV.3.2 Le système FIELD *196*
IV.3.2.1 Principes fondateurs de FIELD 196
IV.3.2.2 Présentation de FIELD 197
IV.3.2.3 Critique de FIELD et comparaison avec Zeugma 200
IV.3.3 Le système Zstep95 ..... *201*
IV.3.3.1 Principes fondateurs de ZStep95 201
IV.3.3.2 Présentation de Zstep95 202
IV.3.3.3 Critique de Zstep95 et comparaison avec Zeugma 204
IV.4 CONCLUSION : SITUATION DE ZEUGMA DANS LE DOMAINE DE LA VISUALISATION DE PROGRAMMES 205
V Conclusion et perspectives 209
V.1 NOS PRINCIPALES CONTRIBUTIONS 209
V.1.1 Représentations d'aspects de programmes *209*
V.1.2 Représentations analogiques de programmes *210*
V.1.2.1 Liens analogiques entre programmes et représentations 210
V.1.2.2 Perceptions analogiques 211
V.1.3 Zeugma *211*
V.1.4 Zeugma comme outil d'expérimentation des représentations *212*

V.2 FAIBLESSES ET EXTENSIONS DE NOTRE SYSTEME 213
V.2.1 Validation de Zeugma par son utilisation *214*
V.2.2 Limitation de la taille des programmes représentés *215*
V.2.3 Application de la méthode à la représentation de données *216*
V.2.4 Description des représentations *217*
V.2.5 Unidirectionnalité du système *218*
Bibliographie 220
Annexe I Code source de Zeugma 235
ANNEXE I.1 FICHIER DE CHARGEMENT DU SYSTEME : ZEUGMA.VLISP 235
ANNEXE I.2 DEFINITION DES DONNEES GLOBALES : ZDEFS.VLISP 235
ANNEXE I.3 DEFINITION DES MENUS : ZMENUS.VLISP 237
ANNEXE I.4 MANIPULATION DES REPRESENTATIONS INTERNES : ZDATA.VLISP 239
ANNEXE I.5 DEFINITION DES ASPECTS PREDEFINIS : ZPRC-DEF.VLISP 240
ANNEXE I.6 DEFINITION DES ANALYSES LIEES AUX POINTS DE VUES : ZPRC-ANALYSES.VLISP 248
ANNEXE I.7 DEFINITIONS DES ANALYSES LIEES AUX POINTS DE VUES (2) : ZANALYSES.VLISP 250
ANNEXE I.8 GENERATION DES ANALOGIES : ZPRC-COMPILE.VLISP 260
ANNEXE I.9 GENERATION DES ANALOGIES (PARTIE GRAPHIQUE) : ZGRAPHIQUES.VLISP 265
ANNEXE I.10 DEFINITION DES ORS : ZORS.VLISP 272
ANNEXE I.11 INFORMATIONS SUR LES FONCTIONS : ZFUNCTION-INFO.VLISP 279
ANNEXE I.12 EDITEUR DE REPRESENTATION : ZMETA-EDITOR.VLISP 290
ANNEXE I.13 ANIMATION DES REPRESENTATIONS : ZIN-OUT.VLISP 304
ANNEXE I.14 CREATION DE L'HISTORIQUE : ZDUMP.VLISP 309
ANNEXE I.15 GESTION DE L'HISTORIQUE : ZSUIVIT.VLISP 316
ANNEXE I.16 INTEGRATION DES ANALOGIES (1) : ZFILE.VLISP 318
ANNEXE I.17 INTEGRATION DES ANALOGIES (2) : ZUSER.VLISP 322
ANNEXE I.18 AIDE EN LIGNE DE ZEUGMA : ZHELP.VLISP 325
Annexe II Fichiers source des définitions des analogies 327
ANNEXE II.1 DEFINITIONS DE L'ANALOGIE « DES PROGRAMMES COMME DES VILLES » 327
ANNEXE II.2 DEFINITIONS DE L'ANALOGIE « LES PROGRAMMES COMME DES ARAIGNEES EN MOUVEMENT SUR UNE TOILE » 339
ANNEXE II.3 DEFINITIONS DE L'ANIMATIONS D'ALGORITHMES DE TRIS 343
Annexe III Modifications de Xbvl 345
ANNEXE III.1 MODIFICATION DE LA LIBRAIRIE MESA-GL : ADDON_DLIST.C 345
ANNEXE III.2 CREATION DE WIDGETS MESA-GL : GLWIDGET.C 346
ANNEXE III.3 GESTION DE LA MEMOIRE ET DES LISTES : GLMEMORY.C 358
ANNEXE III.4 INTERFACE AVEC MESA-GL : MESALISTES.C 362
ANNEXE III.5 FONCTIONNALITES (1) : GLFUNCS.C 365
ANNEXE III.6 FONCTIONNALITES (2) : GLLIGHTS.C 377
ANNEXE III.7 FONCTIONNALITES (3) : GLFONTS.C 378
ANNEXE III.8 FONCTIONNALITES (4) : GLTEXTURE.C 380
ANNEXE III.9 FONCTIONNALITES (5) : GLUFUNCS.C 384
ANNEXE III.10 FONCTIONNALITES (6) : GLIMAGES.C 389
ANNEXE III.11 FONCTIONS D'INTERFACE : GLLIB.C 396
ANNEXE III.12 LIBRAIRIE DE CONSTRUCTION DE MENUS : X-MENUS.VLISP 398
ANNEXE III.13 LIBRAIRIE DE NAVIGATION GRAPHIQUE : GL-DRIVE.VLISP 401

Table des Figures

Chapitre I 1
FIGURE 1.1 STRUCTURE GENERALE DE ZEUGMA 10
FIGURE 1.2 EXEMPLES DE LA REPRESENTATION DES PROGRAMMES PAR DES ARAIGNEES 11
FIGURE 1.3 LISTE DES ASPECTS PREDEFINIS DANS ZEUGMA 12
FIGURE 1.4 EXTRAIT DE LA DEFINITION DU *FLOT DE CONTROLE* COMME UN ASPECT DES PROGRAMMES 13
FIGURE 1.5 CARACTERISTIQUES DU FLOT DE CONTROLE 13
FIGURE 1.6 SCHEMA GENERATEUR DES ARAIGNEES SUR UNE TOILE 14
FIGURE 1.7 ATTACHEMENTS A DES ELEMENTS DU PROGRAMME 16
FIGURE 1.8 ATTACHEMENTS COMME PROPRIETES D'UN ORS 17
FIGURE 1.9 PROPRIETES DE L'ORS *UNE-FONCTION* 18
FIGURE 1.10 INFORMATIONS SUR UN ELEMENT DE LA REPRESENTATION 19
FIGURE 1.11 REPRESENTATION DE LA COMPOSITION D'UNE FONCTION PAR UNE MAISON 21
FIGURE 1.12 STRUCTURE D'UNE REPRESENTATION 21
FIGURE 1.13 MAISON REDUITE 21
FIGURE 1.14 HISTORIQUE D'UNE ANIMATION ANALOGIQUE 23
Chapitre II 25
FIGURE 1.1 « THE FILE ROOM » ANTONIO MUNTADAS, 1994 26
FIGURE 1.2 IMPLEMENTATION DU QUICKSORT EN PROLOG 27
FIGURE 1.3 IMPLEMENTATION DU QUICKSORT EN C 28
FIGURE 1.4 *SENS* D'UN OBJET OU D'UNE IMAGE 29
FIGURE 1.5 RESUME DE L'EVOLUTION DES TECHNIQUES D'INTERACTION [MYERS96, PAGE 795] . 34
FIGURE 2.1 ANALOGIE ENTRE CIRCULATION D'EAU ET DE CHALEUR, [GENTNER89, PAGE 202] ADAPTE DE [BUCKLEY79, PAGES 12 – 25]. 40
FIGURE 2.2 *ESPACE DES SIMILITUDES* : CLASSIFICATION DES SIMILITUDES [GENTNER89, PAGE 207]. 40
FIGURE 2.3 REPRESENTATION DE L'EAU ET DE LA TEMPERATURE DONNEE A SME 41
FIGURE 2.4 REPRESENTATION ANALOGIQUE DE LISTES LISP 42
FIGURE 2.5 PROBLEME 70 ET 71 DE BONGARD [BONGARD70] 44
FIGURE 2.6 REPONSES DE COPYCAT AUX DEUX QUESTIONS 46
FIGURE 3.1 VUES DE L'INTERIEUR DE LA MAISON SHRÖDER 57
FIGURE 3.2 HI-VISUAL [KADO92] ET FSN™ 57
FIGURE 3.3 VISIONS GLOBALES DE DEUX CITEES 60
FIGURE 3.4 ZOOM SUR DES QUARTIERS 64
FIGURE 3.5 EXEMPLES DE MAISONS 66
TABLE 3.1 CORRESPONDANCE INSTRUCTION / COULEUR - POSITION POUR LES TUILES DES MAISONS 67
FIGURE 3.6 DEPLACEMENT D'UN PERSONNAGE 70
FIGURE 3.7 UNE *ARGIOPE BRUENNICHI* [JONES83, P. 267] 74
FIGURE 3.8 DESCRIPTION DES TOILES CONSTRUITES PAR LES ARAIGNEES ORBITELES [JONES83, P.237] 75
FIGURE 3.9 UNE COLONIE D'ARAIGNEES SUR UNE TOILE 78
FIGURE 3.10 ZOOM SUR UNE ARAIGNEE DE LA FIGURE 3.9 79
FIGURE 3.11 VISUALISATION GRAPHIQUE DU DEROULEMENT DE LA RESOLUTION DES TOURS DE HANOI 81
FIGURE 3.12 ETAT DE LA REPRESENTATION ANALOGIQUE AUX DEUX ETAPES DU PROGRAMME HANOI VISUALISE DANS LA FIGURE 3.11 81
FIGURE 3.13 ZOOMS SUR UNE ARAIGNEE A LA PREMIERE ETAPE 82
FIGURE 3.14 ZOOMS SUR UNE ARAIGNEE A LA SECONDE ETAPE (VUE SOUS DEUX ANGLES DIFFERENTS) 82
FIGURE 3.15 DETECTION VISUELLE D'UN GROUPE DE FONCTIONS 84
FIGURE 3.16 DEUX VUES DU TRI PAR FUSION 87
FIGURE 3.17 SCHEMA EXPLICATIF DE L'ANIMATION ANALOGIQUE 87
FIGURE 3.18 VUES SUCCESSIVES DE L'ANIMATION DE LA COPIE D'UNE LISTE DE QUATRE NOMBRES 88
FIGURE 3.19 CODE SOURCE DE LA COPIE D'UNE LISTE 88
FIGURE 3.20 CODE SOURCE DE L'IMPLEMENTATION DU TRI RAPIDE 89
FIGURE 3.21 FILM D'UNE PARTIE DU TRI RAPIDE (DEBUT) 90

FIGURE 3.21 IMAGE FINALE DU FILM D'UNE PARTIE DU TRI RAPIDE 91
FIGURE 3.22 DEUX VUES DE L'IMAGE FINALE DU TRI RAPIDE 92
FIGURE 3.23 CODE SOURCE DU TRI PAR FUSION 93
FIGURE 3.24 PARTIE DU FILM DU TRI PAR FUSION (DEBUT) 93
FIGURE 3.25 SUITE DU FILM DU TRI PAR FUSION 94
FIGURE 3.25 FIN DU FILM DU TRI PAR FUSION 95
FIGURE 3.26 CODE SOURCE DU TRI PAR INSERTION 97
FIGURE 3.27 FILM DU TRI PAR INSERTION 97
Chapitre III 103
FIGURE 1.1 INTERFACE DE XBVL 107
FIGURE 1.2 SCHEMA GENERAL DES REPRESENTATIONS ANALOGIQUES DE PROGRAMMES 111
FIGURE 1.3 ZOOM SUR UN GRAPHE SELON LA METHODE DU FISHEYE [SAKAR93] ET SELON LA METHODE FRACTALE [KOIKE95A] 113
FIGURE 1.4 DEUX FLUX DE CONTROLE D'UN PROGRAMME 116
FIGURE 1.5 FLOT DE DONNEES D'UN PROGRAMME 116
FIGURE 1.6 DEUX REPRESENTATIONS D'UNE FONCTION LISP 116
FIGURE 1.7 DEFINITION DU FLOT DE CONTROLE COMME ASPECT DES PROGRAMMES. 118
FIGURE 1.8 EXEMPLES DE SCHEMES DE GENERATIONS DE REPRESENTATIONS ANALOGIQUES 119
FIGURE 2.1 INTERFACE PRINCIPALE DU SYSTEME ZEUGMA 127
FIGURE 2.2 CREATION D'UNE NOUVELLE REPRESENTATION ANALOGIQUE 129
FIGURE 2.3 FENETRES GENEREES AUTOMATIQUEMENT PAR ZEUGMA 131
FIGURE 2.4 FENETRE DE SPECIFICATION DES ORS RACINE 132
FIGURE 2.5 FENETRE DE SPECIFICATION DES ORS 134
FIGURE 2.6 ACTIONS SUR UN OBJET GRAPHIQUE AU COURS DE L'EXECUTION DES PROGRAMMES. 136
FIGURE 2.7 SPECIFICATION DES ATTACHEMENTS AUTOMATIQUES ENTRE UN ORS ET LES PROGRAMMES. 136
FIGURE 2.8 LISTE DES ORS GENERES A PARTIR DE L'ORS TRAVERSE-TYPES 138
FIGURE 2.9 PROPRIETES DU LIEN ENTRE L'ORS TRAVERSE-TYPES ET L'ORS UNE-FONCTION 138
FIGURE 2.10 SPECIFICATIONS DE LA TRANSLATION EFFECTUEE LORS DE LA TRANSITION ENTRE LES DEUX ORS 138
FIGURE 2.11 : TYPES DE PARCOURS ACTUELLEMENT DEFINIS DANS ZEUGMA 141
FIGURE 2.12 CONDITIONS DES PARCOURS LIES A LA COMPOSITION DES PROGRAMMES. 141
FIGURE 2.13 CONDITIONS LIEES A L'ETUDE DU COMPORTEMENT DES PROGRAMMES 143
FIGURE 2.14 INFORMATION GENERALE DES ATTACHEMENTS SUR UN PROGRAMME 143
FIGURE 2.15 : RESULTAT DES ANALYSES DE NUMERATION SUR LE PROGRAMME *HANOI* 146
FIGURE 2.16 ATTACHEMENTS SUR UNE FONCTION 146
FIGURE 2.17 ATTACHEMENT A UNE EXPRESSION DU PROGRAMME 146
FIGURE 2.18 ATTACHEMENTS SUR UNE VARIABLE 147
TABLE 2.1 INDICATEURS RELATIFS A LA PROGRESSION DANS LA GENERATION D'UNE ANALOGIE 149
TABLE 2.2 INDICATEURS RELATIFS A LA GENERATION D'UNE ANALOGIE (FIN) 150
EQUATION 2.1 : CALCUL DE LA POSITION INITIALE DES ARAIGNEES 151
TABLE 2.3 INDICATEURS RELATIFS A L'OBSERVATION DU COMPORTEMENT D'UN PROGRAMME 152
TABLE 2.4 INDICATEURS ET FONCTIONS LIES A LA POSITION DES OBJETS GRAPHIQUES 152
FIGURE 2.19 SOURCE DE LA FONCTION LISP CALCULANT LE DEPLACEMENT DES ARAIGNEES 153
FIGURE 2.20 FENETRE DE CONTROLE DE LA NAVIGATION GRAPHIQUE 154
FIGURE 2.21 FENETRE DE CONTROLE DES MODES D'AFFICHAGE GRAPHIQUE 155
FIGURE 2.22 ZOOMS VERS UNE MAISON 156
FIGURE 2.23 INFORMATIONS SUR UNE REPRESENTATION 158
FIGURE 2.24 : REDUCTION DE LA COMPLEXITE D'UNE REPRESENTATION GRAPHIQUE 160
FIGURE 2.25 CONTROLE DE L'EXECUTION PAS A PAS 161
FIGURE 2.26 INDICATION DE L'EXECUTION D'UNE EXPRESSION 161
FIGURE 2.27 INDICATION DE L'EXECUTION D'UNE EXPRESSION POSSEDANT UN ATTACHEMENT 161
FIGURE 2.28 : VUES MULTIPLES DE L'EVOLUTION DE L'EXECUTION 162
FIGURE 2.29 INTERFACE DE CONTROLE DE LA VISUALISATION DE L'HISTORIQUE D'UNE EXECUTION 163
FIGURE 2.30 ZOOMS SUR L'ARAIGNEE REPRESENTANT LA FONCTION *DO-HANOI* A LA PREMIERE ET LA TROISIEME ETAPE DE L'EXECUTION 165
FIGURE 2.31 VISUALISATION DE L'HISTORIQUE DE L'EXECUTION 166

FIGURE 2.32 SUITE DE LA VISUALISATION DE L'HISTORIQUE DE L'EXECUTION DU PROGRAMME. 167
Chapitre IV 168
FIGURE 1.1 ELEMENTS GRAPHIQUES DE TPM 171
FIGURE 1.2 PROGRAMME PROLOG VISUALISE DANS LA FIGURE 1.3 172
FIGURE 1.3 INTERFACE DU SYSTEME TPM 173
FIGURE 1.4 NAVIGATION DANS UN DIAGRAMME AORTA AVEC TPM 174
FIGURE 1.6 ABSTRACTION SUR UN ARBRE 175
FIGURE 1.5 COMPRESSION D'UN ARBRE 175
FIGURE 1.7 REPRESENTATION DE FACTORIEL PAR CUBE [NAJORK93] 176
FIGURE 1.8 SCHEMA GENERAL DE LA METHODE TRIP [TAKAHASHI95] 178
FIGURE 1.9 REPRESENTATION GRAPHIQUE DE « X CONSISTE EN P, Q ET R » PAR TRIP 180
FIGURE 1.10 SCHEMA GENERAL DE TRIP2 181
FIGURE 2.1 UNE IMPLEMENTATION DE QUICKSORT. 185
FIGURE 2.2 INITIALISATION GRAPHIQUE D'UN ALGORITHME. 186
FIGURE 2.3 CONTROLE DE L'EXECUTION D'UN ALGORITHME. 186
FIGURE 2.4 COMPARAISON GRAPHIQUE DE DEUX ALGORITHMES. 187
FIGURE 2.5 CALCUL DU COUT D'UN ALGORITHME. 187
FIGURE 2.6 SYNCHRONISATION D'ALGORITHMES. 188
FIGURE 2.7 STRUCTURE DU SYSTEME PAVANE. 190
FIGURE 2.8 VISUALISATION DES TOURS DE HANOI PAR PAVANE. 191
FIGURE 2.8 SPECIFICATION DE LA REGLE *SHOW_ALWAYS*. 192
FIGURE 2.9 SPECIFICATION DE LA REGLE *SHOW_INIT*. 193
FIGURE 2.10 CODE SOURCE DE LA FONCTION DE RESOLUTION EFFECTIVE DES TOURS DE HANOÏ AVEC LES INDICATIONS D'ANIMATION POUR PAVANE. 193
FIGURE 2.11 VISUALISATION D'UN TRI PAR INSERTIONS DANS SORTING OUT SORTING [BAECKER97A, PAGE 45]. 195
FIGURE 3.1 APERÇU DES FONCTIONNALITES PROPOSEES PAR FIELD [REISS98]. 198
FIGURE 3.2 L'EDITEUR D'ANNOTATION DE FIELD [REISS97]. 199
FIGURE 3.3 VISUALISATION DE L'ARBRE D'APPEL DANS FIELD [REISS97]. 199
FIGURE 3.4 VUE DE ZSTEP95 203
FIGURE 3.5 VISUALISATION DES VARIATIONS DES VALEURS MANIPULEES. 203
FIGURE 3.6 PROGRESSION INTERACTIVE GUIDEE PAR LA REPRESENTATION GRAPHIQUE. 204
FIGURE 3.7 AFFICHAGE D'UN MESSAGE D'ERREUR ET INDICATION DE SA LOCALISATION. 204
FIGURE 4.1 FONDEMENT DE LA CLASSIFICATION DES SYSTEMES DE VISUALISATION DE PROGRAMMES DE PRICE [PRICE97] 205

Chapitre I

Une structure conceptuelle d'une complexité adéquate peut poser les mêmes problèmes de compréhension qu'un escalier en spirale qu'on tenterait d'expliquer avec des mots à des personnes qui n'en ont jamais vu. Quand finalement l'explication a été complétée, l'audience a tendance à se trouver dans un état de confusion totale, ou même aliénée par la présentation. Par contre, une présentation visuelle (« l'image qui vaudrait mille mots ») permet de clarifier instantanément la simplicité élégante du concept en sous-entendant sa complexité nécessaire.

Anthony Judge

[Judge94, page 180]

The word idea *derives from the Greek* "idein": *to see. A sound thinker is* sensible, *or posses common* sense; *a creative thinker is* imaginative *or* farsighted, *a productive* dreamer.

Robert Mekin

[Mekin72, page 1]

Introduction

Cette thèse est consacrée à l'élaboration d'une méthode de création de représentations analogiques de programmes, à sa réalisation dans un environnement de programmation analogique, *Zeugma*[1], et à son évaluation à travers une expérimentation avec les représentations analogiques de programmes générées.

Ce système Zeugma permet :

1) De générer automatiquement des représentations analogiques de programmes. Pour cela, l'utilisateur spécifie les aspects du programme pris en compte ainsi que la manière dont ils seront représentés. Zeugma permet donc de générer des représentations multiples d'un même programme, à des niveaux de granularité et d'abstraction choisis par l'utilisateur. La construction d'une représentation de programmes utilisant des concepts externes à la programmation, telle l'élaboration de plans d'urbanisme, l'aménagement paysager, le comportement de colonies d'araignées, permet de capter des aspects des programmes qui, sinon, seraient difficiles à détecter. Par exemple, la représentation d'un programme sous la forme d'une ville fait apparaître visuellement les parties nécessitant une optimisation ou encore des groupes de fonctions ayant des caractéristiques communes.
2) D'utiliser ces représentations à l'intérieur d'un environnement de programmation incluant des facilités d'édition et d'annotation des programmes, de suivi de leur exécution ou encore de traitement des erreurs. Ceci permet à l'utilisateur d'évaluer, par l'expérimentation, l'utilité des représentations qu'il a élaborées et, par la suite, de se construire *son* propre environnement de programmation intégrant les analogies qui lui sont familières.

Un tel système s'impose car les environnements de programmation actuels se centrent trop sur l'aspect purement local ou exécutoire de la programmation, en offrant essentiellement des aides au niveau syntaxique, des vérifications à la volée de quelques aspects sémantiques (comme le traitement des erreurs), des outils d'observation de l'exécution de programmes, le tout sans aucune possibilité de *prendre*, un tant soit peu, *de distance* avec le code ou les données manipulées. Nous savons bien que, pour comprendre un programme, pour y détecter des erreurs, pour y apporter des améliorations, nous, programmeurs, avons besoin de ces outils : ils nous livrent la matière de base de notre travail. Mais nous – programmeurs - y apportons nos vi-

[1] Le terme Zeugma fait référence au terme grec *sigma* dont une traduction possible est *le point où toutes choses se rencontrent*, notre système permettant la rencontre entre des programmes et leurs représentations par des liens analogiques.

sions propres, nos analogies (éventuellement avec d'autres programmes que nous connaissons, maîtrisons mieux), nos abstractions (nécessaires à cause de la limitation de notre mémoire de travail), nos réorganisations conceptuelles (telles que des classifications selon des critères fonctionnels ou structurels), se situant à un niveau de granularité trop large pour être d'une quelconque aide durant l'écriture de programmes, mais nécessaires lors de sa conception et pour la perception des mécanismes sous-jacents d'un programme inconnu. Pourquoi ne pourrions-nous pas utiliser directement dans la machine nos ébauches, nos images, nos dessins, que nous nous construisons lors de ce processus difficile de la compréhension d'un programme, d'un algorithme, d'un problème ?

Pour répondre à cette attente, et en nous basant sur une étude des limitations des systèmes actuels de visualisation de programmes et des environnements de programmation, nous avons développé une théorie de la construction de représentations de programmes basée sur la notion de liens analogiques[2] entre des concepts et des processus informatiques ainsi que des concepts et processus appartenant à d'autres domaines. Afin de valider notre théorie, nous avons implémenté l'environnement de programmation Zeugma et c'est en utilisant ce système que nous avons élaboré et réalisé des représentations analogiques dont ce mémoire présente quelques exemples.

Dans la suite de ce chapitre, nous décrirons d'abord notre méthode de création de représentations analogiques de programmes, ensuite nous décrirons, au travers d'un exemple, l'utilisation de notre système Zeugma, donnant ainsi un premier aperçu de l'impact qu'un tel système peut avoir.

I.2 Méthode de construction de représentations analogiques de programmes

Notre méthode définit les étapes nécessaires à la mise en relation de programmes informatiques avec des représentations analogiques (qui peuvent, par exemple, être graphiques). Nous l'avons développée afin de répondre aux questions suivantes :

1) Pourquoi les systèmes de sonorisation [Brown92] ou de visualisation d'algorithmes[3] ou de programmes[4] sont-ils *spécialisés* par rapport à un aspect *particulier* des programmes[5] ? Cette limitation est-elle due

[2] Le terme *analogie* est ici à considérer en relation avec les métaphores.

[3] Comme ceux basés sur la notion d'*événements intéressants* de Marc Brown [Brown88] ou de *visualisations déclarées* de Gruia-Catalin Roman [Roman97].

[4] Comme TPM [Eisenstadt87]

[5] Par *aspect particulier*, nous entendons ici aussi bien le domaine de la partie visualisée du programme (algorithme, code source, données) que l'ensemble des critères à partir desquels est construite la visualisation (événement intéressant, état du programme ou encore le code source lui-même).

à une contrainte que les concepteurs de ces systèmes se fixent à eux-mêmes ou bien cette contrainte est-elle indispensable pour construire des systèmes de représentation de programmes ?

2) Comment définir des représentations que l'on pourrait qualifier d'*efficaces*, puisqu'elles représentent en effet tel ou tel aspect (l'algorithme, la composition, le comportement) des programmes, et, de plus, permettent aux utilisateurs d'intégrer leurs propres représentations ? Nous constatons que la plupart des systèmes de représentation de programmes qui existent ne fonctionnent que sur des programmes de taille très réduite et, du coup, la représentation choisie se trouve liée à l'algorithme sous-jacent ou au domaine d'application du programme[6]. Les rares systèmes visant à représenter des programmes de taille plus substantielle, comme TPM (*op cit*.), choisissent des représentations issues de particularismes de l'interprétation du langage de programmation utilisé.

Pour échapper à la limitation des représentations à un aspect particulier du programme, nous proposons un outil de construction de *liens analogiques* qui établit des relations entre un (ou plusieurs) aspect(s) du programme et (un ou) divers aspects du domaine choisi pour la représentation [Ploix96b, 97]. D'autre part, durant l'élaboration de notre système Zeugma [Ploix98], nous nous sommes inspirés des travaux sur les analogies[7], sur l'étude de la compréhension[8] et de la génération d'analogies[9].

Les analogies, une forme de métaphore restreinte aux *correspondances* ou *similarités*, offrent un cadre adapté à l'élaboration d'une méthode de construction de représentations de programmes puisque leur étude pose de manière directe les questions suivantes :

- Quelles sont les interprétations possibles de la correspondance analogique ?[10]
- Quels sont les rapprochements entre le domaine d'origine et le domaine cible et quels sont les liens *sémantiques* les unissant ?[11]

---

[6] Voir, à ce sujet, les séries d'exemples de représentations animées d'algorithmes [Brown92] et [Stasko97a], respectivement construites avec les systèmes Zeus [Brown91] et Polka [Stasko93].

[7] La situation sémantique de nos *analogies* se base sur les travaux de Georges Lakoff [Lakoff81, 93] et d'Umberto Eco [Eco90].

[8] Les travaux de Derdre Gentner [Gentner89] et de son équipe sont à la base de notre modèle de spécification des analogies en terme de *relations* et d'*attributs*.

[9] Les travaux de Douglas Hofstadter [Hofstadter95] et de son équipe, fournissent, avec la notion de *Perception de Haut Niveau,* le cadre théorique de notre *prise de distance* d'avec les programmes dans la construction de nos représentations.

[10] On peut retrouver des exemples de ce type d'interrogation chez Umberto Eco [Eco90].

[11] A ce sujet, les travaux de Stella Vosniadou [Vosniadou89] ont particulièrement

- Quelles sont les caractéristiques des objets permettant leur mise en relation par l'analogie, quels sont les liens à établir entre ces caractéristiques et comment choisir celles qui sont utiles dans un certain but et comment définir celles qui ne le sont pas ?[12]

Les réponses à ces trois questions constituent l'ossature de notre méthode dont nous donnerons ci-dessous une première description étape par étape (l'exposé de notre méthode ainsi que des théories sous-jacentes font l'objet du chapitre II de ce mémoire).

I.2.1 Le choix de l'utilisation de la représentation

Sachant qu'un programme exprime un *algorithme* (par exemple de tri de nombres) qui travaille sur certaines *structures de données* (par exemple une table ou une liste), le tout modélisant un problème (par exemple celui d'ordonner une collection d'objets selon un critère précis), nous devons décider, en premier lieu, lequel (ou lesquels) de ces divers aspects doit être pris en compte et illustré par l'analogie. Cette décision est fonction de la première étape de notre méthode : le choix de l'utilisation, de la finalité, de la représentation analogique. L'utilisation d'une représentation analogique pourra, par exemple, être la recherche des lieux du programme où une optimisation est nécessaire ou possible, d'aider à la compréhension des différentes étapes de l'algorithme ou de son implémentation, voire de mettre en évidence des problèmes sémantiques dans l'implémentation.

Zeugma est le seul système qui permet à l'utilisateur de choisir librement l'utilisation qui sera faite de la représentation qu'il construit. Notre méthode se distingue alors fondamentalement :

- des paradigmes actuels de l'animation d'algorithmes qui demandent une connaissance préalable des programmes (autant pour les *événements intéressants* de Marc Brown (*op cit.*) que pour les *visualisations déclarées* de Gruia-Catalin Roman [Roman92]),
- des environnements de programmation incluant des représentations graphiques : soit parce qu'ils imposent la représentation graphique utilisée – comme TPM [Eisenstadt91], soit parce qu'ils ne permettent que des variations minimes des représentations proposées – comme dans FIELD [Reiss88a].

guidé notre réflexion.

[12] Nous faisons ici directement référence aux travaux de Derdre Gentner et de Douglas Hofstadter. Même si ces auteurs semblent être en opposition [Chamlers92, page 196], nous pensons, comme le montrent Clayton Morrison et Eric Dietrich [Morrison95], que la Structure Mapping Theory de Derdre Gentner et al. s'applique à la *mise en évidence* de l'existence d'analogies alors que la *High Level Perception* de Douglas Hofstadter et al. s'applique à la *production* d'analogies à partir d'exemples.

Cette liberté dans le choix de l'utilisation est rendue possible par ce que nous appelons une *prise de distance* par rapport aux programmes et aux représentations générées : les représentations analogiques que nous créons sont définies par la mise en relation d'aspects de programmes avec des caractéristiques descriptives des représentations.

Suivant notre méthode, la seconde étape de la construction d'une représentation analogique de programme est alors le choix du (ou des) aspect(s) [13] des programmes représenté(s). Par *aspect,* nous entendons toute information *calculable* sur les programmes. Nous distinguons alors deux types d'informations :

1) Les informations construites à partir d'une analyse du code source des programmes. Nous permettrons, par exemple, l'utilisation du texte même du programme, sans aucune analyse préalable, ou des structures mettant en relation diverses parties du programme, telles que, par exemple, les flots de contrôle et de données ou encore les clichés [Wills92]. Par ailleurs, notre implémentation de Zeugma permet la prise en compte de toute autre information extraite des programmes, à la condition que l'utilisateur l'ait préalablement communiquée au système.
2) Les informations issues d'une analyse du *comportement* des programmes au cours de leurs exécutions. Ce point de vue *dynamique* sur les programmes, par nature davantage lié au type de langage de programmation utilisé (objet, fonctionnel, logique ou impératif), permet la construction de représentations analogiques à partir desquels l'utilisateur peut déduire des faiblesses au sein des programmes (erreur, nécessité d'optimisation). Il permet aussi de percevoir l'algorithme utilisé (nous verrons des exemples de déductions *visuelles* d'algorithmes à la section II.3.4 de ce mémoire).

Cette étape est fortement influencée par le choix de l'utilisation de la représentation analogique : la recherche d'erreurs utilisera plutôt une structure syntaxique, comme le flot de contrôle, alors que l'illustration d'un algorithme sera davantage liée aux structures de données. Les deux prendront en compte l'évolution des données pendant l'exécution. Pour tout aspect des programmes, les caractéristiques à prendre en compte doivent être précisées par l'utilisateur. Pour reprendre l'exemple du flot de contrôle, il peut être caractérisé en tant que *structure organisationnelle* (dans une considération statique), en tant que *transmission d'un contrôle* (dans une considération dynamique) ou comme une combinaison des deux (dans une représentation alliant les aspects statiques et dynamiques des programmes). Ce sont ces différentes caractéristiques des aspects de programmes qui seront effectivement prises en compte dans la construction des représentations analogiques et mises en relation avec la représentation choisie.

[13] Correspondant alors au domaine d'origine des analogies.

I.2.2 Le domaine de la représentation

En parallèle avec la deuxième étape intervient le choix du domaine auquel appartient la représentation (domaine *cible* de l'analogie). Cette étape, également influencée par le choix de l'utilisation de la représentation analogique finale, doit aussi intégrer les aspects des programmes représentés. Si l'utilisation d'histogrammes permet une distinction visuelle de la répartition ou de l'évolution des données (cf. notre exemple d'animation analogique d'algorithme page 86), des comportements stylisés du monde animalier peuvent s'avérer efficaces pour mettre en évidence des interactions entre les différentes parties d'un programme (cf. notre représentation analogique du comportement des programmes par une colonie d'araignées sur une toile page 73).

Le choix du domaine de la représentation - l'analogie dont la perception renseignera sur des aspects du programme illustré – appartient au concepteur de la représentation analogique. Ce choix peut être influencé par la *tâche* de la représentation [Gerstendörfer87], selon le *public* auquel elle s'adresse [Ford93], comme opposition à des *métaphores* communément admises [Madsen94]. En accord avec Kopache et Glinert [Kopache90], nous pensons que la découverte et la construction de nouvelles analogies sont une activité *ouverte* à de nouvelles propositions. Les représentations analogiques de programmes par des villes et par des araignées en mouvement sur une toile, que nous présentons dans ce mémoire, ajoutent une dimension supplémentaire à ce débat : la dimension de la *perception analogique* ou du transfert de caractéristiques des éléments de la représentation vers des caractéristiques de programmes. Ces représentations utilisent en effet des éléments ayant une *signification* en dehors du contexte de la programmation ou des représentations habituellement utilisées dans la visualisation de programmes. Comme nous le verrons lors de la description de ces représentations, cette signification renseigne sur l'objet représenté (des aspects de programmes).

Notre méthode *encadre* ce degré de liberté dans le choix de la représentation par l'établissement de liens qui la mettent en relation avec les programmes. Ces liens, basés sur la création de *relations analogiques*[14] entre les domaines, conduisent le concepteur des représentations à préciser les liens entre les objets du domaine d'origine et les objets du domaine cible, en décrivant avec précision *toutes* les caractéristiques de programme considérées et en vérifiant qu'elles apparaissent clairement dans la représentation analogique.

De la même manière que pour les aspects des programmes, les objets du domaine cible ne seront pas spécifiés de manière figée mais de manière

[14] La *perception analogique* fait référence à l'aspect sémantique des analogies ([Le Guern75], [Lakoff81, 83] et [Eco90]) alors que les *relations analogiques* font référence aux travaux sur l'étude des processus sous-jacents au transfert de caractéristiques présentes dans les analogies ([Gentner89], [Hofstadter95]).

générative, abstraite. Ils seront spécifiés en termes de propriétés et de relations avec d'autres objets[15] s'appliquant à des objets animés ou statiques[16]. Nous reviendrons plus en détail sur ce dernier point dans la présentation de l'implémentation de notre méthode dans Zeugma (cf. section I.2 de ce chapitre).

I.2.3 L'établissement des liens analogiques

La dernière étape de notre méthode consiste à établir des liens analogiques entre les aspects des programmes et les propriétés des représentations choisies.

L'établissement de ces liens mettra en relation les caractéristiques des aspects des programmes prises en compte avec les propriétés décrivant les représentations générées. L'ensemble de ces liens constituera une *description abstraite* à partir de laquelle sera générée la représentation analogique d'un programme. De plus, nous qualifions ces liens de relations *analogiques* entre deux domaines car, d'une part, ils décrivent le *transfert de caractéristiques* entre les domaines *origine* et *cible* et, d'autre part, ils se situent à un niveau d'abstraction par rapport aux programmes, comparables à la notion de *Perception de Haut Niveau* [Chamlers95].

Pour illustrer ces descriptions abstraites, prenons l'exemple de la construction d'une représentation analogique de la communication entre les différentes fonctions d'un programme par des musiciens de Jazz[17]. Chaque fonction sera mise en relation avec un musicien ou groupe de musiciens, et le passage du contrôle sera relié avec l'activation de ce musicien.

Les aspects des programmes intervenant alors sont :

- La construction d'une classification[18] des fonctions qui permet de les lier avec les différents types d'instruments,
- La détection du transfert de contrôle pendant l'exécution du programme en vue d'activer tel ou tel musicien ou groupe de musiciens.
- La détection des opérations effectuées par les fonctions, les modifica-

[15] Par exemple en terme de relation spatiale si le support choisi est la représentation graphique ou en terme d'intervalles harmoniques si le support choisi est le son.

[16] Par exemple le dessin d'un objet graphique particulier ou la génération d'une mélodie.

[17] L'implémentation actuelle de notre méthode permet la création de représentations analogiques utilisant des objets graphiques. Si notre exemple utilise un support différent, le son, c'est pour illustrer le fait que notre méthode n'est pas liée à un support particulier.

[18] Cette classification pourrait, par exemple, être issue du flot de contrôle en prenant alors comme critère la position dans le flot ou être issue d'une étude de leur composition (par exemple avec le calcul de leur signature [Balmas97b]).

tions des valeurs, etc. qui indique la partition (ou le thème) au musicien actif.

Le premier aspect, qui permet de construire l'orchestre, s'adresse à une étude de la composition des programmes alors que les deux autres, qui vont entraîner la génération de sons, concernent le comportement durant l'exécution. Cet exemple de représentation analogique illustre la distance conceptuelle qui existe entre l'étude de la composition des programmes et celle de leur comportement. De plus, il existe de nombreux critères dans la formation de groupes musicaux comme il existe de nombreux critères dans la construction de programmes. C'est cette multiplicité de critères qui nous a amené à construire la description des représentations analogiques à partir d'abstractions sur les programmes comme sur les représentations. Ces descriptions abstraites permettront d'appliquer ces liens analogiques dans de multiples contextes, à de multiples programmes, par de multiples représentations analogiques – chacune illustrant un point de vue différent.

Voyons maintenant, à travers un scénario de son utilisation, une brève description de notre système Zeugma.

I.3 Scénario d'utilisation de Zeugma

I.3.1 Présentation du système

Zeugma à été élaboré pour aider la spécification et la mise au point de représentations analogiques (cf. figure 1.1 présente la structure générale de Zeugma). Il intègre également des fonctionnalités qui permettent l'utilisation des représentations analogiques générées dans un environnement de programmation. Il a été construit en utilisant le système Xbvl [Greussay76, 77, 82][Wertz83][Sendoya92], un environnement de programmation Lisp intégrant des facilités graphiques 3D [Ploix96a] ainsi que des facilités d'édition, d'annotation et de traitement des erreurs [Wertz78, 83]. La contribution de Zeugma, successeur de Visal [Ploix93], à l'évolution de Xbvl (réalisant ainsi les perspectives annoncées dans [Wetrz93]) est d'ajouter une composante *analogique* dans l'édition, dans la consultation et dans l'observation de programmes en permettant l'utilisation des représentations analogiques au sein de ces activités.

I.3.2 Construction d'une représentation analogique

L'exemple de représentation analogique que nous allons utiliser comme fil conducteur de cette première présentation de notre système est la représentation du comportement des programmes par des araignées sur une toile. Cet exemple sera repris dans la suite de ce mémoire[19].

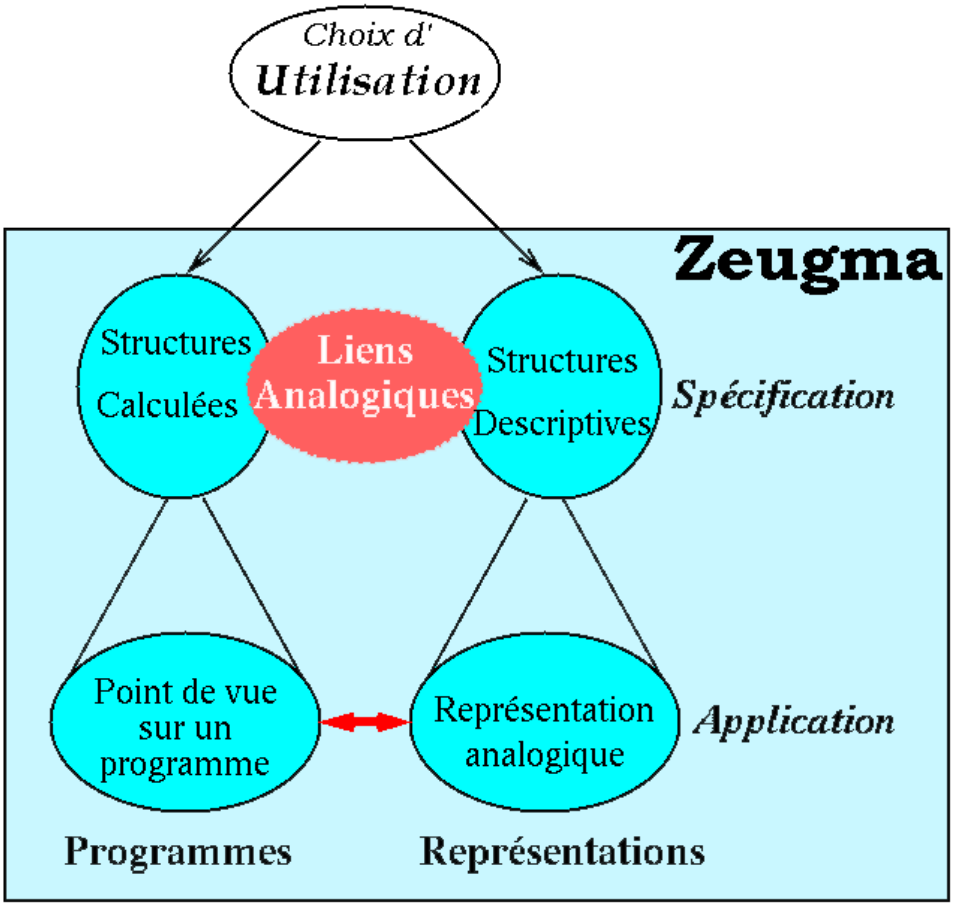


Figure 1.1 Structure générale de Zeugma

Cette représentation aide à la découverte de programmes inconnus et, plus précisément, à la visualisation du rôle *effectif*[20] de leurs différentes fonctions. Elle est composée d'araignées positionnées en cercle autour d'une toile, chacune des araignées symbolisant une fonction du programme. Pendant l'exécution, à chaque appel d'une fonction, l'araignée qui la symbolise se rapprochera de l'araignée qui symbolise la fonction initiatrice de l'appel. A la fin de l'exécution du programme, des groupes d'araignées, formés à la suite de rapprochements successifs, indiqueront la présence de groupes de fonctions qui ont collaboré dans l'exécution d'une tâche. Outre la formation de

[19] Cf. la section II.3.3, où sont décrits en détail ses composants *analogiques*. Cette représentation est également utilisée pour présenter les différents aspects de Zeugma (cf. chapitre III).

[20] Nous nommons un *rôle effectif* celui qui est déduit de l'exécution du programme et *rôle calculé* celui qui est élaboré à partir d'analyses du code source.

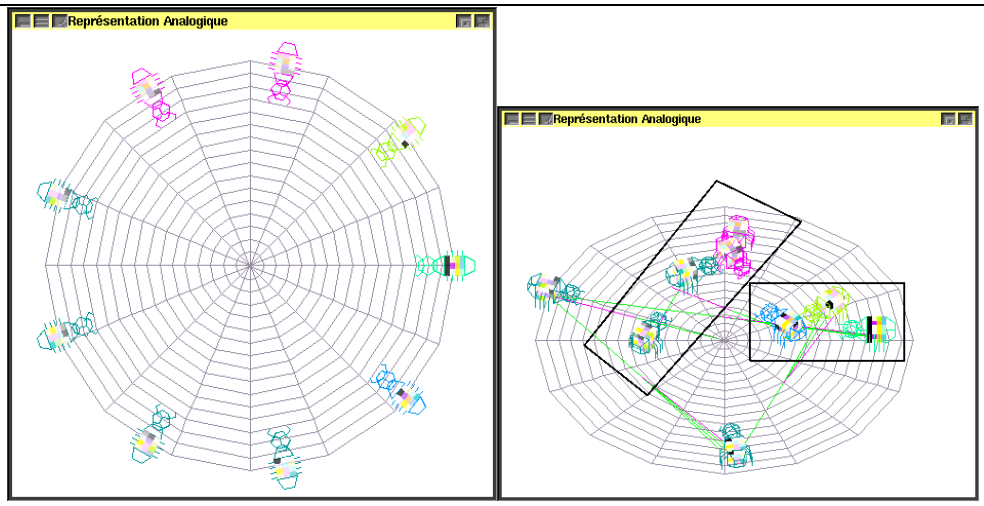


Figure 1.2 Exemples de la représentation des programmes par des araignées

groupes fonctionnels, ces rapprochements successifs d'araignées renseigneront efficacement le programmeur sur les parties du programme dans lesquelles une optimisation peut avoir d'importants effets sur la rapidité du programme : il peut distinguer les fonctions qui se sont appelées à de nombreuses reprises. Des liens symbolisant l'échange de données apparaîtront également au cours de l'exécution, reliant les araignées lors de l'appel et permettant ainsi de reconnaître visuellement la circulation des données pendant l'exécution.

La figure 1.2 propose deux vues de cette représentation. Dans la première vue, les araignées sont disposées de manière circulaire et l'utilisateur, comme nous le verrons plus loin, peut interroger l'image afin d'identifier chaque araignée. Le système lui indique quelle fonction du programme elle représente. La seconde vue propose l'état de la toile d'araignée à la fin de l'exécution. Deux groupes de fonctions se sont constitués (les deux encadrés en haut et à droite de la toile). Outre l'identification de groupes fonctionnels, il apparaît en effet que les trois araignées du groupe du haut interagissent de manière plus importante que la quatrième (encore en déplacement vers le groupe). Une optimisation de ce programme s'attachera donc à examiner ces trois fonctions, leur interaction, et s'il est possible d'en éliminer une pour l'intégrer dans l'une des deux autres afin d'éviter des appels coûteux.

Cette représentation combine deux aspects des programmes :

1) Un aspect lié à la composition des programmes, décidant de la construction de l'image initiale : la position initiale et la forme des araignées,
2) Un aspect lié à l'exécution des programmes, décidant de son animation pendant l'exécution : le mouvement des araignées et le dessin des liens.

Figure 1.3 Liste des aspects prédéfinis dans Zeugma

I.3.2.1 Choix des aspects de programmes

Afin d'aider la construction de cette représentation, Zeugma propose deux séries d'outils. La première permet de spécifier la manière dont sera générée la représentation initiale. La seconde permet de spécifier les liens *dynamiques* entre l'exécution du programme et la représentation analogique générée.

Par exemple, le choix de la disposition des araignées peut être spécifié selon un parcours simple de l'ensemble des fonctions présentes. L'ensemble de ces fonctions peut alors être construit suivant : un ordre syntaxique (par classement alphabétique), l'ordre d'apparition dans le flot de contrôle ou encore selon des critères plus élaborés, déterminant l'ordre des fonctions en dépendance des *types* d'opérations qu'elles effectuent. Ce choix, dans Zeugma, s'effectue par la sélection de l'*aspect principal* dans la liste des aspects définis (cf. figure 1.3). Cette liste d'aspects n'est pas figée : l'utilisateur peut toujours définir de nouveaux aspects qui s'ajouteront aux aspects prédéfinis et seront alors directement utilisables dans la construction des représentations.

```
(define-PDV Flot-de-contrôle              ; définition de l'aspect flot de contrôle
  (nom FDC)  (titre "Flot de contrôle")   ; nom et titre pour les menus de l'interface
  (analyse (analyse-FDC))                 ; fonction d'analyse
  (tests fdc-CAR fdc-CDR fdc-ATOM))       ; caractéristiques liées au parcours du flot

(define-Test fdc-CDR                      ; définition de la caractéristique fdc-CDR
  (titre "Flot de contrôle - Largeur")    ; titre pour les menus
  (test-eval (cdr flow))                  ; test d'activation sur le flot parcouru
  (next-flow (cdr flow)))                 ; progression dans le flot (valeur transmise)
```

Figure 1.4 Extrait de la définition du *flot de contrôle* comme un aspect des programmes

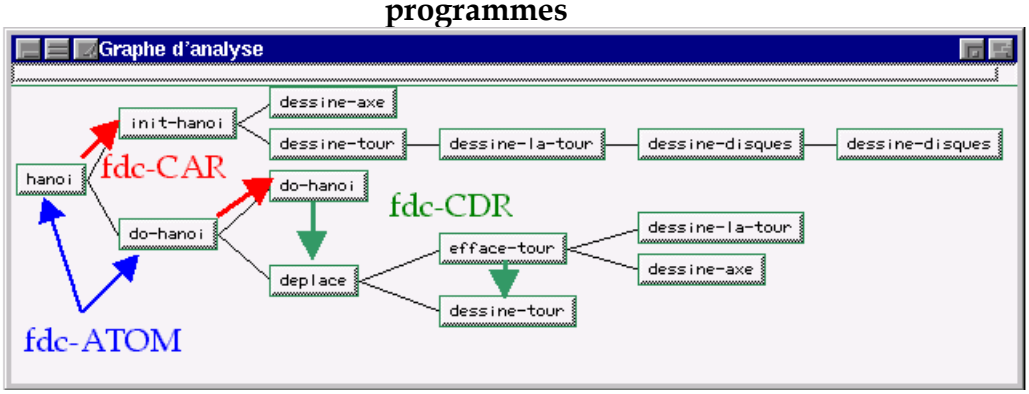


Figure 1.5 Caractéristiques du flot de contrôle

I.3.2.2 Définition d'aspect des programmes

Prenons l'exemple du flot de contrôle[21]. La définition d'une telle structure décrivant les liens syntaxiques entre les fonctions d'un programme comme *aspect des programmes* utilisable dans la construction de représentations analogiques, nécessitera de la part de l'utilisateur :

- La construction d'une fonction calculant les données nécessaires pour le nouvel aspect de programme (en l'occurrence le graphe du flot). Dans la première définition de la figure 1.4 (define-PDV), cette première étape correspond à (analyse (analyse-FDC)), analyse-FDC étant la fonction Lisp construite afin de calculer le flot de contrôle d'un programme à partir de son point d'entrée.
- La spécification d'un ensemble de méthodes travaillant sur ces données. Elles décriront les différentes *caractéristiques* de l'aspect défini et serviront, dans la représentation analogique, de base descriptive du domaine d'origine. Dans la première définition de la figure 1.4, cette étape correspond à (tests fdc-CAR fdc-CDR fdc-ATOM).

Nous avons extrait trois traits caractéristiques du flot de contrôle : les éléments (fdc-ATOM) et les liens reliant les différents éléments (fdc-CAR et

[21] L'arborescence présentée dans la figure 1.5 présente un exemple de l'un de ces flots.

fdc-CDR). Ces liens correspondent respectivement au lien entre fonction appelante et fonction appelée et au lien entre les différentes fonctions appelées. Ces trois caractéristiques (cf. figure 1.5) permettent de distinguer chacun des éléments et de parcourir l'ensemble du flot. La deuxième définition de la figure 1.4 (define-test fdc-CDR), présente la définition de la méthode relative à la caractéristique fdc-CDR (de parcours de l'ensemble des fonctions appelées par une fonction). Dans cette définition, le champ test-eval correspond à la condition d'activation de cette caractéristique (l'ensemble des fonctions parcourues n'est pas vide) et le champ next-flow indique l'effet de l'application de la méthode sur le flot parcouru. La construction des représentations analogiques, comme nous allons maintenant le voir, ne fait ainsi plus directement référence à la structure particulière du résultat de la fonction d'analyse mais à des méthodes, spécifiées et documentées par le concepteur de l'aspect, permettant de traiter les données relatives à cet aspect.

I.3.2.3 Spécification du schéma générateur de représentations

Le travail principal, dans la construction d'une représentation analogique de programme, réside dans l'établissement des liens entre les programmes et les représentations. Comme nous l'avons dit précédemment, la validité d'une représentation analogique, mis à part le coté subjectif du domaine analogique choisi, est dépendante des relations établies entre les objets présents. Notre implémentation de ce point se base sur la création de *schèmes générateurs de représentations analogiques*. Ces schèmes sont composés d'*Objets de Relations Structurelles* (ORS) spécifiant la manière dont une représentation graphique va être générée à partir d'un programme.

Les schèmes générateurs et les ORS ont la particularité d'établir des liens entre les relations et les attributs appartenant aux deux domaines de la représentation analogique (les programmes et les images) de la manière suivante :

- chaque ORS lie une caractéristique d'un aspect des programmes, nommée alors *condition d'activation*, à un objet graphique particulier,

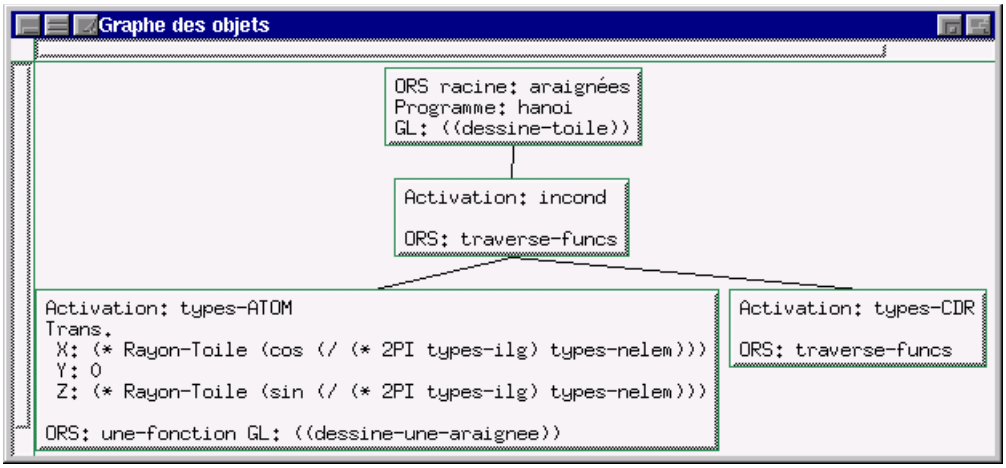


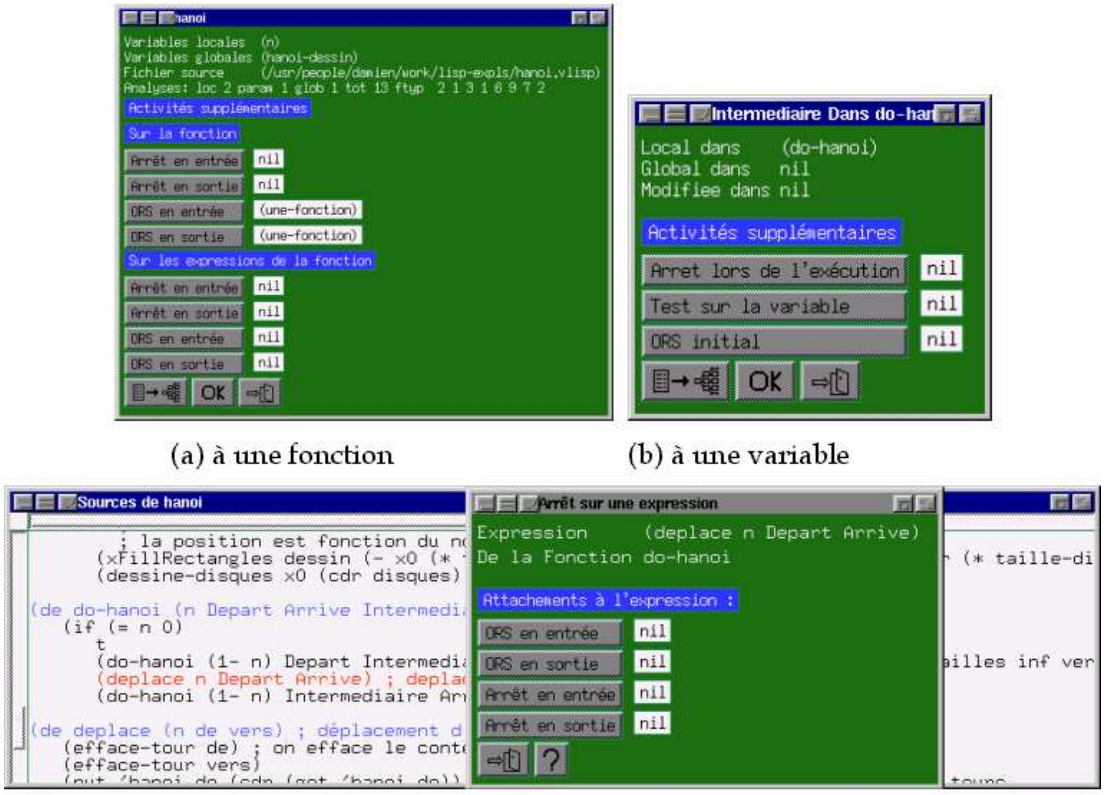
Figure 1.6 Schéma générateur des araignées sur une toile

- la transition d'un ORS à un autre, correspond à la progression dans le parcours des données correspondant à un aspect du programme représenté. Elle décrit les *transformations* graphiques à effectuer dans la représentation analogique.

La figure 1.6 présente le schéma générateur de la représentation des programmes par des araignées sur une toile. Ce schéma va déterminer la procédure utilisée pour générer l'image *initiale* de la représentation : le placement des araignées autour de la toile en suivant les différentes fonctions présentes dans le programme étudié. Il est ainsi composé de trois ORS différents :

1) l'ORS *racine*, ou initiateur de la génération de la représentation, nommé araignées. Cet ORS, comme l'indique la représentation, précise le point d'entrée du programme étudié (la fonction hanoi). De plus, son activation au moment de la génération entraînera le dessin de la toile sur laquelle vont venir se placer les araignées (dessine-toile).
2) l'ORS travers-func. Cet ORS a comme fonction de *parcourir* l'ensemble des fonctions du programme. Il est activé de manière inconditionnelle (incond) au début de la génération de la représentation et le lien auto-référent lié à la condition d'activation *types-CDR* sera actif durant tout le parcours de l'ensemble des fonctions du programme.
3) l'ORS une-fonction : correspondant à chaque élément de l'ensemble des fonctions, son activation aboutira au dessin des araignées (dessine-une-araignée). De plus, son activation entraîne une translation, positionnant les différents éléments autour de la toile (par rapport à leur numéro d'ordre types-ilg dans l'ensemble des fonctions, types-nelem indiquant le nombre d'éléments de cet ensemble).

La particularité de ces schèmes est de décrire la manière dont des aspects des programmes seront considérés dans une représentation analogique : le schéma que nous venons de décrire n'est pas lié à un programme particulier et peut être appliqué à tout programme, qu'il soit connu ou inconnu, aboutissant à la création d'une image particulière à chaque programme étudié. Il suffira pour cela de changer le point d'entrée du programme.



(a) à une fonction (b) à une variable

(c) à une expression particulière

Figure 1.7 Attachements à des éléments du programme

I.3.2.4 Attachements dynamiques

Le schéma générateur que nous venons de décrire se base sur une étude de la composition des programmes. La seconde dimension de notre représentation analogique est liée au comportement du programme pendant son exécution.

Notre système permet de spécifier ces liens *dynamiques*, que nous nommons *attachements* d'une activité à un élément du programme, aussi bien par les ORS que par les éléments du programme représentés. La figure 1.7 montre les fenêtres de spécifications correspondant aux méthodes d'attachements d'une activité supplémentaire aux différents types d'éléments du programme. La figure 1.8 présente la spécification des attachements automatiques entre un ORS et des éléments du programme.

La différence principale entre ces deux méthodes est que, dans la spécification d'attachements à un élément du programme, ceux-ci seront spécifiques à l'élément pris en compte (une fonction, une expression particulière ou encore une variable) alors que ceux établis comme propriété d'un ORS entraînent leur application à tous les éléments du programme désigné (toutes les fonctions, toutes les expressions ou toutes les variables du programme).

On peut dire alors que la première méthode d'attachement rapproche notre système des paradigmes des *événements supplémentaires* ou des *visualisations déclarées* évoqués plus haut, mais que la seconde le distingue nettement :

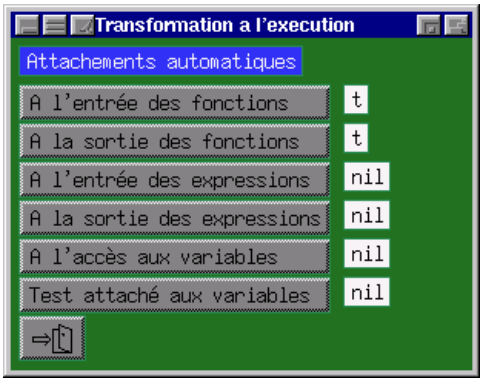


Figure 1.8 Attachements comme propriétés d'un ORS

la construction d'animation du comportement des programmes suivant ces paradigmes demande à l'utilisateur de *bien* connaître les différents éléments du programme visualisé. De plus, il est important de noter que, du fait de l'utilisation du système Xbvl comme interprète sous-jacent à Zeugma, ces différents attachements sont transparents dans notre système : ils ne requièrent pas de modification du programme de la part de l'utilisateur ni ne modifient son comportement pendant l'exécution.

Dans notre représentation de programmes par des araignées en mouvement, les attachements dynamiques concernent l'entrée et la sortie de toutes les fonctions du programme représenté. La méthode choisie afin de spécifier ce mode d'attachement sera donc la seconde : un attachement automatique entre l'ORS *une-fonction*, lié au dessin et au placement des araignées, et (cf. figure 1.8) l'entrée et la sortie des fonctions.

I.3.2.5 *Conclusion : propriétés de l'ORS* une-fonction

L'ORS *une-fonction* joue donc un rôle central dans cette représentation analogique. Il spécifie le dessin et le placement des araignées pendant la génération de l'image initiale et son activation pendant l'exécution du programme entraîne leur animation.

La figure 1.9 présente l'affichage par Zeugma des caractéristiques de cet ORS. Détaillons la signification de ces informations :

- Le champ *données graphiques* renseigne sur les instructions de dessin à évaluer au moment de la génération de l'image initiale. Dans le cas présent, la fonction Lisp *dessine-une-araignée*, dont le résultat est le dessin d'une araignée, sera activée.
- Le champ *reconstruction*, spécifiant si l'objet graphique est générique (toujours le même) ou spécifique (prenant en compte des éléments liés à la caractéristique ayant entraîné l'activation de l'ORS), a comme valeur *t* du fait du caractère particulier du dessin de chaque araignée. Ces dernières sont d'une couleur relative à des statistiques effectuées sur les fonctions qu'elles représentent et, comme l'indique ce champ, demandent une reconstruction pour chaque dessin initial.

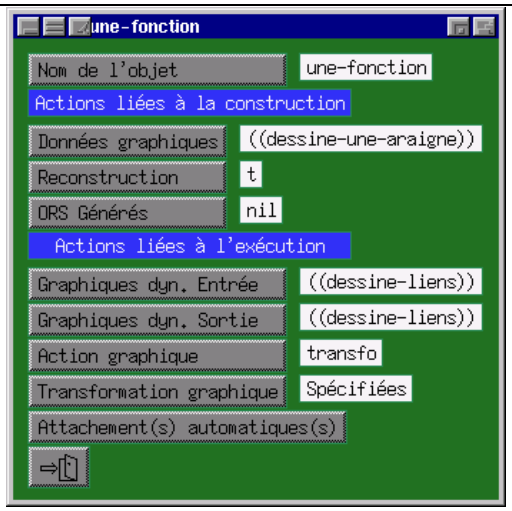


Figure 1.9 Propriétés de l'ORS *une-fonction*

- Le champ *ORS Générés* indique que cet ORS n'entraîne pas l'activation d'autres ORS dans le schéma générateur de la représentation.
- Les champs *Graphiques dyn. Entrée* et *Sortie* sont relatifs, dans notre visualisation, au dessin des liens reliant les araignées pendant l'exécution. Symbolisant l'échange de données, ils seront mis à jour à chaque entrée (pour afficher les paramètres d'appel) et à chaque sortie (pour afficher la valeur retournée) des fonctions.
- L'*action graphique* dynamique est une transformation simple, aboutissant au déplacement (spécifié par le champ *Transformation graphique)* de l'araignée vers celle qui symbolise la fonction originaire de l'appel ayant entraîné son activation.
- Le bouton de commande *Attachement(s) automatique(s)*, permettant de spécifier les attachements automatiques liés à l'ORS, entraîne l'affichage de la fenêtre présentée à la figure 1.8 décrite précédemment.

Les éléments centraux de notre implémentation de la création de représentations analogiques de programmes sont ces structures de données que nous avons nommées les *Objets de Relations Structurelles*. Elles constituent les *schèmes de génération des représentations* et permettent également la spécification des animations des représentations.

I.3.3 Observation de la représentation analogique

Après avoir spécifié les différentes caractéristiques de la représentation, tant au niveau du schème générateur de la représentation qu'au niveau des différents liens dynamiques, l'utilisateur peut demander au système Zeugma de *générer* la représentation analogique d'un programme. Le résultat de cette première phase de l'utilisation des représentations aboutit à l'affichage de l'image *initiale* de la représentation (cf. figure 1.2 gauche). Cette image représente l'application d'une méthode de génération de représentation sur des analyses de la composition. L'activation des liens dynamiques se fera lors de

l'exécution du programme.

Notre système propose alors différents outils :

1) d'aide à la lecture des représentations, permettant de *consulter* les images générées afin de visualiser les éléments du programme représentés,
2) de réduction interactivement, si nécessaire, de la complexité des images générées,
3) de contrôle du déroulement de l'exécution pas à pas,
4) de visualisation d'historiques d'exécutions combinant la présentation des événements de l'exécution avec les différentes animations de la représentation analogique.

I.3.3.1 Consultation des liens

Dans la représentation dont nous avons décrit l'élaboration, chaque araignée représente une fonction du programme. Même si l'aspect (la couleur, ou les éléments de l'abdomen) distingue les araignées les unes des autres et se trouve construit suivant les caractéristiques de la fonction représentée (nous détaillons ce point dans la section II.3.3.2.2), l'outil d'aide à la lecture des représentations de Zeugma permet, par la simple sélection d'une araignée, d'afficher les informations relatives à la fonction représentée. Ces informations sont (cf. figure 1.10) : la fonction du programme, les instructions graphiques et les transformations successives ayant abouti au dessin de l'araignée et des informations supplémentaires sur la fonction elle-même (à savoir, par exemple, les variables locales ou globales qu'elle manipule, son fichier source et des statistiques sur sa composition).

Les informations présentes dans cette fenêtre sont spécifiques au type d'élément du programme qu'elles concernent. Si l'élément de la représentation analogique sélectionné est d'un autre type (par exemple une expression ou une variable) les informations présentées seront alors relatives à ce dernier. Ainsi, l'interrogation d'une représentation, outre la lecture des liens avec le programme, permet de guider sa découverte en renseignant l'utilisateur sur sa composition.

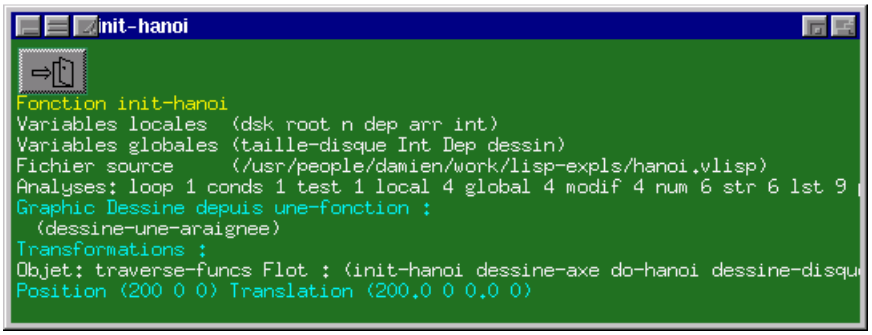


Figure 1.10 Informations sur un élément de la représentation

I.3.3.2 Réduction de la complexité

La représentation analogique du comportement des programmes par des araignées contient un nombre d'éléments graphiques (les araignées) relatif au nombre de fonctions présentes dans le programme. Sa complexité peut ainsi devenir importante dans le cas d'un programme de grande taille. Elle restera cependant relativement peu importante comparée à des représentations analogiques qui représentent les programmes à un niveau de granularité plus *fin* (comme celles qui intègrent la composition des fonctions). Ces représentations – même si le nombre de fonctions présentes dans un programme est peu important – peuvent rapidement atteindre un niveau de complexité auquel des outils de navigation dans une image (positionnement, zooms, rotations) ne permettent plus de distinguer clairement les différentes informations.

Pour aider l'utilisateur dans la lecture de telles représentations, Zeugma propose un outil qui permet de régler a posteriori (une fois la représentation générée) le niveau de granularité présenté. Ceci nous amène à introduire la notion de *structure des représentations générées* correspondant aux différentes étapes de l'application du schème générateur de représentations à un programme.

La figure 1.12 montre la structure de la maison présentée à la figure 1.11, construite à l'aide de la représentation analogique des programmes comme des villes[22]. Chaque élément de cette structure représente l'activation d'un ORS lié soit à une transformation graphique, soit à la génération d'un objet graphique, soit aux deux opérations. Il est ainsi possible, en suivant cette structure, de distinguer les différents liens entre le programme et la représentation et les différentes étapes de la construction de la représentation.

Une partie de la structure représentée dans la figure 1.12 apparaît de couleur différente. Cette partie à été sélectionnée en vue d'une réduction de la complexité de la représentation. La confirmation de sa sélection aboutira à la disparition dans la représentation graphique des éléments correspondants. La maison présentée à la figure 1.11 deviendra alors celle de la figure 1.13. Il apparaît que les éléments graphiques présents à droite de la maison ont disparu, ils ont été réduits.

[22] Nous présentons en détail cette représentation analogique des programmes comme des villes à la section II.3.1.

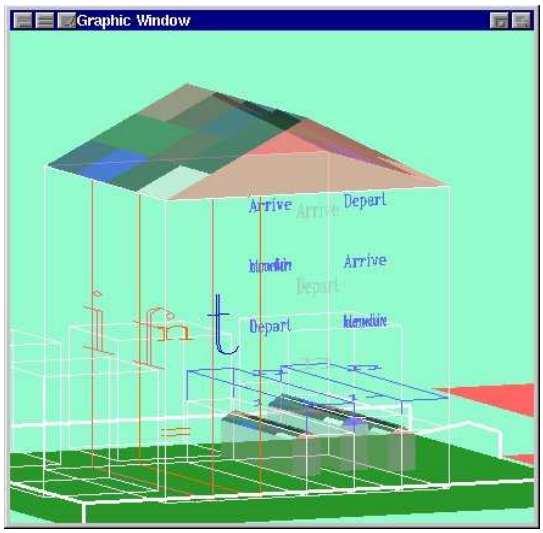


Figure 1.11 Représentation de la composition d'une fonction par une maison

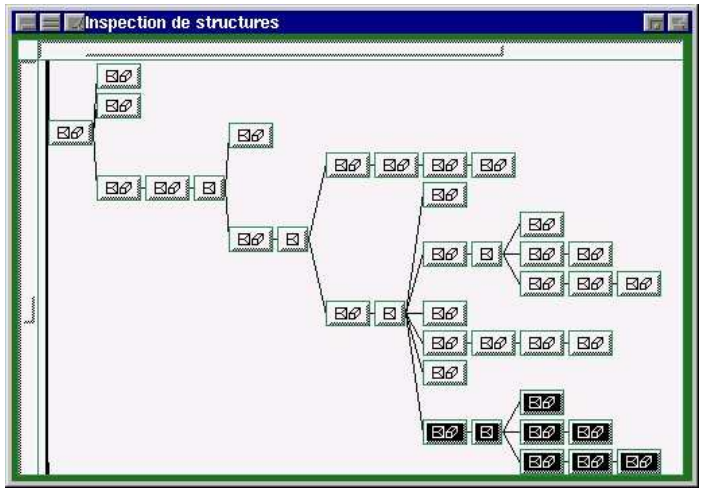


Figure 1.12 Structure d'une représentation

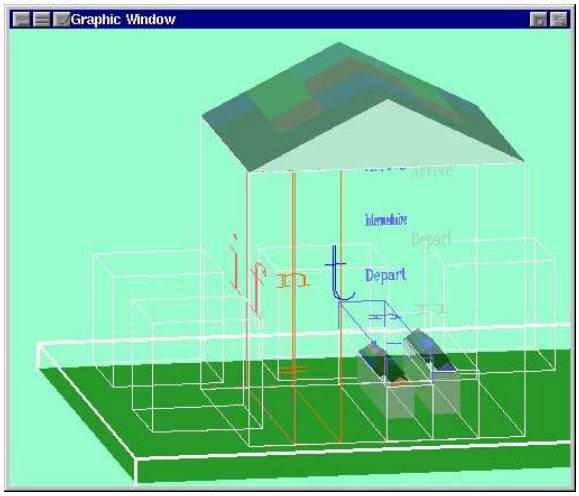


Figure 1.13 Maison réduite

I.3.3.3 Contrôle des animations

Un autre aspect de notre système Zeugma, portant sur l'utilisation et l'observation des représentations analogiques, concerne les différentes étapes de l'animation des représentations. Relatives à une exécution d'un programme, ces animations en offrent une trace analogique. Afin de renseigner l'utilisateur de ces représentations sur la signification des liens actifs entre l'exécution d'un programme et une animation, et d'utiliser pleinement cette dimension d'observation des programmes, notre système offre deux types d'outils : un outil de contrôle du déroulement d'une exécution et un outil de visualisation de l'historique d'une exécution.

Le premier outil, qui permet le contrôle d'une exécution, consiste en la possibilité d'activer un mode *pas à pas*. La particularité de ce mode est que l'utilisateur peut spécifier le type d'opération entraînant l'arrêt de l'exécution et ce, de manière générale (tous les éléments d'un certain type) ou particulière (tel élément spécifique du programme étudié). Les fenêtres présentes dans figure 1.7 (page 16) montrent les différents types de spécifications des attachements entre les éléments d'une représentation et les différents types d'éléments des programmes. Ces fenêtres permettent également de spécifier si l'évaluation d'une (ou des) fonction(s), d'une (ou des) variable(s) ou d'une (ou des) expression(s) arrêtera l'exécution du programme étudié. Ceci permettra alors d'observer, pendant l'exécution du programme, les conditions de l'activation d'une animation et d'interroger le système sur le contexte programmatoire (valeur des variables, pile d'exécution, etc.) ayant conduit à l'exécution de tel ou tel élément. Le partage des données relatives aux attachements dynamiques par un certain type d'élément, par exemple, toutes les fonctions du programme, entraînera alors l'activation d'un mode pas à pas relatif à ce type d'élément.

I.3.3.4 Navigation analogique dans un historique

Le second type d'outil est la visualisation de l'historique d'une exécution en parallèle avec son animation analogique. Travaillant sur les données relatives à une exécution passée, l'utilisateur de cet outil peut *naviguer* dans une exécution et en visualiser les différentes étapes en détail. La particularité de cet outil (cf. figure 1.14), outre la visualisation étape par étape du programme combinant l'objet programmatoire exécuté (la fonction, l'expression, la variable), les valeurs manipulées et l'animation analogique, est de permettre une navigation en avant et en arrière dans l'exécution visualisée. Ce qui rend possible de retrouver l'origine d'un état du programme particulier et de retracer à volonté les différentes étapes qui y ont conduit.

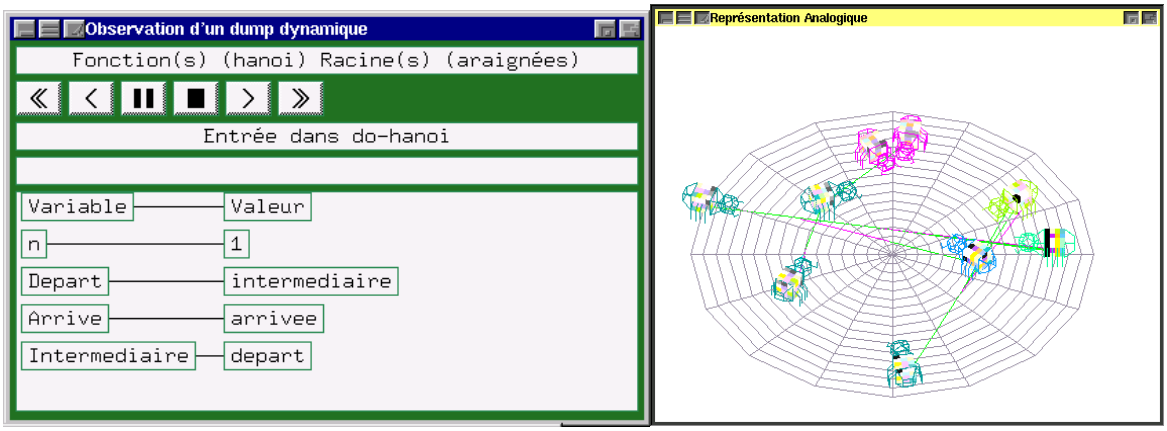


Figure 1.14 Historique d'une animation analogique

I.3.3.5 Impact de Zeugma

Le système Zeugma implémente notre méthode de construction de représentations analogiques de programmes. Il se situe de fait parmi les systèmes de visualisation d'algorithmes mais, également, parmi les systèmes de visualisation de programmes (nous argumenterons ce point en détail dans le chapitre IV). De plus, il a comme singularité d'être intégré dans un environnement de programmation et d'offrir des fonctionnalités permettant d'utiliser les représentations analogiques dans ce contexte. Notre travail englobe ainsi une double dimension : la création et l'utilisation de programmes. Car Zeugma est à la fois un outil d'aide à la création expérimentale de représentations analogiques et, dans l'environnement de programmation associé, un outil d'aide à la compréhension et à l'observation de programmes et d'algorithmes.

I.4 Plan de lecture

La suite de notre étude se répartit comme suit :

Dans le chapitre II, après avoir présenté les différentes influences et réflexions à la source de son élaboration, nous exposerons en détail notre méthode de création de représentations analogiques de programmes. Nous insisterons alors particulièrement sur différentes approches automatisant la reconnaissance ou la création d'analogies, l'étude des liens analogiques étant une dimension essentielle de notre méthode. Dans un troisième temps, nous décrirons une série d'exemples de représentations analogiques de programmes construites suivant notre méthode.

Le chapitre III présente notre environnement de programmation Zeugma implémenté sur la base de notre méthode de création de représentations analogiques. Dans un premier temps, nous présenterons les caractéristiques de ce système, puis, en suivant un exemple, le détail de son interface.

Le chapitre IV compare notre système Zeugma avec des systèmes appartenant à trois domaines directement liés à notre travail : la visualisation

de programmes, la visualisation d'algorithmes et les environnements de programmation intégrant des visualisations graphiques. Il conclut en présentant la situation de notre système parmi les systèmes de visualisation de programmes.

Enfin, dans le chapitre V, nous évaluerons les limites actuelles de notre système, tout en en prévoyant les extensions et les perspectives de développement.

En annexe, nous donnerons le code source de Zeugma, les définitions complètes des différentes analogies présentées dans ce mémoire, le code source des programmes représentés et l'implémentation de l'interface entre Xbvl et Open GL que nous avons réalisée au cours du développement de Zeugma.

Chapitre II

Computer science is not only chronologically new but also radically new in kind. Its mental processes, its languages, and the realities with which it deals are different from those of other sciences.

Gerald Johnson

[Johnson94, p. 97]

II Analogie, Métaphore et Visualisation de Programmes

II.1 Prolégomènes

II.1.1 Question de *complexité*

II.1.1.1 En informatique

Les utilisateurs d'ordinateurs, qu'ils soient novices ou experts, réalisent certainement dans de très rares occasions la distance qui existe entre les termes utilisés pour désigner tel ou tel objet informatique et leur réalité physique. Ainsi, Gerald Johnson [Johnson94] donne comme exemple le terme *fichier* qui est bien loin de désigner l'ensemble des informations réparties de manière plus ou moins aléatoire sur une surface magnétique appelée aussi *disque dur*[23]. Mais on pourrait citer d'autres exemples appartenant maintenant au sens commun :

- Les utilisateurs de systèmes récents connaissent certainement un outil nommé *Défragmenteur de disque*. Or cet outil, s'il s'avère être directement lié à la signification *physique* des fichiers, n'a par contre aucun lien direct, dans sa dénomination, avec le sens propre du terme *fichier*.
- Tous les systèmes modernes sont maintenant *multitâches*, pouvant, au sens littéral du terme, effectuer plusieurs tâches en même

Figure 1.1 « The File Room » Antonio Muntadas, 1994

[23] Je ne connais pas la provenance du terme *disque dur*, qui doit certainement être liée au terme *floppy disk* dont la traduction n'est pas *disque mou* mais *disquette*…

temps. Un fait demeure souvent caché : ces tâches sont gérées la plupart du temps par un unique processeur[24]. La notion de temps ou de simultanéité est différente pour un processeur ou pour un utilisateur.

Les exemples précédents s'adressent à tous les utilisateurs d'ordinateur aujourd'hui. Mais ce problème se retrouve aussi, et de manière plus cruciale, dans la distance existant entre l'explication d'un algorithme, même simple, et son implémentation. Prenons l'exemple du QuickSort [Hoar62], algorithme de tri d'une liste de nombres. Voici une description possible de cet algorithme [Convington97, page 184] :

1) Prendre arbitrairement un élément de la liste (par exemple le premier). Le nommer *pivot*.
2) Diviser le reste de la liste en deux : l'une contenant les éléments plus petits que *pivot*, liste que nous nommerons *PlusPetits*, et l'autre contenant les éléments plus grands que *pivot*, liste *PlusGrands.*
3) Trier récursivement *PlusPetits* et *PlusGrands*, avec comme résultat respectifs *PlusPetitsTriés* et *PlusGrandsTriés*.
4) Concaténer *PlusPetitsTriés* avec *pivot* et *PlusGrandsTriés* pour obtenir le résultat du tri.

Cette description, liée au contexte de l'ouvrage dont elle est issue (un ouvrage sur la programmation en langage Prolog) donne en effet des indications précises sur l'implémentation de cet algorithme dans *ce* langage (la figure 1.2 présente l'implémentation de cet algorithme tiré du même ouvrage). Mais que dire du rapport entre cette description et l'implémentation de cet algorithme présenté par Brian Kernighan et Denis Ritchie [Kernighan78, page 87] donnée dans la figure 1.3 ?

A la lecture de ces programmes, on peut remarquer que : outre la distance entre la description algorithmique ci-dessus et son implémentation

```
partition([X|Fin], Pivot, [X|PlusPetits], PlusGrands) :-
X < Pivot, !, partition(Fin, Pivot, PlusPetits, PlusGrands).
partition([X|Fin], Pivot, PlusPetits, [X|PlusGrands]) :-
        partition(Fin, Pivot, PlusPetits, PlusGrands).
partition([], _, [], []).
quicksort([X|Fin], Resultat) :-
        !, partition(Fin, X, PlusPetits, PlusGrands),
        quicksort(PlusPetits, PlusPetitsTriés),
        quicksort(PlusGrands, PlusGrandsTriés),
        append(PlusPetitsTriés, [X|PlusGrandsTriés], Resultat).
```

Figure 1.2 Implémentation du QuickSort en Prolog

[24] On pourrait plutôt dire que le lien est rarement explicitement fait (mis à part dans les ouvrages techniques).

```
Void qsort(int v[], int gauche, int droite)
{
        int i, dernier ;
        void swap(int v[], int i, int j) ;

        if(gauche >= droite) return ;
        swap(v, gauche, (gauche + droite)/2) ;
        dernier = gauche ;
        for(i = gauche + 1 ; i <= droite ; i++)
           if(v[i] < v[gauche])
                 swap(v, ++dernier, i) ;
        swap(v, gauche, dernier) ;
        qsort(v, gauche, dernier-1) ;
        qsort(v, dernier+1, droite) ;
}
```

Figure 1.3 Implémentation du QuickSort en C

dans le langage C, des différences dans les techniques de programmation utilisées (forme complètement récursive ou plutôt itérative) distinguent également les deux programmes. Ainsi, même si ces deux programmes s'inspirent du même algorithme, leurs concepteurs le comprennent différemment, probablement fortement influencés par le *langage* de programmation qu'ils utilisent.

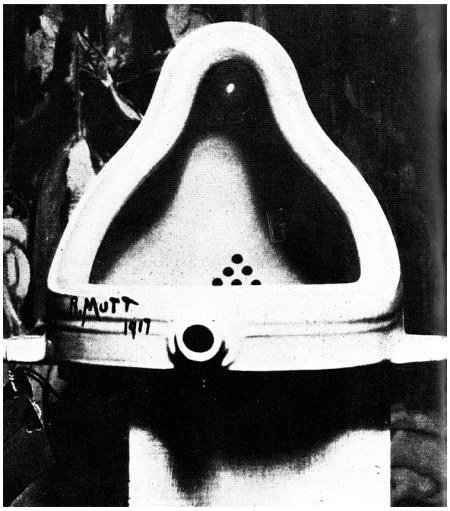
Certaines techniques de programmation s'inspirent de processus présents dans la nature, comme les algorithmes génétiques et la vie artificielle [Dumeur94], la programmation par agent [Abchiche99] ou encore les techniques de communication entre robots [Ali Cherif95]. Elles illustrent bien la distance entre les concepts appartenant au sens commun et leur transposition dans l'informatique. En effet, même si ces techniques de programmation s'inspirent de processus naturels, aucune ne prétend permettre une correspondance complète entre ces processus et leur représentation algorithmique. De plus, comme le montrent remarquablement les travaux de Bersini [], la construction des algorithmes les plus performants (souci majeur de l'informatique) passe souvent par la combinaison de plusieurs types d'algorithmiques, ajoutant une distance encore plus grande entre les référents alors pris en comptes.

Dès lors on peut se demander : *Que désigne un terme, un nom d'outil, une expression ou encore un algorithme ?* Et, au-delà, *Comment exprimer la (ou les) réalité(s) sous-jacente(s) de manière compréhensible ?*

II.1.1.2 Dans l'art

Un autre domaine où la question est pertinente est celui de la relation entre l'art et la perception ou la compréhension d'une œuvre. Cette relation est de même nature que celle de la perception de la dimension esthétique des représentations analogiques que nous proposons.

Des exemples apparaissent clairement dans le cinéma avec des réalisateurs comme Eisenstein dont le *montage attractif* est décrit par Ado Kyrou



La Fontaine de M. Duchamp "La Trahison des Images" de R. Magritte

Figure 1.4 *Sens* d'un objet ou d'une image

[Kyrou85, page 145] :

> Par attraction Eisenstein entendait choc direct produisant une impression presque physique sur le spectateur. Le *montage* devait unir ces attractions pour en tirer par leur rapprochement de nouveaux chocs émotifs. Le montage n'était plus logique, ni seulement symbolique mais obéissait à une nécessité sensible, expliquée philosophiquement et systématiquement (par des exemples tirés de l'écriture sanscrite et égyptienne) par Eisenstein.

Cette révolution dans la conception du montage, remarquablement illustrée par son film *Le cuirassé Potemkine*, nous montre qu'au-delà du récit, ou d'une suite d'images, notre sensibilité peut être touchée par la mise en scène et l'utilisation de formes métaphoriques comme la synecdoque[25] : « *A partir d'un élément bien choisi, on pouvait amener le spectateur à la conscience et au sentiment du tout* ».[26] [Fernandez75, page 92].

Avec les deux propositions de René Magritte (illustré par le « Ceci n'est pas une pipe » de *La Trahison des Images* (cf. figure 1.4)) et de Marcel Duchamp (« C'est le regardeur qui fait le tableau »), l'art se pose de façon nouvelle la question du statut de l'image.

Que devient alors la liaison entre représentation et représenté ? La pro-

[25] Umberto Eco [Eco88, page 52], à propos du processus mis en œuvre dans la compréhension de cette figure de rhétorique, pose la question : « *Ce processus est-il si différent de celui par lequel je passe d'un effet à sa cause ?* »

[26] Ceci pour illustrer la scène dans laquelle le médecin, jeté par-dessus bord par les révoltés, est représenté uniquement par son lorgnon, accroché au cordage par un fil de soie. « *Image de la myopie morale du docteur, son pince-nez désigne aussitôt pour le spectateur les caractères de la classe sociale à laquelle il appartient.* » [Fernendez75, page 92].

position de Marcel Duchamp et de ses *readymade*[27] (cf. figure 1.4), en tant que remise en cause fondamentale des notions d'*œuvre artistique* et de la perception des objets du quotidien, amène ainsi à un glissement permettant l'utilisation d'une forme, quelle qu'elle soit, hors de son contexte dans l'unique but de transmettre un message, d'interroger ou même d'interpeller. C'est ce glissement de sens qui se retrouve également dans le processus de la création artistique, comme le décrit par exemple David Smith [Smith77, page 33] dans l'introduction de *PYGMALION*[28] :

> Lorsqu'il est confronté à une situation nouvelle, l'artiste y applique un schéma développé lorsqu'il était confronté à d'autres situations, produisant une juxtaposition des deux. Il amène une manière de voir préétablie dans un nouveau contexte. Les contextes visuels des artistes sont ainsi différents du commun. Ils ont créé un ensemble de règles leur permettant de décrypter les images. Les innovations dans l'art sont le résultat de la modification de cet ensemble de règles.[29]

Selon lui, une création résulterait non seulement d'un *entraînement* à percevoir tel ou tel aspect dans une image, une œuvre, un programme, mais aussi de la *remise en cause* des codes habituellement établis pour les décrire[30]. Cette position ajoute une nouvelle clé de lecture aux questions que nous nous posons (et peut être une voie vers une solution) : *Comment remettre en cause les codes habituellement utilisés dans la description ou l'explication de tel ou tel aspect particulier afin, grâce à un éclairage différent, d'en montrer la pluralité d'interprétation, qui est de l'ordre de la complexité ?*

---

[27] Au-delà de l'interrogation sur l'*image*, les *readymade* de Duchamp questionnent sur la nature même de la création artistique, du statu de l'artiste, … : « *Le ready-made est emprunté au monde des objets usuels, tout faits (« ready »).*[…] *Mais l'artiste l'a choisi, et c'est ce choix souverain qui suffit, sans autre intervention matérielle, à faire de l'objet quelconque une œuvre d'art.* » [Pingaud, page 18]

[28] Rappelons que PYGMALION fut l'un des premiers systèmes de programmation visuelle et que, dans son ouvrage, l'auteur présente une remarquable étude de la « *pensée créatrice* » (creative thinking) et du rapport entre *création* et *changement de représentation*.

[29] « When faced with a new situation, the artist applies a schema developed in handling other situations, producing a juxtaposition of the two. He brings an established way of look at things into a new context. The visual contexts of artists are different from those of laymen. Artists have created a code of rules which they obey in looking at pictures. The innovations in art have been modifications of this code. »

[30] Rejoignant alors Duchamp et le concept de « C'est le regardeur qui fait l'œuvre d'art » ou de la problématique du *hasard* telle qu'elle apparaît, par exemple, dans l'analyse des *readymade* par Thierry de Duve [de Duve77]

II.1.1.3 Eléments de réponse

Ces questions ne sont pas l'apanage de l'informatique ou de l'art. Dans son introduction à la collection d'articles *Metaphor and Thought* dont il est l'éditeur, Andrew Orthony dit [Orthony93, page 2] :

> Il semble utile d'essayer de mettre en relation les deux approches alternatives de la métaphore (la métaphore comme une caractéristique essentielle de la créativité du langage et la métaphore comme déviations ou parasitages de l'usage normal) à une différence d'opinion plus fondamentale sur la relation entre le langage et le monde.[31]

La relation entre le langage et le monde ou encore, par extension, les relations entre l'expression et la pensée ou la perception et la compréhension seraient donc, selon Orthony[32], intimement liées au regard posé sur le rôle, ou le lien, entre un terme, une expression, l'objet désigné[33].

Cette relation métaphorique se trouve particulièrement mise en évidence dans les problèmes liés à la traduction d'expressions poétiques ou idiosyncratiques. Par exemple, Umberto Eco [Eco90, page 170] décrit la difficulté, pour un lecteur français, de comprendre l'analyse métaphorique de la phrase « *the rose melted* » proposée par Levin [Levin77]. En effet, dit-il, la traduction littérale de cette phrase (« *la rose a fondu* ») occulte une grande partie du sens du verbe *to melt* ayant également des connotations de chagrin, d'évanescence, de dissolution de quelque chose qui passerait en se dissipant d'un état à un autre[34].

La question revient alors d'elle-même : *Quelle interprétation donnons-nous aux termes, à l'échange présent dans un dialogue ?*

La démarche que nous avons mise en œuvre afin d'élaborer notre mé-

---

[31] « […] It seems useful to attempt to relate two alternative approaches to metaphor – metaphor as an essential characteristic of the creativity of language, and metaphor as deviant and parasitic upon normal usage – to a more fundamental and pervasive difference of opinion about the relationship between language and the world. »

[32] Cette opposition décrite par Orthony est à replacer dans son contexte : l'auteur met en opposition une conception *constructiviste* (faisant référence entre autres à Kant, Chomsky ou Lévi-Strauss), dans laquelle les métaphores jouent un rôle important, à une conception *non-constructiviste* (faisant alors référence à Russell ou Wittgenstein).

[33] La signification du terme *métaphore*, donnée par le Petit Robert, est : « *Figure de rhétorique, procédé de langage qui consiste dans un transfert de sens (terme concret dans un contexte abstrait) par substitution analogique.* », définition appelant alors directement celle du terme *analogie* : « *Ressemblance établie par l'imagination (souvent consacrée dans le langage par les diverses acceptions d'un même mot) entre deux ou plusieurs objets de pensée essentiellement différents.* »

[34] Umberto Eco souligne que ces différentes connotations sont dans le dictionnaire.

thode de construction de représentations analogiques de programmes, d'implémenter le système Zeugma basé sur cette méthode ou d'élaborer les représentations analogiques de programmes elles-mêmes s'inspire de cette réflexion sur la question de l'interprétation donnée à une image ou à une représentation. Afin de guider cette démarche, nous avons choisi de puiser dans les théories relatives aux métaphores (principalement celle de George Lakoff), aux analogies (de Derdre Gentner et Douglas Hofstadter) ainsi que dans des applications concrètes de ces théories (avec le *Structure Mapping Engine* [Gentner87] et CopyCat [Mitchell90]).

II.1.2 Représentation analogique de programmes

II.1.2.1 Précision sur les termes

Le titre de notre mémoire inclut : *environnement de programmation analogique*. L'utilisation du terme *analogie* demande nécessairement des précisions. En premier lieu, et comme le montre bien Orthony cité plus haut, nous devons préciser quelle est notre approche de ce domaine[35].

Nous situant dans une approche qu'il définit comme *constructiviste*, nous prenons alors, comme définition générale des métaphores, celle exposée par Lakoff et Johnson dans leur ouvrage *Metaphor we live by* [Lakoff80] :

> La métaphore est une figure de rhétorique dont l'essence est la compréhension et l'expérience d'une sorte de chose par les termes d'une autre.[36]

Mais Lakoff va plus loin en exposant [Lakoff93] son *Principe d'Invariance* :

> La mise en relation métaphorique préserve la topologie cognitive du domaine d'origine[37] d'une manière consistante avec la structure inhérente du domaine cible[38].[39]

Ce principe précise en effet la manière dont le transfert s'effectue entre les domaines cognitifs en jeu dans une métaphore. Ceci amène également Georges Lakoff à poser comme corollaire que la limitation induite par ce principe est que la structure inhérente du *domaine cible* limite automatique-

[35] Comme nous le verrons dans la section suivante, l'absence de précision de la part des concepteurs des systèmes de visualisation de programmes amène à des critiques quant à la justification même des visualisations.

[36] Metaphor is a rhetoric figure, whose essence is understanding and experiencing one kind of thing in terms of another.

[37] *Source domain*

[38] *Target domain*

[39] Metaphorical mappings preserve the cognitive topology of the source domain, in a way consistent with the inherent structure of the target domain.

ment les liens possibles entre les deux domaines.

Pour reprendre les termes de Tiziana Catarci [Catarci95], appliquant les métaphores de Georges Lakoff à la visualisation du contenu des bases de données, c'est grâce aux métaphores que nous pouvons aller vers de nouveaux concepts en partant de concepts familiers, incorporant ainsi de nouvelles connaissances à d'anciennes.

Notre méthode, comme nous allons le voir en détail dans le chapitre suivant, a pour finalité de mettre en relation des abstractions sur les programmes et des représentations graphiques animées.

Toutefois les programmes, même sous forme d'abstractions, ne constituent pas à eux seuls l'unique domaine de l'informatique nécessitant une approche métaphorique. De nombreux travaux existent aujourd'hui sur la conception d'interfaces utilisateurs[40] (également nommées *interfaces homme – machine*) et certains travaux s'inspirent directement de la notion de métaphore[41]. Notre réflexion nous ayant conduit à prendre comme domaine d'origine une étude sur les programmes, pose la question suivante : *Est-ce que la distinction entre environnement de programmation et interface utilisateur a un sens ?* ou bien la différence n'est-elle pas uniquement liée au public auquel ces outils s'adressent ? En effet, les environnements de programmation doivent aujourd'hui permettre aux programmeurs non seulement de construire des programmes utilisant les différents médias disponibles mais également, et c'est là la principale motivation de notre recherche, de les intégrer comme des éléments à part entière de *l'environnement de programmation* lui-même.

A ce point nous devons préciser le domaine cible dont la finalité est de fournir un éclairage différent, utilisant des *concepts familiers*, sur le domaine d'origine. Si nos métaphores se limitent souvent à l'utilisation d'images ou de représentations graphiques comme média final dans la construction des représentations de programmes, ceci vient de la limitation actuelle des technologies offrant des facilités graphiques. Même si cette justification est valable, nous aimerions la resituer dans le contexte plus large de l'évolution des technologies ou des méthodes d'interaction laissant entrevoir une évolution vers des médias multi-sensoriels, comme l'ont évoqué Brad Myers (cf. figure 1.5) ou plus récemment Andries van Dam (dans sa communication à EuroGraphics'98 dans laquelle il présente des voies de recherche pour l'ère post-WIMP[42] [vanDam98]). Ceci précise le média : nous allons construire des représentations graphiques en trois dimensions, étape vers une intégration de nouveaux médias comme le son et une interaction différente comme la réalité

[40] Voir, à ce sujet, le bref historique de l'évolution des technologies de l'interaction homme – machine de Brad Myers [Myers96].

[41] La plus connue reste celle de la métaphore de la table de travail ayant servi aux membres de l'équipe de recherche de Xerox à construire le modèle des interfaces encore utilisées aujourd'hui. Il est également important de noter l'influence du travail de David Smith, évoqué plus haut, dans l'élaboration de cette interface.

[42] WIMP : Windows, Icônes, Menus, Point-and-Click.

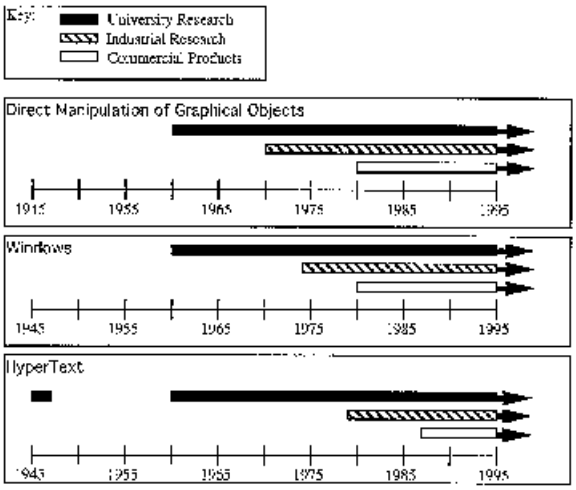


Figure 1.5 Résumé de l'évolution des techniques d'interaction [Myers96, page 795]

virtuelle. Outre ce média, les représentations que nous allons construire visent à utiliser des *concepts familiers*. Ce choix est motivé, comme le verrons, par la remise en cause des méthodes de construction de représentations visuelles actuelles ainsi que par le modèle cognitif sous-jacent à notre méthode.

II.1.2.2 Critiques sur la visualisation de programmes et nécessité d'un modèle cognitif

La visualisation de programmes, ou la programmation visuelle, est un domaine en constante évolution. Des critiques de fond quant aux directions prises par les différentes équipes de recherche apparaissent toutefois. On peut distinguer deux types de critiques :

1) La justification cognitive de ces systèmes.
2) La mise en garde par rapport aux résultats.

Détaillons maintenant chacune de ces critiques.

1) Critiques sur la justification cognitive des systèmes de visualisation de programmes

Nous pensons qu'il est possible de situer l'origine de cette critique dans l'article de Allan Blackwell [Blackwell96b] dans lequel il passe en revue les arguments *métacognitifs* utilisés par les concepteurs de systèmes de programmation visuelle. Il conclut en disant :

> La plupart des arguments métacognitifs trouvés dans cette étude trouvent leurs fondements logiques dans l'introspection, les théories cognitives ou la psychologie. Dans certains cas ils sont pertinents, mais dans d'autres il existe des évidences logiques ou empiriques les rendant

inappropriée.[43]

Il paraît important, dans l'élaboration d'un système reliant des programmes et des représentations graphiques, de garder à l'esprit la nécessité d'un modèle cognitif permettant de construire des représentations que nous qualifierons d'*efficaces.* Ce débat a pris une autre dimension par l'article de Marian Petre, Allan Blackwell et Thomas Green [Petre97] posant en préambule une question sur ce qui semblait pourtant communément admis : la définition de la *Visualisation de Programmes* donnée par Brad Myers [Myers90] :

> Dans la visualisation de programmes, les programmes sont spécifiés de manière conventionnelle, textuelle, et le graphisme est utilisé pour illustrer certains aspects du programme ou de son exécution.[44]

La critique de Petre et al., qui nous semble fondée, est la suivante : cette définition est trop restrictive. Pourquoi ne pas construire des visualisations de programmes issus de systèmes de programmation visuelle (dans lesquels les programmes sont spécifiés en utilisant des objets graphiques) ou encore des visualisations de programmes textuels par des mises en forme typographiques (comme par exemple SeeSoft [Baeker90]) ?

Cette critique propose également une approche différente du domaine en posant des questions qui, malheureusement, apparaissent rarement dans les taxonomies existantes. Ces questions, d'un ordre cognitif, résument notre démarche dans l'élaboration de notre méthode de construction de représentations analogiques de programmes et rejoignent les interrogations présentées dans la section précédente. Deux questions nous semblent essentielles :

> Quel modèle représentons-nous ?[45] [Petre97, page 464]
>
> Comment réconcilier des conventions en opposition ? et comment les conventions se construisent-elles ?[46] [Petre97, page 474]

La première question interroge ce que nous nommons le domaine d'origine et désigne les programmes au sens large[47]. En se basant sur

---

[43] « Most of the metacognitive beliefs found in this survey have a logical basis in introspection, cognitive theory, or folk psychology. In some cases, they are well-founded, but in others there is logical or empirical evidence making them inappropriate. »

[44] « In Program Visualization, the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspects of the program or its run-time execution. »

[45] « What model are we representing? »

[46] « How can we reconcile warring conventions? And how do conventions arise? »

[47] *Programmes au sens large* désigne ici non seulement les programmes eux-mêmes mais aussi les abstractions issues d'études sur les programmes ou encore les algorithmes sous-jacents.

l'exemple des structures de *plans* telles qu'elles apparaissent dans les résultats des expériences de Soloway et Ehrlich [Soloway84] et de leurs racines qualifiées de *cognitives,* la question est alors : *quel niveau d'abstraction sur les programmes est représenté et pourquoi représenter tel niveau plutôt que tel autre ?* Nous ajoutons alors, comme nous le montrerons dans nos exemples de représentations analogiques de programmes, que pour réaliser une représentation effective de tâches cognitives complexes (comme la reconnaissance d'algorithmes ou l'aide à la lecture et la compréhension de programmes), les représentations doivent regrouper plusieurs aspects, parfois très différents, des programmes.

La seconde question s'adresse à ce que nous considérons comme domaine cible : les représentations graphiques. Elle met en exergue les problèmes issus du choix de telle ou telle représentation pouvant, suivant le lecteur, prendre des significations très éloignées, parfois conflictuelles. Cette question est la principale motivation de l'élaboration de notre méthode. Elle demande en effet que les liens construits entre l'objet représenté et la représentation soient clairement exprimés, répondent à une méthode les rendant *intuitivement* compréhensibles.

2) Mise en garde par rapport aux résultats

La seconde critique, qui s'applique plus précisément aux systèmes de programmation visuelle vaut également dans la construction de représentations graphiques de programmes : elle met l'accent sur la confusion possible entre la métaphore graphique utilisée comme support au langage visuel et le domaine applicatif des programmes élaborés en utilisant ce langage. Par exemple, Allan Blackwell [Blackwell96a] critique le langage de programmation visuel LabView[48], utilisant la métaphore de circuits électroniques, en disant :

> Considérez ce qui pourrait arriver dans LabView si un programmeur développait un programme de simulation de circuit. La confusion entre les câbles et les composants du domaine du programme en construction et ceux de LabView rendrait la tâche impossible.[49]

Nous adhérons à cette remarque qui interroge le concepteur de représentations analogiques sur la *distance* existant entre la représentation choisie et le domaine applicatif du programme représenté. Toutefois, comme le précise Stella Vosniadou [Vosniadou89], le problème doit aussi être considéré sous l'angle de l'*utilisation* qui sera faite de l'analogie. Partant d'une critique de la qualification donnée par Derdre Gentner des analogies intra-domaines

[48] LabView est un produit de National Instruments Corporation.

[49] « Consider what would happen in LabView, if the programmer were developing a circuit simulation program. Confusion between the wires and components in their problem domain and the wires and components in the LabView device domain would make the task impossible. »

comme de simples similarités (classification également à la base de la critique de Allan Blackwell à propos de LabView et que nous présentons à la page 40 de ce mémoire), elle distingue la caractérisation des analogies (ou leur reconnaissance, leur détection automatique ou non) d'un raisonnement utilisant des analogies dans lequel la promiscuité des domaines n'est pas rédhibitoire à condition que [Vosniadou89, page 416] :

> Le raisonnement analogique peut être employé entre deux éléments appartenants à la même catégorie fondamentale à condition qu'il mette en œuvre le transfert d'une structure explicative d'un élément à l'autre.[50]

La pertinence d'une analogie ne dépend donc pas du choix des domaines d'origine et cible mais du partage de structures sous-jacentes. Un exemple de cette vision des analogies serait de considérer les différentes implémentations possibles d'un algorithme comme ses différentes représentations analogiques. Le principal étant que ces différents programmes utilisent effectivement la démarche[51] décrite par l'algorithme. La recherche et la construction de nouvelles représentations analogiques d'un algorithme devront ainsi prendre en compte la reconnaissance des différentes étapes qui le caractérisent. Cette vision des analogies comme partage de structures sans limitation par rapport aux domaines mis en relation, ouvre de nouveaux champs de représentations qui ne seront pas limitées par un domaine particulier tant que la contrainte *structurelle* sera respectée.

Ces critiques de la programmation visuelle, également pertinentes pour la visualisation de programmes ou d'algorithmes, font apparaître la nécessité d'une méthode basée sur une considération cognitive des liens unissant les objets manipulés (les programmes *et* les représentations). Cette méthode permet alors une référence aux analogies, telles qu'elles peuvent être décrites dans les travaux que nous avons cités, et aux liens qu'elles tissent avec des activités comme l'apprentissage, la transmission de messages ou encore la représentation même de l'expression d'idées. De plus, comme nous l'avons vu avec l'exemple d'Eisenstein, des formes de rhétoriques, intimement liées aux analogies et aux métaphores, peuvent constituer la source d'idées nouvelles dans la construction de représentations analogiques de programmes.

[50] « Analogical reasoning can be employed between any two items that belong to the same fundamental category if it involves transferring an explanatory structure from one item to the other. »

[51] Considéré alors comme liant *structurel*.

II.2 Méthode de construction des représentations analogiques de programmes

> *En Occident, on se préoccupe davantage du contenu et de la signification du modèle que de sa construction, de sa structure, de son fonctionnement et des objectifs qu'il est supposé atteindre.*
>
> *Edward Hall*
>
> *[Hall79, pages 19-20]*

II.2.1 Introduction

Afin d'élaborer notre méthode de construction de représentations analogiques de programmes, nous avons besoin de considérer les analogies sous deux angles complémentaires :

1) le premier, que nous qualifierons de *formel*, devra nous permettre de construire une représentation des domaines d'origine et cible en jeu dans la visualisation de programmes afin de les utiliser comme base de la construction de représentations *analogiques*,
2) le second, que nous qualifierons de *génératif,* permettra de relier les descriptions formelles des deux domaines et d'appliquer ces descriptions d'analogies à des cas concrets.

Comme nous allons le voir, la *Structure Mapping Theory* élaborée par Derde Gentner et le concept de *High Level Perception* décrit par Douglas Hofstadter donnent des éléments de réponse et c'est en nous inspirant de ces théories, et de leurs applications, que nous avons élaboré notre méthode.

II.2.1.1 Modèle pour la compréhension des analogies

La *Structure Mapping Theory* de Derdre Gentner, élaborée en vue de fournir un modèle aux mécanismes en jeu dans la compréhension des analogies, propose une représentation descriptive des domaines en jeu. Dans son article *The mechanisms of analogical learning*, elle [Gentner89, page 201] dit :

> L'idée centrale dans la *corrélation de structures* est qu'une analogie est un *transfert* de connaissances d'un domaine (d'origine) vers un autre (cible) impliquant qu'un système de relations présent dans les objets d'origine sera également présent dans les objets cible. [52]

[52] « The central idea in structure-mapping is that an analogy is a mapping of knowledge from one domain (the base) into another (the target), which conveys that a system of relations that holds among the base objects also holds among the target objects. »

Plusieurs questions sont alors à préciser :

1) Que signifie l'utilisation du terme *knowledge* (connaissance) ?
2) Que désigne ce *system of relations* (système, ensemble de relations) ?
3) Qu'implique cette notion de *mapping* (transfert, correspondance, corrélation) ?

C'est en effet autour de ces trois termes que se construit sa *Structure Mapping Theory*, que nous utilisons comme base cognitive de notre considération des domaines d'origine et cible dans la construction de représentations analogiques de programmes. Afin de donner les précisions sur ces termes, nous nous baserons sur un exemple décrivant le fonctionnement du *Structure Mapping Engine* (SME) [Gentner87][Falkenhainer89], construit sur la base de cette théorie et simulant le processus de corrélation de structures entre des objets appartenant à des domaines différents.

II.2.1.1.1 Présentation du Structure Mapping Engine (SME)

L'exemple présenté par Derdre Gentner dans [Gentner89] est le suivant : il s'agit de la description de la comparaison entre la circulation d'eau et la circulation de chaleur (cf. figure 2.1). L'interprétation de l'analogie, ou l'essence de la connaissance qu'elle véhicule, est la relation de correspondance entre le processus de transfert de la *pression* et de la *température*. C'est à ce premier niveau que se situe la notion de *connaissance*, relativisée toutefois, comme elle le précise, par le manque de précision des correspondances entre les exemples donnés.

Gentner définit alors une représentation des différents objets présents dans l'analogie par une série d'*attributs* et de *relations*[53].

- Les *attributs* désignent des caractéristiques physiques comme la pression de l'eau dans la carafe ou dans le tube à essai, ou encore le fait que de l'eau passe de l'un vers l'autre. Ces attributs seront alors notés : PRESSION(carafe), PRESSION(tube-à-essai) et CIRCULATION(eau, tuyau, carafe, tube-à-essai).
- Les *relations* désignent des propriétés reliant plusieurs attributs. Par exemple, une relation de grandeur existe entre la pression présente dans la carafe et celle présente dans le tube à essai. Un autre exemple est celui de la relation de causalité entre la présence de cette différence de pression et la présence de la circulation d'eau. Ces différentes relations seront alors notées : PLUSGRAND[PRESSION(carafe), PRESSION(tube-à-essai)] et CAUSE{PLUSGRAND[PRESSION(carafe), PRESSION(tube-à-essai)], CIRCULATION(eau, tuyau, carafe, tube-à-essai)}.

[53] Déterminant alors le *Système de relations*.

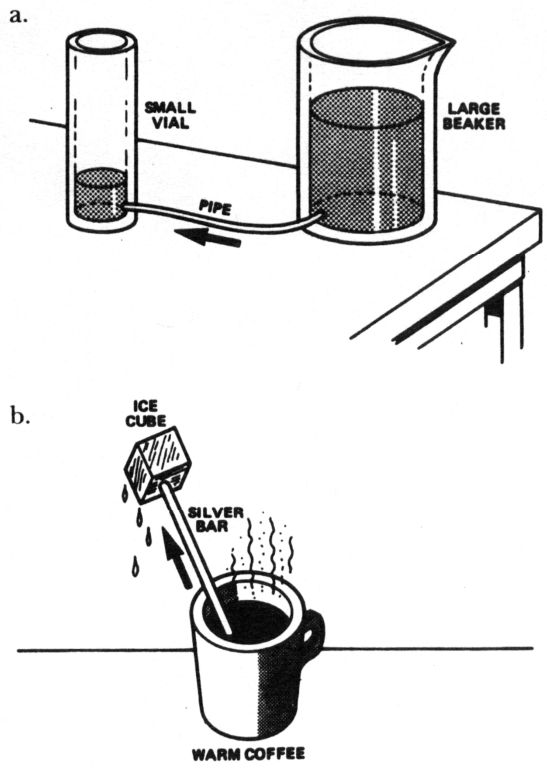


Figure 2.1 Analogie entre circulation d'eau et de chaleur, [Gentner89, page 202] adapté de [Buckley79, pages 12 – 25].

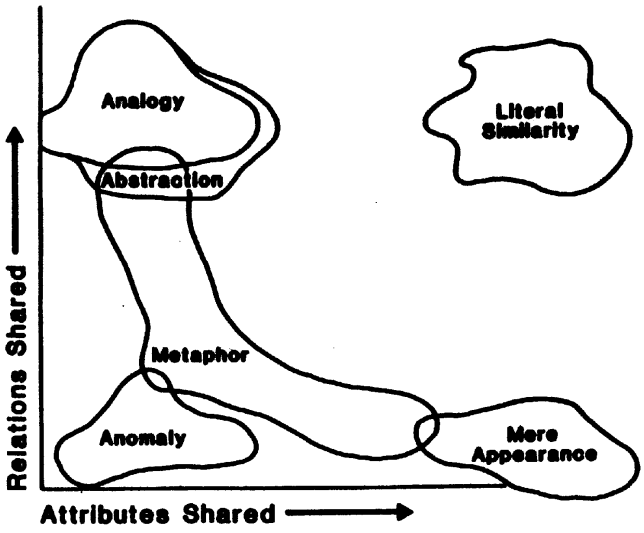


Figure 2.2 ***Espace des similitudes*** **: classification des similitudes [Gentner89, page 207].**

A partir de ces définitions de relations et d'attributs, elle définit une classification des différents types de similitudes suivant le type de prédicats

partagés par différents objets (cf. figure 2.2)[54]. La particularité de cette classification, que nous utilisons dans notre méthode de création de représentations analogiques de programmes, est le rapport présenté entre *abstractions* et *analogies*. Comme Gentner, nous pensons que les activités liées à l'*abstraction*, relativement répandues en ingénierie de l'informatique, offrent une base de travail quant à la transposition de tels modèles dans l'étude de programmes.

Le travail effectué par le système SME est alors, à partir d'une description des différentes relations et attributs de l'origine et du but de l'analogie, de calculer les liens possibles, d'en construire les *correspondances*.

Pour reprendre l'exemple décrit plus haut, l'ensemble de relations et d'attributs présentés dans la figure 2.3 va être fourni à SME.

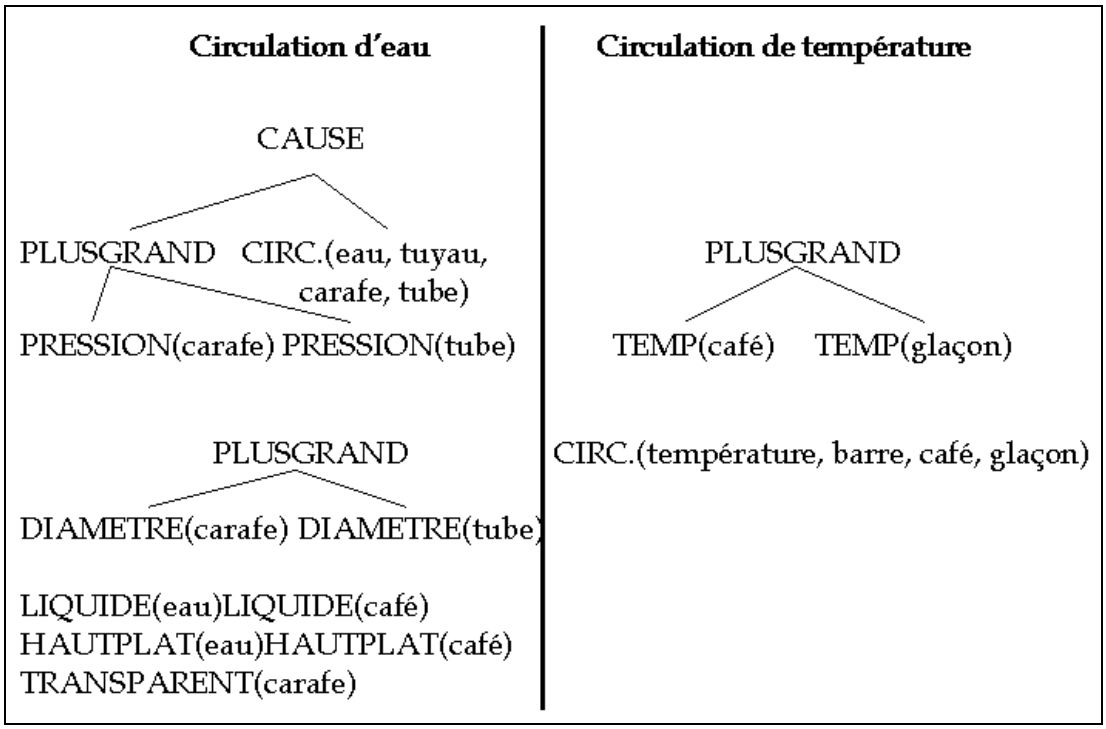


Figure 2.3 Représentation de l'eau et de la température donnée à SME

Des relations hors du contexte de l'analogie, par exemple l'attribut HAUTPLAT ou la relation PLUSGRAND[DIAMETRE(carafe), DIAMETRE(tube)], ont intentionnellement été fournies à SME dans le but de symboliser l'approche approximative que peuvent avoir des personnes dans l'interprétation des analogies.

A partir de ces relations et attributs, SME effectue trois passes :

1) déterminer des correspondances locales entre attributs ou relations (par comparaison du nom par exemple),
2) la construction d'une correspondance globale, alors synonyme de

[54] C'est cette classification, qualifiant de simple ressemblance ou de similitude littérale le partage d'attributs, que critique Stella Vosniadou, comme nous l'avons cité plus haut, quant à la construction d'analogies intra-domaines.

corrélation structurelle entre les deux domaines,

3) une évaluation visant à discriminer les différents candidats possibles selon des critères comme le nombre de relations ou d'attributs partagés.

La conclusion alors fournie par SME est :

CAUSE{ PLUSGRAND[TEMP(café), TEMP(glaçon)],
CIRC.(température, barre, café, glaçon)}.

II.2.1.1.2 Application à la visualisation de programmes

La méthode décrite par le SME permet de détecter, même à partir de représentations incomplètes, des relations décrivant des phénomènes appartenant à des domaines distincts, si une analogie est présente.

Cette démarche peut aisément être transposée dans le domaine de la visualisation de programmes. En effet, considérant les programmes comme domaine d'origine et les visualisations comme domaine cible, elle permet alors de confirmer l'étendue des liens les unissant. Par exemple, considérant la représentation des listes du langage Lisp sous la forme de *boîtes de cons* (la figure 2.4 présente un exemple de cette représentation), la démarche du SME devrait permettre de confirmer l'équivalence analogique entre les deux représentations. Pour cela, et pour que la démarche soit applicable, il faut en premier lieu définir ce que signifie les notions de *relations* et d'*attributs* de telle manière qu'elles soient identifiables dans les deux domaines. Une représentation intuitive de ces relations et attributs :

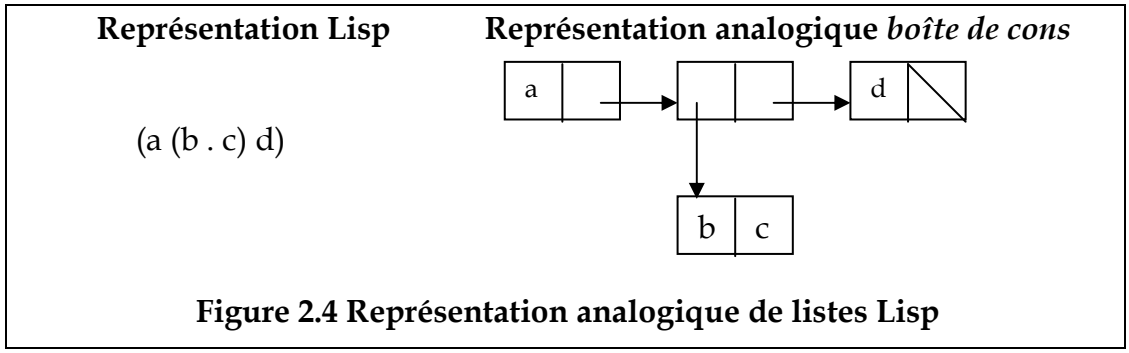


Figure 2.4 Représentation analogique de listes Lisp

Pour la représentation sous forme de listes Lisp :

CONTIENT(liste1, a, CONTIENT(liste2, CONTIENT(liste3, a, b),
CONTIENT(liste4, d, nil))

Pour la représentation sous forme de boîte de cons :

CONTIENT(boite1, a, pointeur-vers-boite2)
CONTIENT(boite2, pointeur-vers-boite3, pointeur-vers-boite4)
CONTIENT(boite3, b, c)
CONTIENT(boite4, d, rien)

Pour que SME puisse identifier, il est nécessaire de reconstruire l'équivalence entre les pointeurs graphiques (les flèches) et les pointeurs im-

plicites de la représentation Lisp. Ces derniers sont désignés par le terme *implicite* du fait de leur présence effective dans la représentation interne des listes Lisp mais cachées dans leur représentation externe. Pour les listes Lisp, la représentation permettant de conclure à une équivalence analogique entre les deux représentations serait :

CONTIENT(liste1, a, pointeur-vers-liste2)
CONTIENT(liste2, pointeur-vers-liste3, pointeur-vers-liste4)
CONTIENT(liste3, b, c)
CONTIENT(liste4, d, rien)

A partir de ces représentations, SME serait certainement capable de conclure que les boîtes sont des représentations analogiques des listes.

Cet exemple à mis en évidence deux types de questions :

1) quelle représentation des programmes et des représentations construire afin de pouvoir appliquer une démarche analogique que nous qualifierons de *stricte* dans le sens ou elle répond à des critères d'équivalence des relations entre les domaines concernés ?
2) SME étant adapté à la vérification, ou la détection, de la présence d'analogies entre différents domaines, comment passer à la *génération* d'analogies, permettant alors de *créer* des représentations analogiques de programmes ?

Nous avons donc besoin d'un modèle permettant d'une part de se baser sur ces notions de *connaissance*, de *systèmes de relations* et de *transfert* présentes dans la théorie développée par Derdre Gentner et d'autres part de pouvoir utiliser ces notions afin de passer dans un mode *génératif* permettant, à partir de la description d'une relation analogique, de *construire* des représentations de programmes.

II.2.1.2 Modèle pour la génération d'analogie

Cette nouvelle vision des analogies ayant pour finalité la *génération* d'analogies plus que leur *reconnaissance*, est l'objet même du concept de *High Level Perception*[55] (Perception de Haut Niveau) développé par Douglas Hofstadter et son équipe [Chamlers95].

Avant de présenter un exemple de génération d'analogies par le système CopyCat, développé à partir de ce concept par Melanie Mitchell [Mitchell90] [Hofstadter95], nous allons présenter les idées qui, de notre point de vue, ont amené à son élaboration. En effet, ce concept de perception de haut niveau apparaît déjà chez Hofstadter dans son ouvrage *Gödel, Escher, Bach* [Hofstadter85] sous une forme différente. Il y étudie alors les implications issues de l'étude des problèmes de Bongard [Bongard70] dont nous présentons ici deux exemples (cf. figure 2.5). La problématique générale est de déterminer les différences entre les images présentées à gauche et celles présen-

[55] Perception de Haut Niveau.

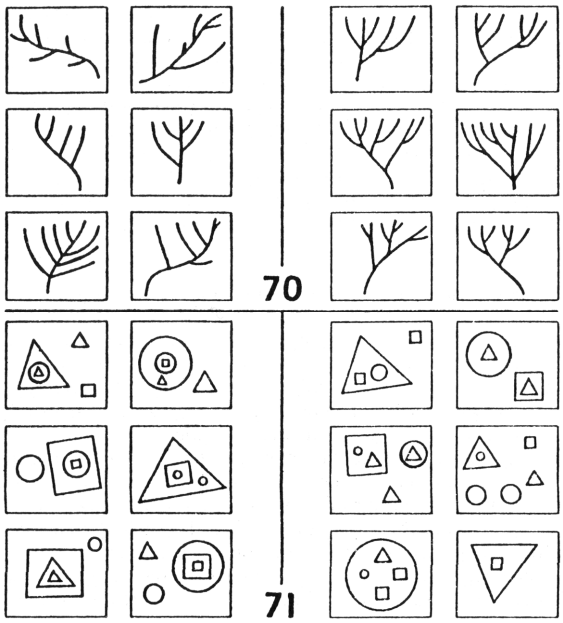


Figure 2.5 Problème 70 et 71 de Bongard [Bongard70]

tées à droite. Un traitement automatique de ces problèmes, selon Douglas Hofstadter, amènerait à considérer une perception d'un niveau supérieur à une simple observation, mettant alors en jeu ce qu'il nomme des *squelettes conceptuels* :

> Un squelette conceptuel est comme un ensemble de caractéristiques constantes (par opposition à des paramètres ou des variables), c'est-à-dire de caractéristiques qui ne devraient pas être modifiées dans un ralenti conditionnel ou une opération de correspondance. Dénué de tout paramètre et variable, il peut servir de cœur invariable à plusieurs idées différentes. [Hofstadter85, page 752]

Ce problème de la représentation et de l'abstraction est au cœur du développement de la perception de haut niveau ainsi que de l'élaboration du système CopyCat. De plus, c'est à partir de cette idée de *squelette conceptuel*, auquel nous avons joint les notions de relations et d'attributs de Derdre Gentner, que nous avons élaboré nos représentations des domaines d'origine et cible dans notre méthode de construction de représentations analogiques (nous présenterons dans la section suivante ce dernier point en détail).

Cette théorie de la perception de haut niveau peut être décrite de la manière suivante :

> La perception de Haut Niveau commence à un niveau de traitement [en terme de perception cognitive] où les concepts prennent un rôle important.[56]

Pour préciser ce terme de *concept*, Chamlers et al. décrivent différents exemples suivant ce qu'ils nomment le *spectre d'application de la perception de*

[56] « High-level perception begins at that level of processing where concepts begin to play an important role. »

haut niveau. Par exemple, la reconnaissance d'objet (le processus d'association entre une image physique et sa dénomination) serait le plus bas niveau de ce type de perception. Puis viennent des perceptions plus *abstraites* comme la capacité à percevoir des *relations* de type spatial (inclusion, dessus, dessous, etc.) pour aboutir à des abstractions de haut et très haut niveau (de l'appartenance à une famille politique jusqu'à la perception des relations présentes dans une histoire d'amour ou une guerre).

A partir de ce concept, ils développent l'idée qu'un système travaillant sur les analogies devrait être capable en premier lieu de déduire les relations et attributs présents dans une situation (faisant alors preuve de *perception d'un haut niveau*) puis, dans un second temps, de les vérifier en les appliquant à une nouvelle situation dans laquelle, par exemple, uniquement la moitié des relations décrivant une situation a été spécifiée. Les relations ainsi déduites ne seraient plus liées à une situation spécifique mais décriraient une abstraction sur l'analogie, consistant en un ensemble de relations comparables à un squelette conceptuel, et c'est celui-ci qu'il serait alors possible de transposer dans une nouvelle situation. A partir de cette vision de l'étude des analogies, ils critiquent (entre autres) le système SME considérant que si ce système fonctionne, c'est en grande partie (si ce n'est uniquement) grâce au fait qu'on lui a fourni une description du problème (c'est-à-dire l'ensemble des *relations* et des *attributs*) permettant la déduction de l'analogie[57].

Le système CopyCat, élaboré par Melanie Mitchell (op cit.) sur cette base, résout le type de problème suivant :

1. supposez que la chaîne de caractères ***abc*** ait été changée en ***abd*** ; comment pourriez-vous changer la chaîne ***ijk*** « de la même manière » ?
2. ou encore, supposez que la chaîne ***aabc*** ait été changée en ***aabd*** ; comment pourriez vous changer la chaîne ***ijkk*** « de la même manière » ?

Si nous avons cité deux exemples de questions auxquelles ce système est capable de donner une réponse, c'est pour illustrer les différents aspects des relations et attributs sur lesquels il travaille. En effet, si la réponse ***ikl*** semble naturelle, due à une observation de la transformation *la lettre la plus à droite a été remplacée par son successeur alphabétique* ainsi que de la *régularité* présente à la fois dans ***abc*** et ***ijk***, la réponse à la seconde question demande une analyse plus fine. Comme l'explique Douglas Hofstadter et Melanie Mitchell dans [Hofstadter93], plusieurs réponses sont possibles :

- en appliquant le même raisonnement que pour la question précédente, on pourrait proposer ***ijkl***,
- en considérant les deux ***k*** comme une unité insécable on pourrait alors proposer ***ijll***,

[57] Cf. note 6 page 5 pour notre point de vue sur ce débat.

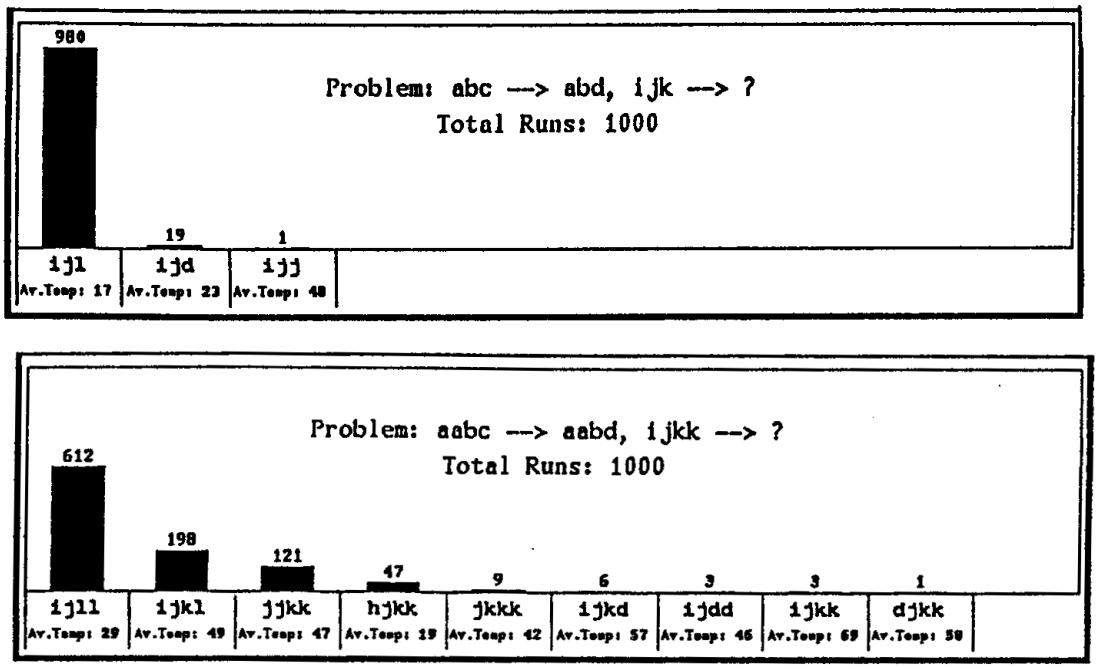


Figure 2.6 Réponses de Copycat aux deux questions

- en observant que deux régularités apparaissent, l'une d'un côté : ***aa***, et l'autre de l'autre : ***kk***, et en considérant alors ***ijkk*** comme une forme *inversée*, on pourrait proposer ***jjkk***,
- finalement, considérant les deux liens croisés de la précédente proposition (***aa*** ⇔ ***kk*** et ***c*** ⇔ ***i***) et le fait que l'inversion est une invitation à lire la seconde proposition en ordre décroissant, le remplacement de la lettre ***i*** ne serait plus alors par son successeur mais par son prédécesseur, aboutissant à ***hjkk***.

Les relations en jeu dépassent ainsi les simples notions de prédécesseur ou de successeur mais, suivant les cas, peuvent aboutir à des combinaisons d'une grande complexité. C'est cette complexité qui demande alors une perception abstraite des relations et qui symbolise le concept de perception de haut niveau décrit plus haut. Les résultats de CopyCat ne sont finalement pas des certitudes quant à l'interprétation de telle ou telle situation mais une description des possibilités déduites des situations présentées (la figure 2.6 présente les réponses données aux deux questions précédemment citées).

L'application telle quelle de CopyCat à la génération de représentations analogiques de programmes ne semble pas réalisable. En effet, le problème soulevé lors de la présentation de SME reste entier : il ne s'agit pas de construire une représentation analogique à partir d'un exemple existant mais de *construire* de nouvelles analogies entre des programmes et des images. Dans l'élaboration d'un modèle pouvant résoudre ce problème, l'apport de CopyCat et de l'étude sur les analogies ou les différents niveaux de perception fut de fournir une approche cognitive permettant de « prendre de la distance » par rapport aux programmes ou au représentations graphiques afin de pouvoir construire une approche *structurelle* de ces domaines, base essentielle à

la prise en compte des conclusions de Derdre Gentner aussi bien que de Douglas Hofstadter.

II.2.2 Présentation de la méthode de construction de représentations analogiques de programmes

Notre méthode vise à résoudre la question suivante : *Comment procéder afin de construire une représentation graphique (ou sous tout autre forme) de programme répondant à des contraintes analogiques ?*

II.2.2.1 Introduction : Précisions sur les domaines d'origine et cible

Afin de répondre à cette question, nous proposons une approche que nous nommons : *l'approche structurelle*[58]. Nous allons maintenant définir cette approche par rapport aux deux angles correspondant aux domaines d'application de notre méthode : les programmes et les représentations graphiques.

a) le domaine d'origine : *points de vue* sur les programmes

Les programmes, quel que soit le langage de programmation utilisé, sont des textes structurés. Ces structures sont de différents types suivant qu'elles sont la représentation d'une organisation interne, issue du langage de programmation, ou celle d'une abstraction, résultat d'une étude sur les programmes eux-mêmes.

De plus, les structures issues d'une organisation interne des programmes sont elles-mêmes différenciables par le contexte qu'elles décrivent :

- qu'il s'agisse d'une structure indiquant, syntaxiquement, un transfert de contrôle ou un échange de données entre une partie et une autre du programme (communément nommés les flots de contrôle et de données),
- qu'il s'agisse d'une structure indiquant un transfert de contrôle ou de données effectif entre différentes parties du programme, détectable alors uniquement au cours de l'exécution du programme.

Les structures issues d'une étude sur les programmes existent en très grand nombre et constituent toujours un domaine de recherche actif. Rappelons l'exemple des *clichés* du *Programmer's Apprentice*[59] [Rich81] ou encore des *signatures* [Balmas95b] qui, chacun à sa manière, construit automatiquement une représentation abstraite des éléments fonctionnels présents dans des programmes, représentations intrinsèquement structurées.

---

[58] En référence à la fois aux notions de relations et d'attributs de Gentner et au squelette structurel de Hofstadter.

[59] Ils se basent sur la démonstration de la présence de clichés psychologiques dans l'activité de programmation présentée par Soloway et Ehrlich [Soloway84].

Ainsi, nous pensons qu'il est possible de considérer les programmes sous une forme structurelle, quel que soit le niveau d'abstraction, que nous nommerons *aspect*, considéré par rapport à sa composition ou son fonctionnement[60]. C'est cette considération structurelle des programmes qui constituera le domaine *d'origine*[61] de la construction de nos représentations analogiques.

b) le domaine cible : les représentations graphiques

Le second domaine, *cible*, de construction des représentations analogiques est l'image, la représentation graphique. Afin de pouvoir établir un lien entre les programmes et ce domaine, nous considérons les représentations graphiques comme des objets structurés. Ces structures, d'un type totalement différent de celles présentes dans les programmes, se rapprochent des structures prédicatives à la base des modèles de construction [Pineda88a] ou d'interprétation [Reiter89] d'images. Nous considérons en effet les différents objets présents dans les images dans une description relative (en placement ou en propriété) aux autres objets présents. Ainsi, l'image d'un paysage ne sera pas une image globale mais un ensemble de descriptions d'objets. Cet ensemble aura alors comme structure les indications de déplacements relatifs ou de similarité dans telle ou telle propriété existant entre les objets la composant.

Cette considération des représentations graphiques nous permet d'une part d'établir un lien de corrélation entre structures de programmes et structure présente dans une représentation graphique et d'autre part de pouvoir *spécifier* une représentation graphique par un ensemble de relations et d'attributs (les objets graphiques). Ces deux points de vue nous permettent alors d'appliquer les conclusions des travaux de Derdre Gentner et de Douglas Hofstadter mais également de proposer une vision abstraite des images comme un schème descriptif composé d'objets et de contraintes afin de composer des représentations liées à un contexte analogique particulier.

II.2.2.2 Présentation de la méthode

Notre méthode de construction de représentations analogiques de programmes permet de lier la description de *points de vue* sur les programmes avec la description de représentations graphiques structurées.

La construction d'une représentation analogique doit, selon nous, répondre à la démarche suivante :

---

[60] Ces structures, envisagées comme des graphes ou sous toute autre forme, ne peuvent malheureusement pas encore être considérées comme de dimension fractale : il existe toujours un niveau insécable, que l'on parle d'instruction, d'expression, de flot, de plan ou de signature.

[61] A comprendre comme *l'échange de sens*, dans une métaphore ou une analogie, entre un domaine *d'origine* et un domaine *cible*.

1) le choix de l'utilisation de la représentation analogique,

   2.a) le choix de l'aspect des programmes,

   2.b) le choix du domaine de la représentation,

3) la spécification des liens unissant l'aspect et la représentation en fonction du choix sémantique.

Cette démarche prend donc en considération les deux niveaux présents dans les analogies et les métaphores :

1) le niveau sémantique[62] ou l'interprétation de l'analogie. Quelle que soit la théorie considérée sur ce sujet, on s'accorde généralement sur le fait que les métaphores sont des vecteurs de sens qui permettent d'exprimer un concept, un fait, etc. appartenant à un domaine cognitif particulier par l'empreint d'une *image* dans un autre domaine[63].

2) L'étude des liens entre les objets des domaines cognitifs en jeu dans une analogie. Comme nous l'avons vu, la description de ces liens peut être basée sur une étude contextuelle[64] ou, comme c'est le cas dans notre approche, une étude que nous qualifierons de *méta-contextuelle*[65], définissant la présence d'analogies par la présence de liens structurels entre les objets et posant ces liens comme indépendants d'un contexte[66] particulier.

Détaillons maintenant les étapes de notre méthode.

II.2.2.3 Le choix sémantique

Le choix sémantique est la partie essentielle de la construction d'une représentation analogique : il permet de répondre à la question *Quelle est l'utilisation qui sera faite de la représentation ?* Ce choix a de fortes implications qui dépassent cette simple question :

1) il influence le type même de système implémentant la méthode. Par exemple, les différents types de systèmes existant dans les disciplines de la visualisation de programmes sont construits à partir d'un choix sémantique :

   - les systèmes de visualisation d'algorithmes tentent de répondre à des questions comme :

[62] Aussi nommé *connaissance* chez Gentner.

[63] Et ce quelle que soit la distance présente entre les domaines considérés.

[64] Une métaphore sera détectée par la présence de tel ou tel mot ou telle ou telle expression dans un contexte sémantique particulier.

[65] Référence direct à la notion de *squelette conceptuel*.

[66] La notion de contexte désignant alors le couple « *programme représenté – représentation* ».

- *Quel type de visualisation est la plus appropriée pour enseigner tel ou tel algorithme ?*
- *Quel type de visualisation utiliser pour explorer tel ou tel algorithme ?*
- *Quel type de visualisation utiliser pour comparer les performances de différents algorithmes ?*

- Les systèmes de visualisation de programmes s'attachent, pour leur part, à construire des représentations graphiques permettant d'illustrer les différents éléments présents dans un langage de programmation particulier ainsi que son fonctionnement.
- Quant aux environnements de programmation *analogique*[67], ils cherchent davantage à permettre de construire ou d'utiliser des représentations graphiques particulières pour les différentes tâches présentes dans le processus de construction de programmes. De plus, ils peuvent tenter de construire des représentations graphiques applicables à différents langages de programmation. Ainsi, les questions sémantiques auxquelles ces systèmes tentent de répondre seront par exemple la pertinence de telle ou telle représentation par rapport à telle ou telle tâche du processus de la construction d'un programme.

Nous qualifions notre système Zeugma d'environnement de programmation analogique. Mais, de façon spécifique, il permet d'apporter des éléments de réponse aux trois types de systèmes énoncés ci-dessus. Comme nous le verrons dans nos exemples de représentations analogiques présentés dans la section suivante, il permet de construire des analogies utilisables pour l'optimisation, l'aide à la découverte de programmes ou encore la visualisation d'algorithmes.

2) Il influence les choix dans les étapes suivantes de la méthode. Cette influence est plus que déterminante.

Le choix sémantique :

- implique une notion de complétude. Notion que l'on peut ex-

[67] Nous entendons par *environnement de programmation analogique* un environnement dans lequel les différentes tâches de la mise au point et de l'observation du comportement des programmes sont aidées par l'utilisation d'outils proposant des représentations graphiques. Ce qui les distingue principalement des systèmes de visualisation de programmes, c'est une dimension d'ouverture présente dans les environnements de programmation : ils doivent donner la possibilité à l'utilisateur d'ajouter de nouveaux outils liés à telle ou telle tâche particulière, que ces outils proposent des liens avec des représentations graphiques ou non. De plus, les environnements de programmation peuvent permettre l'utilisation de ces outils sur différents langages de programmation, ce que ne permettent, par définition, pas les systèmes de visualisation de programmes.

primer comme la réponse à la question *Est-ce que l'analogie représente bien TOUT ce qu'elle doit représenter ?*

- influence le choix du ou des points de vue sur les programmes pris en comptes,
- influence, ou restreint, les choix possibles dans les représentations graphiques possibles. En effet, la construction d'une analogie ayant pour finalité d'illustrer le fonctionnement d'un programme[68] devra nécessairement contenir des éléments animés[69].

En fait, il nous paraît déterminant de ne pas perdre de vue ce choix et ses implications dans la construction d'une représentation analogique ou même dans l'élaboration d'un système aidant à leur construction. Comme nous le décrivons dans le chapitre IV de ce mémoire, on rencontre trop souvent des exemples de représentations ayant pour vocation d'illustrer tel ou tel aspect des programmes mais qui en fait oublient une partie du message initial, du but même de la représentation.

II.2.2.4 Choix des aspects des programmes pris en compte

Le choix des aspects des programmes pris en compte dans une représentation analogique dépend entièrement du but même de cette représentation.

En effet, suivant sa finalité[70], l'aspect du programme illustré sera totalement différent : une visualisation tendant à aider à la compréhension d'un algorithme sera liée à la progression de la modification des données au cours de l'exécution du programme alors qu'une visualisation dont la finalité est la recherche d'optimisation dans la construction d'un programme s'attachera plus à prendre en compte sa composition.

De plus, comme nous l'avons précisé, le choix sémantique influence le type même de système implémentant la méthode. Ainsi, ces types de systèmes proposeront des points de vue liés à ce choix sémantique. On peut alors distinguer les systèmes d'animation d'algorithmes et de visualisation de programmes d'une part et les environnements de programmation d'autre

[68] D'un programme ou d'un algorithme.

[69] Que ces éléments soient animés au sens propre du terme (avec des objets en mouvement) ou au sens figuré (l'utilisation de couleurs, de placements relatifs ou même de formes particulières permettent d'obtenir une *illusion* d'animation sur des images fixes).

[70] Nous incluons ici l'utilisation des représentations analogiques dans l'aide à la construction, la mise au point ou la documentation de programmes, ainsi que dans l'enseignement des méthodes de programmation, tâche pour laquelle l'utilisation de représentations analogiques a montré son efficacité [Baecker97b], [Bazik97], [Mulholland97].

part :

- les systèmes de visualisation d'algorithmes proposent généralement l'accès, sous différentes formes, aux valeurs manipulées par les programmes au cours de leur exécution. Les systèmes de visualisation de programmes utilisent[71], eux, généralement des points de vue liés à l'étude de la composition des programmes. Ces deux types de systèmes fixent alors généralement l'ensemble des points de vue utilisés.
- Les environnements de programmation se basent, eux, sur un ensemble de points de vue plus large. Ils proposent en effet généralement des outils permettant d'illustrer un large panel de choix sémantiques et doivent ainsi disposer des informations sur les programmes relatifs à ces questions. Inversement, notre méthode permet de déduire les questions sémantiques auxquelles un environnement de programmation peut répondre en fonction des points de vue sur les programmes dont il dispose et à partir desquels sont construites les représentations graphiques de l'environnement.

Dans Zeugma, nous avons choisi de considérer comme ouvert le choix des points de vue sur les programmes afin de permettre la construction de représentations analogiques s'appliquant au plus large nombre de questions sémantiques possible. Cette ouverture, comme nous le présenterons dans le troisième chapitre de ce mémoire, est rendue possible en offrant à l'utilisateur la possibilité de définir ses propres points de vue sur les programmes et de les ajouter au système.

II.2.2.5 Choix de la représentation graphique

Plus encore que le choix des aspects des programmes, le choix de la représentation graphique constitue un domaine qui doit être considéré comme ouvert. Ce choix est en effet intimement lié à des considérations de culture [Kehoe96], de contexte humain[72], mais aussi à la tâche prise en compte [Gerstendörfer87] ou à la perception qu'ont les programmeurs de leurs programmes [Ford93]. Marian Petre et Blain Price, dans leur critique de la consi-

[71] Ces systèmes proposent généralement des représentations graphiques préétablies et détaillent rarement les méthodes utilisées pour lier tel ou tel aspect des programmes à leur représentation graphique.

[72] L'exemple suivant, même si, à la différence des autres références, il ne se situe pas dans l'informatique, illustre notre propos : « *Le violoncelliste danois Per-Olof Hacker raconte l'histoire émouvante d'une fille sourde de sa famille qui voulait […] se représenter la musique à travers les couleurs. […] La fille avait remarqué que certaines musiques passaient surtout dans l'enceinte des aigus, et peu ou pas du tout dans celle des graves. Elle les appelait les musiques bleues. […] Quant à celles qui mettaient tout en branle, ajoute Hacker, "Ingrid les trouvait noires, alors que nous les aurions naturellement associées à l'idée de blancheur."* » [Ninio89, page 78]

dération des systèmes présentant les représentations graphiques comme la panacée, disent par exemple [Petre92, page 221] :

> L'utilisation de notations est complexe et nécessairement hétérogène. Même lorsqu'il existe une unique tâche globale, comme la compréhension de programmes, les sous-tâches induites incluent la reconnaissance, la lecture, la compréhension et la recherche, parfois même simultanément. Ainsi, la décision concernant la technique graphique a utiliser, si elle existe, est une tâche non triviale et demande une analyse détaillée de la tâche représentée.[73]

Or, il ne nous paraît pas possible d'établir une classification exhaustive des liens entre les choix sémantiques possibles et les différentes représentations graphiques existantes et à venir. En fait, comme le disent clairement (entre autres) Marian Petre et Blain Price : c'est une tâche non triviale. Toutefois, l'implémentation de notre méthode, comme nous le verrons dans la section III.2.1.3.2 de ce mémoire, fournit les éléments permettant de tenter l'ébauche d'une telle classification.

II.2.2.6 Spécification de liens analogiques entre points de vue sur les programmes et représentations graphiques

Les précédents aspects de notre méthode (le choix sémantique et le choix des différents éléments pris en compte) abordent principalement des considérations subjectives, liées à un contexte et aux acteurs en jeu dans le processus de l'élaboration d'une représentation analogique[74]. Toutefois, il nous paraît important d'insister sur les liens existant entre eux puisque des choix judicieux permettent de réaliser des représentations analogiques intuitivement compréhensibles pour une majorité d'utilisateurs.

Un autre aspect important de notre méthode, et qui constitue la base algorithmique de notre implémentation, est le moyen mis en œuvre dans le tissage des liens entre les éléments du domaine *d'origine* (les points de vue sur les programmes) et les éléments du domaine *cible* (les représentations graphiques). Comme nous le montrons dans le chapitre IV de ce mémoire,

---

[73] « The uses of notations are complex and heterogeneous. Even when there is a single overall goal, such as program understanding, sub-tasks could include recognition/scanning, reading, comprehension, and searching, sometimes simultaneously. Thus the decision as to which graphical techniques to use, if any, is non-trivial and requires a detailed task analysis. »

[74] Les acteurs de la construction d'une représentation analogique peuvent être constitués de personnes distinctes comme le programmeur, l'animateur et l'observateur du modèle présenté par Gruia-Catalin Roman [Roman88]. Toutefois, nous ne pensons pas que cette séparation soit une absolue nécessité : un programmeur peut décider de se construire une représentation analogique de programme à ses propres fins.

les systèmes de construction d'animations d'algorithmes ou de visualisations de programmes considèrent ce point comme central dans le débat animant ce domaine de recherche. Certains considèrent en effet qu'il ne faut en aucun cas modifier les programmes[75], alors que d'autres pensent que l'important est de notifier les différentes opérations intéressantes, ce qui n'est pas forcément, selon eux, déductible du programme lui-même.

Notre approche se distingue de ce débat par la proposition d'établissement de liens non plus entre un élément du programme et une représentation mais entre un aspect des programmes et la description d'une représentation graphique. Ainsi, notre méthode est conçue pour s'appliquer à une abstraction sur les programmes, sans être liée à un programme ou à un domaine applicatif particulier. En fait, une représentation analogique construite avec notre méthode doit pouvoir s'appliquer à tout programme existant[76].

Dans notre implémentation de cette méthode, nous définissons des objets particuliers dont la fonction est la spécification de ces liens unissant les deux domaines. Ces objets, nommés *Objets de Relations Structurelles* (ORS) et présentés en détail dans le chapitre suivant de ce mémoire, se caractérisent par les propriétés suivantes :

- chaque ORS correspond à un contexte[77] issu d'un aspect et lie ce contexte au dessin d'un objet graphique,
- chaque ORS peut être lié, via des contraintes relatives à un aspect[78], à d'autres ORS. Chacun de ces liens est mis en relation avec une transformation graphique[79].
- les liens entre les ORS peuvent entraîner un changement de l'aspect des programmes. Ceci permet alors de composer des représentations analogiques combinant plusieurs points de vue.

Un ensemble d'ORS, spécifiant une représentation analogique, constitue alors un *schème de génération de représentation analogique*. En effet, cet ensemble d'objets définit des liens entre des points de vue sur les programmes et la spécification (en terme de génération d'objets et de leur positionnement relatif) d'une représentation graphique.

Voyons maintenant des exemples de représentations analogiques de

---

[75] Tout en aboutissant souvent à des systèmes dans lesquels des modifications dans les programmes observés deviennent nécessaires.

[76] Tout en prenant toutefois en considération le système qui implémente le modèle et particulièrement le langage de programmation avec lequel il travaille.

[77] Le terme *contexte* est comparable aux nœuds d'un graphe.

[78] Le terme *contrainte* est à considérer, conservant la métaphore des graphes, comme les arcs reliant les nœuds.

[79] *Transformation* désignant ici autant des notions de positionnement spatial que des propriétés comme la couleur ou la taille.

programmes construites en suivant cette méthode.

II.3 Exemples de représentations analogiques de programmes

II.3.1 Introduction

Nous allons maintenant décrire une série d'exemples de représentations analogiques de programmes construites en appliquant notre méthode et en utilisant notre système Zeugma. Ces exemples présentent trois types d'analogies :

1) L'analogie entre des programmes et des plans de villes (ou des cités). Comme nous allons le voir cette première représentation analogique permet une approche ou lecture des programmes, discriminant de manière visuelle ses différents composants, leur complexité et structure, ainsi que leur importance et leur rôle dans le déroulement de l'exécution.
2) L'analogie entre des programmes et des araignées en mouvement sur une toile. Cette représentation analogique s'attache à représenter graphiquement les échanges entre les différentes parties des programmes au cours de leur exécution. Cette analogie est plus directement liée à une vision de la dynamique en jeu dans l'interaction des parties du programme que la précédente analogie.
3) Une représentation analogique animée construite afin d'illustrer les différentes étapes en jeu dans des algorithmes de tri. Cette représentation utilise les histogrammes et combine une animation du déroulement de la résolution des algorithmes avec la persistance de son historique. Nous montrons dans la description de cette visualisation qu'elle permet de déduire visuellement les différentes étapes en jeu dans trois algorithmes de tri : le tri rapide, le tri par fusion et le tri par insertion.

Dans la description de ces représentations analogiques de programmes, nous préciserons en premier lieu la source de l'analogie, quelles sont les motivations de sa création, les liens qu'elles établissent entre des programmes et des objets comme des maisons ou des araignées. Puis nous allons parcourir les différents types d'informations qu'elles présentent en décrivant une série d'images représentatives de ces représentations reliées aux différents niveaux de lecture de l'analogie. Ces différents niveaux de lecture partent d'une vision globale de la représentation graphique pour aboutir aux détails des éléments les composants et à leur rôle actif ou passif dans la visualisation du déroulement de l'exécution des programmes représentés.

II.3.2 Les programmes comme des plans de cités

« Dans son ensemble, le développement de la ville, dé-

pendant d'un commandement unique (le conseil édilitaire), donnera une sensation d'unité, de cohérence, - chose rassurante.

Dans son détail, ce développement comportant l'éclosion de cellules individuelles (les maisons), qui sont chacune un individu, tend à l'incohérence. » [Le Corbusier25, p. 64]

II.3.2.1 Motivation de l'analogie

L'urbanisme et l'architecture abordent, pour des créateurs comme Le Corbusier, ou même chez ses détracteurs comme le Team X, l'espace et son organisation de manière fonctionnelle. Le souci de cohérence présent chez Le Corbusier se manifeste par exemple dans sa contribution à la Charte d'Athènes (1943) dans laquelle il définit avec les membres des Congrès Internationaux d'Architecture Moderne les quatre fonctions élémentaires qui régissent l'organisation de la ville : habiter, travailler, se recréer, circuler. Même si cette catégorisation n'a plus aujourd'hui la même influence qu'à l'époque, elle pose la ville ou la cité comme un ensemble organisé autour de fonctions élémentaires. Cette vue est similaire à notre perception de programmes comme une combinaison, une mise en relation, d'un ensemble de fonctions élémentaires.

« *Dans l'une de nos vues (à droite), la cloison est fermée entre les chambres d'enfants, mais ouverte vers l'escalier. Le mur vers la chambre de la mère est également replié. De ce fait on voit, au-delà des lits des filles installés l'un à coté de l'autre dans un coin, la rampe d'escalier, l'échelle qui est montée dessus et mène à la trappe,*

ainsi que l'armoire et la fenêtre de la chambre. Dans l'autre vue, la cloison donnant sur la salle de séjour/salle à manger est fermée, de sorte qu'une partie peut bouger, comme une porte. » [DeStijl91, p. 141]

Figure 3.1 Vues de l'intérieur de la maison Shröder

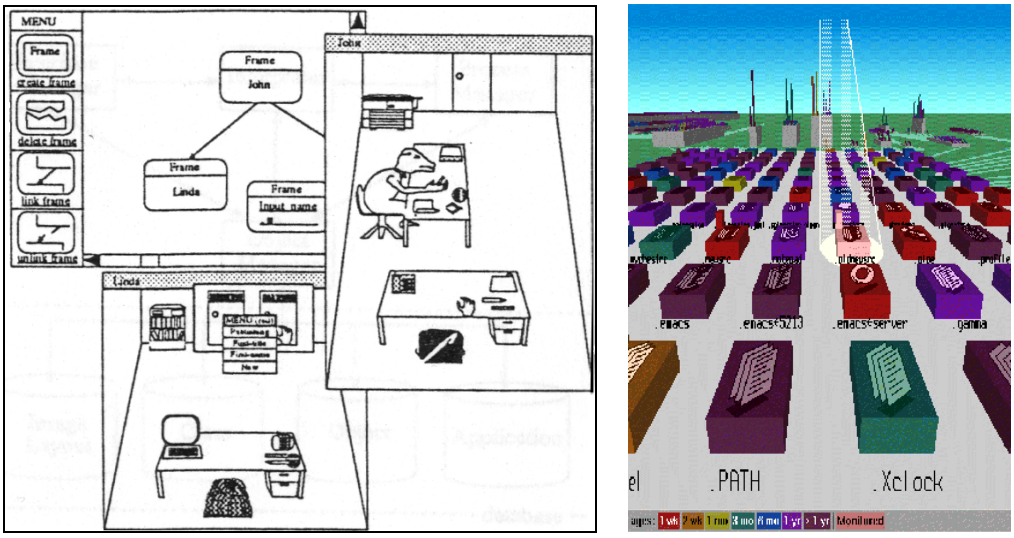


Figure 3.2 HI-VISUAL [Kado92] et FSN™

D'autres exemples liés à l'art et à l'architecture, comme le mouvement De Stijl, considèrent la couleur comme matériau[80] et posent une réflexion sur l'utilisation et la modularité de l'espace qui serait commune à la peinture, la sculpture ou l'architecture, ce à quoi nous ajoutons les *programmes*[81]. Ce groupe était composé entres autres du peintre Piet Mondrian et de l'architecte Gerrit Rietveld, dont la maison Schröder[82] réalisée en 1924 (figure 3.1) est considérée comme *La réalisation* du groupe.

Dans la construction d'environnements ou d'interfaces utilisateur, la métaphore des cités[83] ou même de l'organisation de pièces[84] se développe

[80] La couleur comme matériau, ou relier l'effet d'une couleur sur la perception de l'espace : une pièce sombre paraîtra plus petite qu'une pièce claire de même dimension, les couleurs dans la gamme des rouges paraissent plus *chaudes* que celles des bleus, …

[81] Notons les travaux de Arnowitz et al. [Arnowitz97] qui s'inspirent des concepts élaborés par le groupe De Stijl dans la conception d'interface utilisateur.

[82] Cette réalisation offre comme caractéristique principale, outre l'utilisation de la couleur comme matériau, la modularité de chaque espace intérieur, permettant de modifier à tout moment l'utilisation de telle ou telle partie d'une pièce tant dans sa dimension que dans la fonction qu'elle offre. De plus, cette maison conserve une constante graphique (couleur, forme) entre l'aspect extérieur et les espaces intérieurs.

[83] Avec, par exemple, l'utilitaire FSN de parcours de données structurées développé par Silicon Graphics Inc., rendu célèbre par son apparition dans le film *Jurassic Parc*. La figure 3.2 présente un exemple de vue construite avec cet utilitaire.

depuis quelques années. Peut-être Vernor Vinge fut-il inspiré par ces travaux dans l'écriture de sa nouvelle de science fiction True Names [Vinge87] car il y présente l'évolution de personnages dans un monde virtuel composé de paysages et de villes. La particularité de cette nouvelle est que l'auteur y établi un lien entre les éléments graphiques virtuels et la machinerie informatique sous-jacente. Pour lui, une porte cachée *sera* la correspondance graphique d'un mot de passe, une araignée un programme, …

Nous établissons les liens suivants avec l'étude de la représentation analogique de programmes :

- Les premiers travaux sur la visualisation de programmes [Nassi73] basée sur des diagrammes utilisent une discrimination visuelle des fonctionnalités par des structures spatiales en deux dimensions. La correspondance entre fonctionnalité et organisation spatiale est à la base de l'organisation des différents niveaux de notre représentation : plans de cités, quartiers, bâtiments, composition interne des bâtiments, … Ainsi, notre analogie permet, comme nous allons le montrer par la suite, de discriminer visuellement les différents types de fonctionnalités présentes dans les parties composant un programme.
- Pour comprendre le fonctionnement d'un programme il est essentiel de le considérer dans sa globalité *cohérente* face à ces parties qui, prises séparément, semblent *incohérentes*[85]. Ainsi, les programmes peuvent être considérés comme des groupements d'espaces fonctionnels dans lesquels chaque partie correspond à un espace particulier[86]. Cette vue nous a amené, dans la construction de la représentation graphique, à imposer une cohérence visuelle entre objets programmatoires d'un même type et entre les moyens de discrimination des particularités de ces objets. Cette cohérence permet à notre analogie de distinguer visuellement les groupes fonctionnels présents d'un simple coup d'œil.
- Outre une approche basée sur la visualisation d'éléments figés (des maisons, des routes, des quartiers) vecteurs en eux-mêmes d'informations, nous avons animé notre analogie par le déplacement d'un personnage dans la ville. Cette animation permet de suivre visuellement la progression dans l'exécution du programme.

Nous allons maintenant détailler cette analogie par une série d'exemples, sa finalité étant de tenter d'illustrer cette cohérence globale d'un programme[87].

---

[84] avec en particulier les travaux autour de l'interface utilisateur iconique HI-VISUAL [[Hirakawa86, 90] [Ichikawa87] [Kado92] [Miller95] dont une vue est présentée dans la figure 3.2

[85] Selon les termes de Le Corbusier.

[86] Cette notion d'espace fonctionnel se retrouve également dans les travaux de [XXX], qui construit des espaces de solutions pour des programmes Prolog.

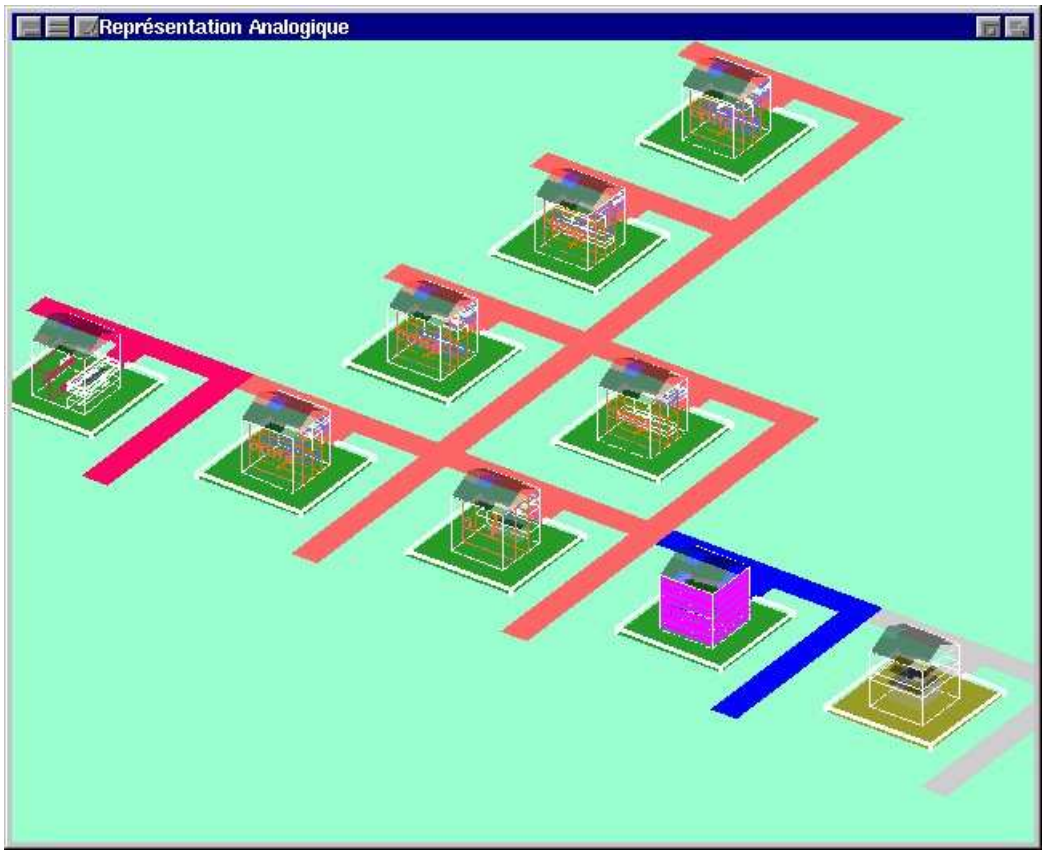
[87] Cohérence bien évidemment dépendante du résultat donné par le programme par

II.3.2.2 Vision globale d'une cité

Les deux images de la figure 3.3 présentent deux visions globales de deux cités, nommées cité *Graphe* et cité *Hanoi*, représentant chacune un programme Lisp différent. Cette première vision globale amène une série de remarques :

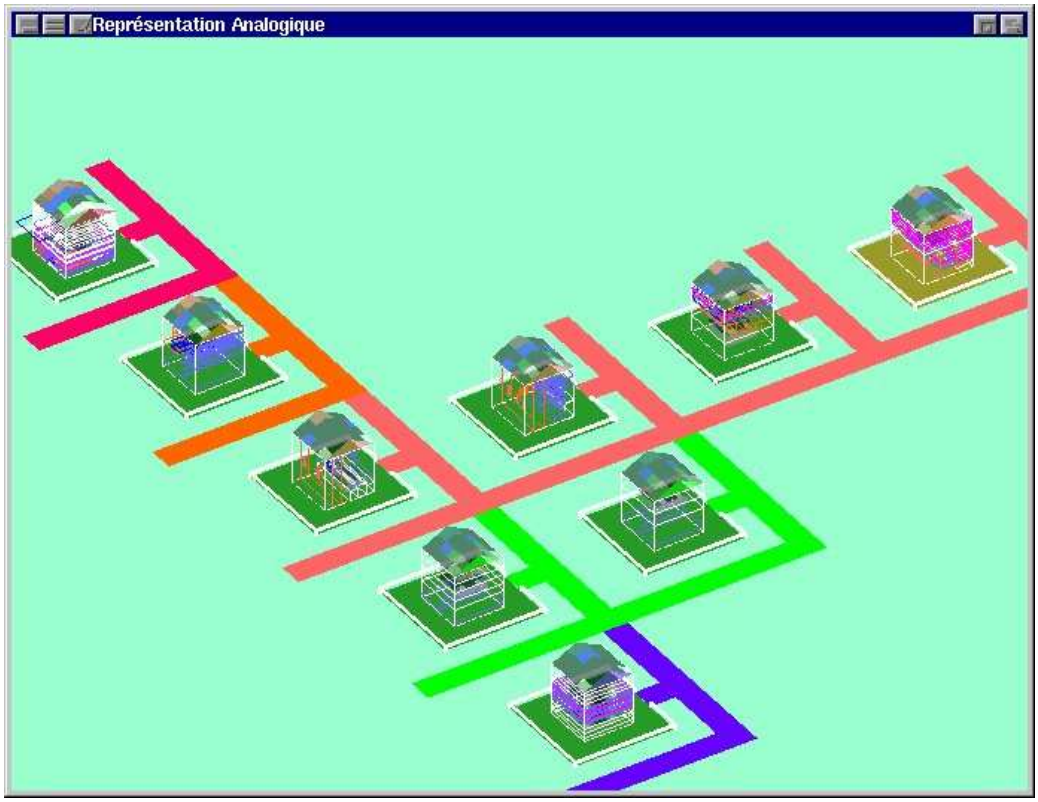
a) On peut distinguer des régularités graphiques dans l'organisation globale des cités, régularités dans la structure urbaine :

- Les rues sont toutes rectilignes, ce qui amène à déduire que toutes les parties de la cité, quartiers ou maisons, communiquent entre elles d'une manière uniforme.
- Les différents quartiers (ou présupposés comme tels suivant la couleur des rues) sont distincts les uns des autres. Ainsi ces quartiers regroupent des entités (des maisons) ayant des points communs, partageant des particularismes.



(a) Cité *Graphe*

rapport au résultat escompté. En d'autres termes, le programme comporte t'il des erreurs dans sa syntaxe ou sa sémantique ?



(b) Cité *Hanoi*

Figure 3.3 Visions globales de deux citées

- Toutes les maisons sont pareilles ! Ou du moins pour leur aspect général extérieur comme leur taille, la possession d'un jardin et d'une entrée unique.

b) Des particularismes, dans les quartiers ou les maisons apparaissent eux aussi :

- Les quartiers ne semblent pas uniformes dans leur composition, regroupant ainsi plus ou moins de maisons.
- Certaines maisons apparaissent différentes par leur opacité ou, à l'inverse, par le peu de choses qu'elles contiennent.

Ces remarques, issues uniquement d'une observation des représentations graphiques, peuvent être liées à des caractéristiques des programmes observés. Pour cela, il est nécessaire de clarifier le sens analogique des composants graphiques :

- Les liaisons entre maisons ou quartiers symbolisent des voies de communication entre les parties du programme. Ces voiries seront utilisées dans la représentation de l'exécution du programme par un personnage en mouvement. L'organisation choisie (regroupement par quartier) aboutit à un placement relatif des maisons n'étant pas lié au flot de contrôle ou de données. Cependant, cette organisation n'est pas figée et une méthode d'organisation diffé-

rente, dans l'implémentation de l'analogie, peut être aisément spécifiée[88].

- Le regroupement par quartiers est issu d'un regroupement fonctionnel des parties du programme. Nous détaillerons la signification analogique de ce regroupement dans la section suivante
- Chacune des maisons représente une fonction du programme étudié. Leur aspect général répond à la conservation de la forme, dans la représentation interne comme dans la syntaxe, entre différents objets manipulés par le langage. Nous présentons plus loin le détail des correspondances analogiques entre les fonctions Lisp et les maisons.

Après l'énoncé de la signification de ces liens entre représentation graphique et programme il nous est possible, à partir des images, de déduire les points suivants sur les programmes observés :

- La cité *Graphe* représente un programme constitué de quatre quartiers différents. Toutefois, un quartier semble prédominant (de couleur rose) même si la maison du quartier bleu attire le regard par sa complexité, présentant un attrait visuel guidant le parcours de ce programme.
- De même, la cité *Hanoi* est organisée suivant différents quartiers. Le nombre de quartiers apparaît ici plus important (cinq au lieu de quatre) même si le quartier prédominant semble être le même que pour la cité *Graphe*.

La vision globale de la représentation d'un programme Lisp par une cité constitue alors un premier guide dans l'exploration de celui-ci. Elle attire l'attention sur des points particuliers (des éléments visiblement différents des autres) ou sur des impressions plus globales (l'étendue de tel ou tel quartier).

Continuons l'exploration de cette représentation analogique par une description en zoom avant, la prochaine étape étant les quartiers.

II.3.2.3 Les différents quartiers

La figure 3.4 présente deux vues rapprochées de quartiers des cités présentées dans la figure 3.3. Par rapport à la vision globale, permettant déjà de distinguer la caractéristique graphique principale des quartiers (une différentiation par l'utilisation de couleurs particulières pour les chemins), ces nouvelles vues apportent des informations supplémentaires :

[88] Par la spécification et l'utilisation d'un *point de vue* différent sur l'organisation des fonctions du programme. Nous considérons les flots de contrôle et de données ou encore, comme dans cette représentation analogique, un ordonnancement issu d'une classification comme autant de points de vue particuliers.

- les couleurs caractérisant les quartiers ne sont pas uniquement présentes sur les voies de communication mais aussi sur le fronton des maisons,
- des panneaux indicateurs sont présents devant chaque quartier,
- ce zoom permet de mieux distinguer le contenu de certaines maisons et elles semblent contenir d'autres maisons.

Ces observations visuelles amènent à décrire de manière plus précise les liens analogiques permettant la construction des quartiers : les fonctions d'un quartier partagent des caractéristiques communes. Dans cette représentation analogique nous proposons ainsi une classification des fonctions. Cette classification est établie à partir de l'étude de la composition des fonctions et plus particulièrement sur la détection de primitives particulières dans leur code source.

Nous avons ainsi regroupé les primitives du langage Xbvl dans les sept classes suivantes :

1) les boucles ou répétitions (n'utilisant pas les lambdas ou les possibilités de code auto modifiant). Les éléments de cette classe de fonction seront dans les quartiers de couleur .
2) Les expressions lambda, récursives ou non, définissant ainsi des sortes de fonctions temporaires internes aux fonctions (quartiers de couleur).
3) Les branchements, que ce soit sous la forme de tests (*if*, *cond*, …) ou d'opérateurs booléens (quartiers).
4) Les entrées / sorties par des impressions ou lectures standard mais aussi par l'utilisation de primitives liées aux librairies graphiques (X Windows ou Open GL) de Xbvl (quartiers).
5) Les primitives spécifiques aux nombres (addition, multiplication, …) (quartiers).
6) Les primitives spécifiques aux chaînes (concaténation, …) (quartiers).
7) Les primitives spécifiques aux listes (*car*, *cdr*, …) (quartiers).

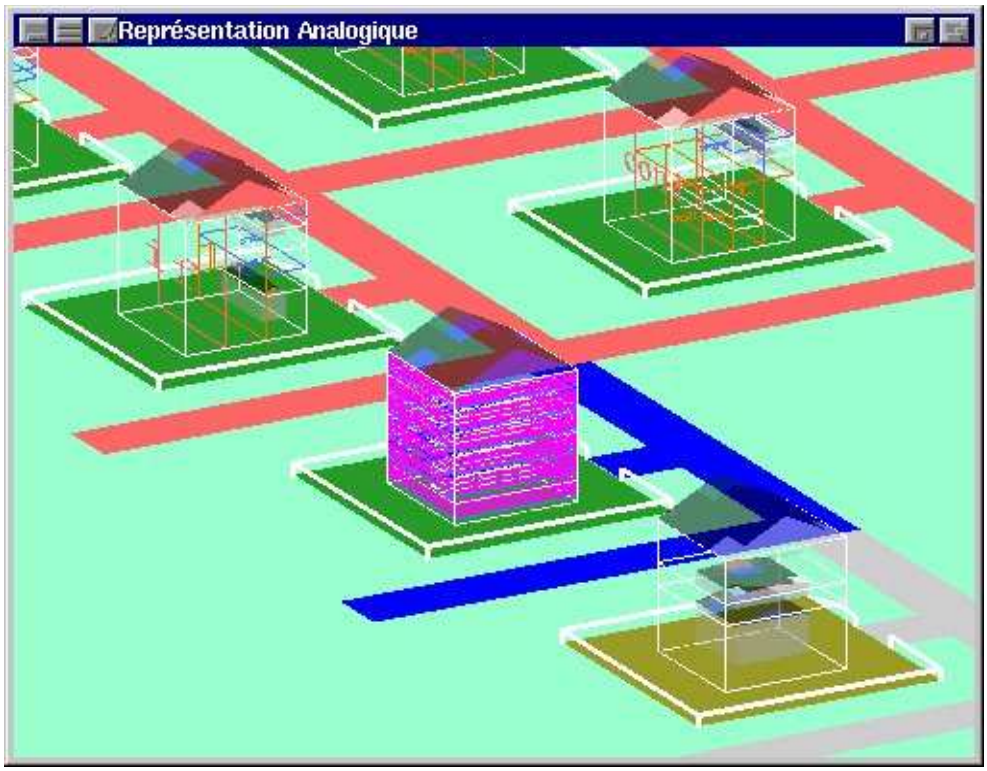
Enfin, il est possible que des fonctions n'utilisent pas de primitives du langage, ne faisant alors qu'utiliser d'autres fonctions du programme. Nous regroupons ces dernières dans la classe des fonctions utilisant « d'autres » primitives (leur quartier sera alors de couleur).

Pour pouvoir départager des fonctions utilisant plusieurs types de primitives nous avons établi cette classification de manière hiérarchique. Ainsi, par exemple, si une fonction utilise une primitive de répétition et une primitive d'entrée sortie, sa classe sera la classe des répétitions. Cette classification hiérarchique arbitraire se base sur la volonté de distinguer particulièrement :

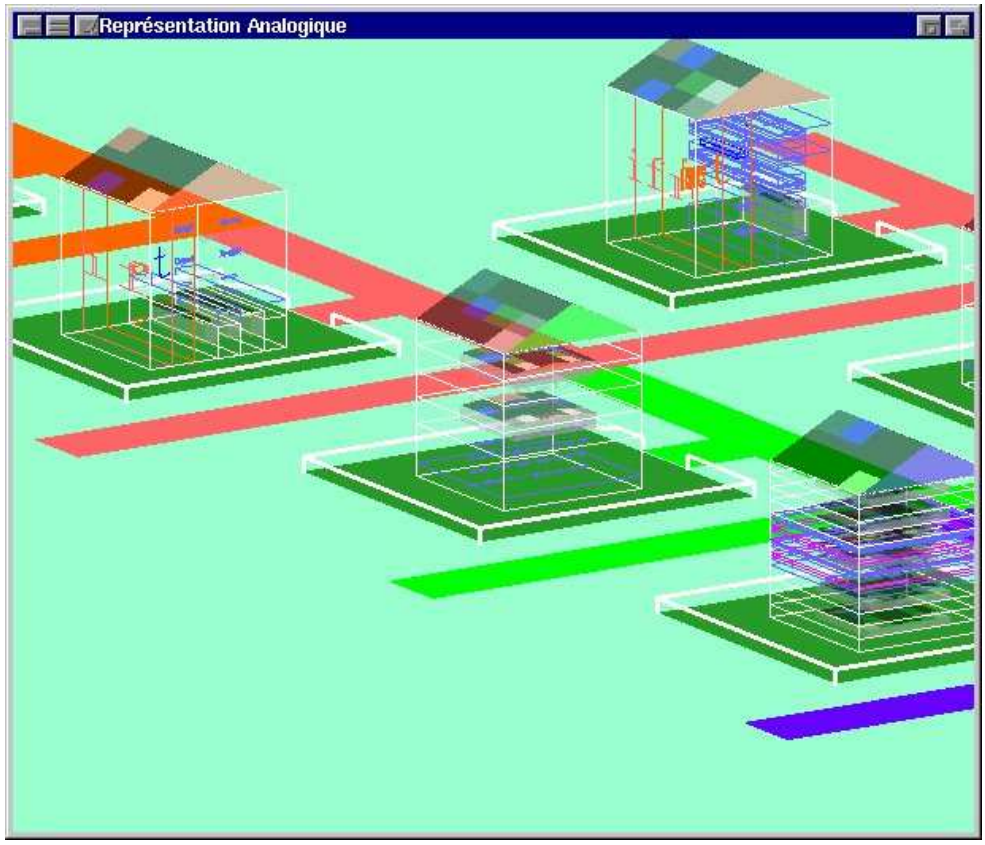
1) en premier lieu les fonctions agissant de manière significative sur le flot de contrôle par des répétitions, la définition de lambdas ou des branchements,
2) en second lieu le type de données manipulées.

Il est important de préciser ici que cette définition de classification des fonctions n'est pas la seule possible, des travaux comme ceux de F. Balmas sur la construction de *signatures* de groupes opérationnels [Balmas95a, 95b et 97] pourraient constituer la base de l'élaboration d'un autre type de classification, un autre point de vue utilisable dans l'élaboration d'une représentation analogique.

Cette classification est utilisée dans la représentation analogique de programmes par des cités afin de déterminer le quartier et, par-là même, la couleur utilisée pour les voies de communication entourant les maisons représentant les fonctions du programme. En supplément des couleurs caractéristiques, la reconnaissance des quartiers est aidée par les panneaux indicateurs contenant le nom de la classe de fonction qu'ils désignent.



(a) Zoom avant sur la cité *Graphe*



(b) Zoom avant sur la cité *Hanoi*

Figure 3.4 Zoom sur des quartiers

Le dernier point visuel remarqué en introduction de ce paragraphe est la présence de maisons à l'intérieur d'autres maisons. Nous étudierons en détail le contenu et l'organisation de l'intérieur des maisons dans le prochain paragraphe mais nous pouvons remarquer ici que la présence de la couleur du quartier sur le fronton de la maison permet de distinguer par un simple coup d'œil si une maison incluse appartient au même quartier ou non que la maison dans laquelle elle apparaît. Sachant que l'inclusion d'une maison indique un appel à une fonction du programme, il est ainsi possible de distinguer les liens entre classes de fonctions à partir de leur représentation analogique.

Par rapport aux observations effectuées sur les vues globales des deux cités nous pouvons maintenant déduire que :

- La cité *Graphe* est principalement constituée de fonctions utilisant des tests. Ce programme, s'il effectue un travail demandant un parcours, l'effectue ainsi principalement par l'utilisation d'appels de fonctions et non en utilisant des boucles ou des lambdas.

  La maison présente dans le quartier bleu travaille ainsi uniquement sur des atomes. Comme on peut le distinguer grâce à la couleur de son fronton, elle est également présente en première position de la première maison (dont le jardin est marron) représentation du point d'entrée du programme. Ainsi, cette maison semble constituer une première étape dans la résolution du problème traité par le programme[89].

- Le principal quartier de la cité *Hanoi*, comme pour la cité *Graphe*, regroupe des fonctions de test. De cette indication nous pouvons déduire que, comme le programme Graphe, le programme Hanoi effectue une partie de son travail en utilisant des récursions.

  La présence d'un quartier vert indique que ce programme effectue des entrées / sorties et qu'il définit pour cela des fonctions spécialisées.

Continuons notre visite de cette représentation analogique par la description des analogies graphiques utilisées pour construire l'organisation interne des maisons.

[89] Ceci se trouve confirmé par la consultation du programme source montrant que cette fonction place les données manipulées par le programme dans la p-liste d'une série d'atomes.

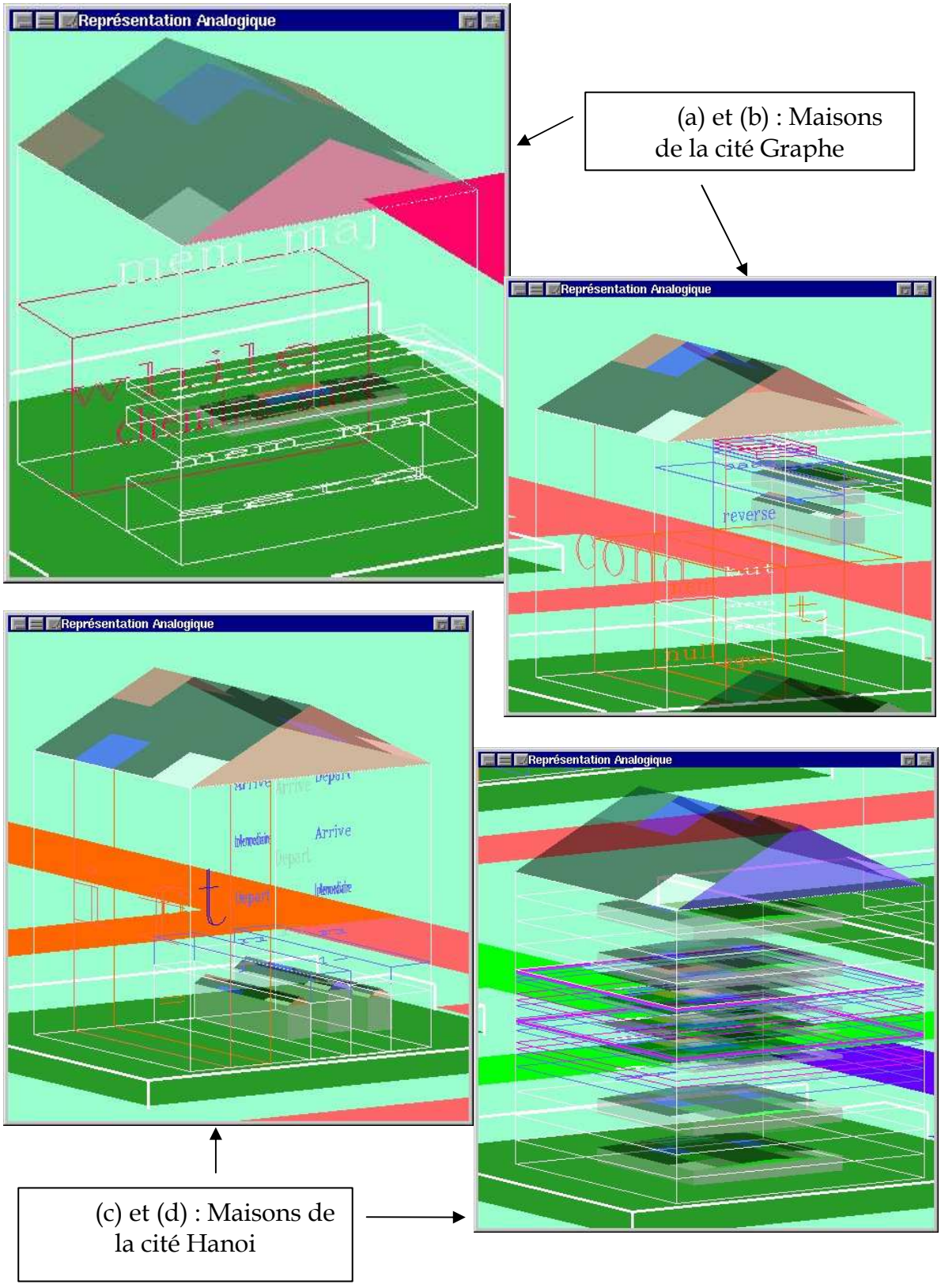


Figure 3.5 Exemples de maisons

Colonnes / lignes	1 Branchements	2 Données	3 Entrées/ Sorties	4 Variables
1	Boucles (while, repeat)	Numériques (+, -, *, …)	Impression (print, …)	Locales (paramètres et lambdas)
2	Conditions (cond, if, ifn, …)	Chaînes (strcat, …)	Librairie X (xRealize, …)	Globales (et utilisées dans la fonction)
3	Tests (or, and, eq, …)	Listes (cons, car, …)	Librairie GL (Glcolor, …)	Modifiées

Table 3.1 Correspondance Instruction / Couleur - Position pour les tuiles des maisons

II.3.2.4 Visite guidée de maisons

Entrons maintenant dans la description des liens analogiques entre les maisons, les pièces les composant, leurs forme, couleur et contenu avec les éléments appartenant au monde des programmes écrits en langage Lisp.

En premier lieu, nous pouvons décrire les éléments graphiques présents, puis nous établirons les liens avec les différents éléments représentés. Ainsi, les maisons sont composées de :

- un toit, couvert de tuiles colorées, dont l'allure semble être particulière à chaque maison,
- un fronton coloré, lui, de la même manière que les routes environnant la maison.

Les pièces sont organisées soit horizontalement, soit verticalement ; elles sont de tailles différentes et peuvent contenir soit des inscriptions, soit d'autres maisons. Voyons maintenant les liens avec les éléments des programmes, permettant ainsi une lecture analogique de la composition des maisons.

a) Signification analogique des tuiles

En fait, les tuiles présentes sur le toit des maisons constituent un moyen de discrimination entre les maisons. Leurs caractéristiques (couleurs) sont établies selon un principe identique à celui utilisé pour le choix des couleurs des quartiers ; les quartiers présentant une sorte de catégorisation des fonctions par la détection de la présence d'instructions et les tuiles un affinage de ces catégories. Cet affinage se base sur la représentation du résultat des analyses utilisant une représentation spatiale (l'emplacement des tuiles) et de coloration commune à toutes les maisons. La table 3.1 présente les liens établis entre les tuiles (leur position et la couleur choisie) et le type d'instruction énuméré dans la fonction représentée. Ce tableau se lit de la

manière suivante :

- Les numéros indiquent la position (ligne – colonne) sur le toit.
- Les couleurs sont proportionnelles (dans leur intensité) au nombre d'instructions du type étudié présentes dans la fonction : si elle n'en possède aucune la tuile apparaîtra grise et l'intensité augmentera avec le nombre d'instructions présentes (la dernière colonne, représentant le nombre et le type de variables utilisées présentes dans la fonction, utilise une variation de niveaux de gris entre le noir et le blanc).

Par rapport à la classification présentée au paragraphe précédent, les tuiles apportent un affinage par la prise en compte du poids (le nombre) de chaque type d'instruction, la présence dans un espace réduit des différentes indications d'intensités permet de relativiser le poids de tel ou tel type d'instruction par rapport à tel autre. De plus, cette nouvelle classification permet de distinguer le type d'entrée / sortie effectué.

Par exemple :

- la maison (a) de la figure 3.5, outre l'instruction de répétition ayant conduit à son positionnement dans le quartier rouge, contient une instruction relative à la manipulation des listes (tuile bleue). La maison (b) contient, elle aussi, des instructions relatives aux listes mais visiblement plus que la maison (a).
- la maison (c) contient, dans trois de ses pièces, des maisons. Par la couleur des tuiles de ces dernières on peut voir que deux (la première et la dernière) ont les mêmes couleurs de tuiles qu'elle et que celle du milieu a le toit de la maison (d). De ce fait, et comme la maison (c) est la seule à posséder un toit de cette couleur, on peut déduire qu'elle effectue un appel récursif, un appel à une autre fonction du programme, puis un autre appel récursif.
- La maison (d) fait appel à de nombreuses autres fonctions. On peut toutefois remarquer que certaines de celles-ci ont, dans leur représentation analogique, un fronton vert. Ainsi, cette maison, appelée depuis (c), appelle des maisons aboutissant à des opérations d'entrée / sortie.

b) Organisation de l'intérieur des maisons

L'intérieur des maisons est composé de pièces simples (dans lesquelles n'est présent qu'un élément « atomique ») ou composées de plusieurs autres pièces. Les points caractéristiques des pièces sont leur taille, leur position et leur orientation.

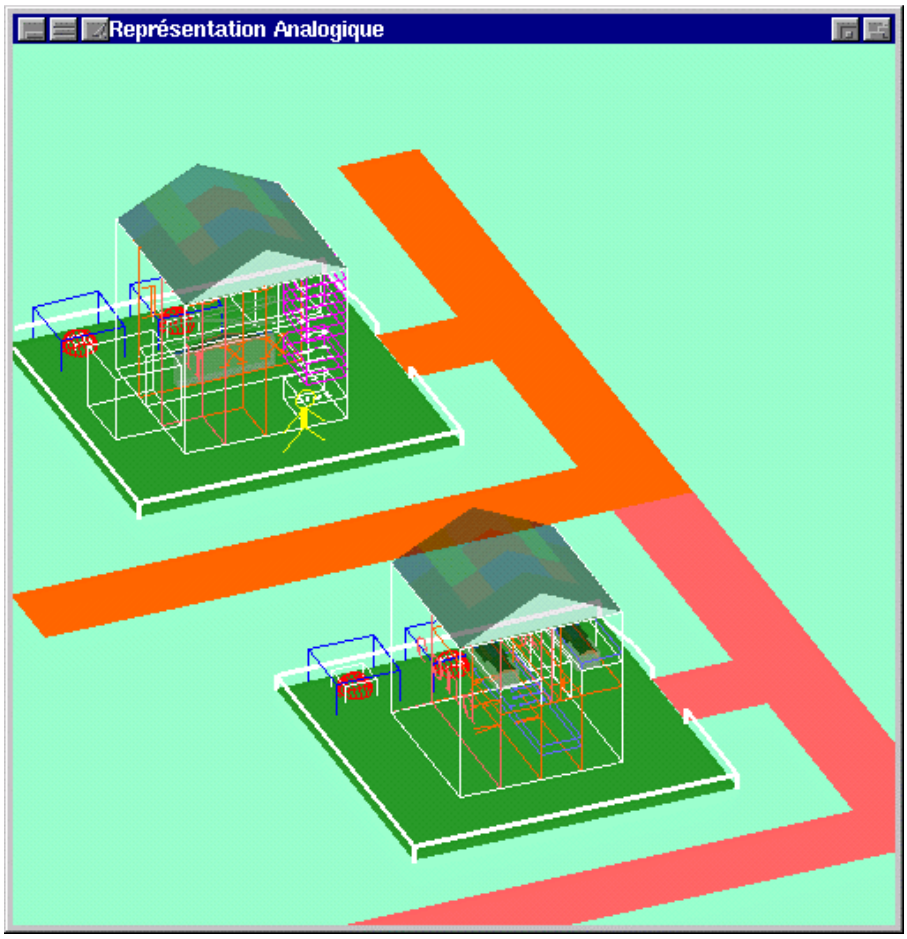
- La taille d'une pièce est fonction de la *profondeur*[90] de l'instruction Lisp qu'elle représente dans le code source de la fonction.

[90] La profondeur, comme nous le détaillerons par la suite, doit être comprise ici comme le niveau d'imbrication de l'expression.

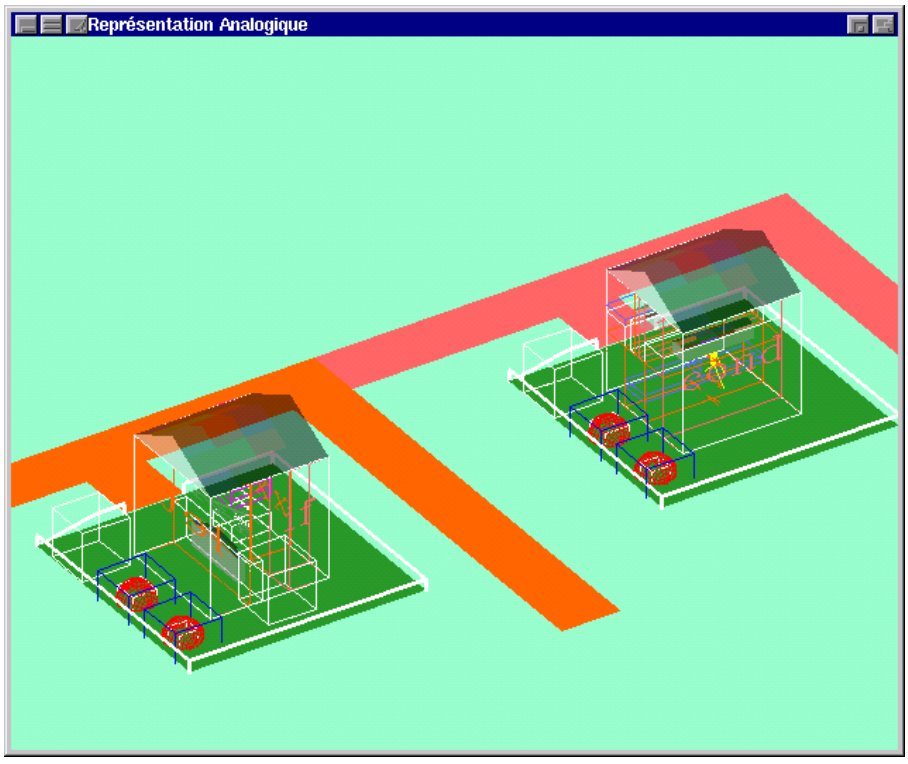
- L'organisation spatiale des pièces dépend du type d'instruction représentée :
    1. Les composants d'une expression de type *conditions* seront organisés horizontalement en largeur (comme, dans les maisons (b) et (c)).
    2. Les composants d'une expression de type *lambdas* ou *boucles* seront organisés horizontalement en profondeur (comme la maison (a)).
    3. Les composants d'une expression d'un autre type seront organisés en hauteur (comme la maison (d)).

  Ce point implique que la position d'une pièce est toujours relative à la position d'une autre pièce, dans son orientation (suivant le type) ou dans sa taille (chaque partie d'une expression occupe le même espace quelle que soit sa complexité).
- La couleur d'une pièce dépend du type d'instruction contenue, suivant la même répartition couleur / type d'instruction que celle utilisée pour la construction des tuiles du toit de la maison.
- L'objet graphique contenu peut être soit une maison (dans le cas d'un appel à une fonction du programme, la maison représentant cette dernière sera affichée (l'intérieur des maisons incluses est alors remplacé par une boîte grise transparente car l'affichage du contenu graphique rendrait impossible la représentation de fonctions récursives). Si l'instruction représentée dans la pièce n'est pas une fonction du programme son nom sera alors affiché.

Cette organisation interne des maisons permet, de par la couleur, la taille et l'orientation des pièces, de distinguer d'un coup d'œil la complexité de la fonction (en terme de taille), une vue détaillée sur le type d'opérations effectuées. De plus, l'utilisation de maisons incluses, pour représenter les appels à des fonctions du programme, permet de distinguer visuellement les liens entre les fonctions du programme (et entre quartiers puisque la couleur les caractérisant est un élément de leur représentation graphique).



(a)



(b)

Figure 3.6 Déplacement d'un personnage

II.3.2.5 Un personnage se déplace et la cité s'anime

La figure 3.6 présente deux vues, prises sous des angles différents, d'une cité construite à partir du programme effectuant le calcul du plus grand commun diviseur (pgcd), au moment de l'exécution du programme. Dans ces images, les éléments présents dans les jardins ont changé de forme et un personnage se déplace.

L'animation de la représentation est en effet composée de deux éléments graphiques permettant de visualiser deux types d'éléments appartenant au programme évoluant au cours de l'exécution :

- les boîtes présentes dans les jardins, placées dès la construction des maisons, représentent les données manipulées par les fonctions. Au moment de l'exécution du programme ces boîtes changent de forme selon le type de donnée manipulé par le programme :
  - si la donnée est une valeur numérique, une sphère est affichée. La taille de cet objet dépend alors du rapport entre la valeur actuellement manipulée et les valeurs maximale et minimale précédemment contenues.
  - Si la donnée est une chaîne ou un atome, son nom sera affiché dans la boîte,
  - Si la donnée est une liste, la représentation des valeurs qu'elle contient sera affichée sous la forme de *boîtes de cons*.

  La présence des boîtes dans les jardins exprime un lien entre la variable qu'elle désigne et la fonction dans laquelle elle est utilisée. Ainsi, si deux fonctions définissent et utilisent une variable de même nom, ces deux variables sont traitées indépendamment par deux boîtes distinctes pouvant contenir des objets graphiques différents.
- le personnage en déplacement dans la représentation indique la progression dans l'exécution du programme. Il va d'objet graphique (maison, pièce, boîte) en objet graphique, guidé par le passage du contrôle à une nouvelle fonction, la progression dans l'exécution d'une fonction, ou la modification de la valeur d'une variable.

La différence entre les images (a) et (b) de la figure 3.6 n'est pas uniquement dans l'orientation ou la taille de la prise de vue. Elles représentent deux étapes différentes de l'exécution du programme. Dans la figure (a) le personnage sort de la maison du haut et se dirige vers l'entrée de la maison afin de se rendre dans la maison du bas. Si le personnage passe par le jardin c'est pour indiquer plus précisément de quel endroit de la maison il provient (de l'arrière). Dans l'image (b), le personnage est en train de se déplacer à l'intérieur de la maison, passant d'une pièce à l'autre indiquant ainsi quelle instruction est en cours d'évaluation et l'enchaînement entre les instructions

de la fonction.

II.3.2.6 Conclusion

Dans la présentation de cette représentation analogique nous avons principalement décrit les étapes 2 et 3 de notre méthode de construction de représentations analogiques. Cette présentation peut sembler contradictoire avec les remarques que nous avons faites lors de la présentation même de la méthode à savoir que le point nous apparaissant comme la source de la construction des autres étapes est ce que nous n'avons pas décrit dans cette représentation : le choix de la sémantique de la représentation analogique. A quoi sert-elle ? De fait, ce choix est déterminé. Il provient de ce que nous estimons également comme fondamentale dans une représentation analogique de programme : c'est par une étude approfondie des différents aspects d'une représentation analogique, et pas forcément dans l'annonce de ce qu'elle est censée représenter, que l'on peut réellement dire ce qu'elle est capable de visualiser, son message ou sa signification.

Dans la présentation de la représentation analogique de programmes *comme* des villes, différents types d'informations ont pu être déduits directement des images :

- des vues globales sur les cités nous avons pu déduire que les programmes présentés étaient construits de manière récursive, sans pour autant se baser sur une visualisation des enchaînements entre les fonctions[91].
- De la syntaxe graphique utilisée pour construire les maisons nous avons pu déduire des liens de type fonctionnel (telle fonction fait appel à tel type de fonction) et structurels (la complexité apparente et une vision immédiate des différents types d'organisation interne).
- Avec l'animation du personnage dans la ville nous avons pu suivre le déroulement étape par étape de l'exécution dans la visualisation analogique même. De cette animation il est également aisé de déduire les parties du programme retenant ou attirant le plus le personnage, nécessitant le plus de temps de calcul.

Ainsi cette visualisation semble être un outil adapté pour aider à la découverte d'un programme, guidant efficacement le lecteur et lui indiquant de plus les points sensibles du programme et nécessitant une optimisation.

[91] La seule visualisation du flot de contrôle d'un programme, présente par ailleurs dans Zeugma (cf. figure 2.3 du troisième chapitre de ce mémoire, page 116, pour le flot de contrôle du programme *Graphe*), ne permet pas, à elle seule, de déduire si le programme est itératif pur (par l'utilisation d'instructions de répétitions), itératif par l'utilisation de lambdas ou récursif.

II.3.3 Les programmes *comme* des araignées en mouvement sur une toile

II.3.3.1 Motivation de l'analogie

La modélisation de colonies d'insectes, sous une forme matérielle avec des robots (Ben) ou virtuelle avec la vie artificielle [Dumeur94], s'impose de plus en plus, car :

1) chaque insecte individuel est d'une structure relativement simple et aussi aisée à modéliser que les interactions entre individus,
2) le comportement d'un grand nombre de ces animaux peut vite devenir extrêmement complexe et généralement, dans des modélisations comme celles de la vie artificielle, il émerge d'une combinaison de multiples comportements individuels.

Afin d'illustrer le comportement des programmes, et plus particulièrement les échanges entre les fonctions qui les composent, nous avons choisi d'utiliser la métaphore d'une colonie d'araignées sur une toile. En effet, les araignées (cf. Figure 3.7) les plus communes possèdent généralement les caractéristiques suivantes :

- Elles génèrent leur habitat et terrain de chasse (la toile) à partir d'un matériel qu'elles sécrètent (cf. Figure 3.8).
- Les fils constituent, outre le matériau de base de la toile, des moyens de communication, d'information, permettant à une araignée de savoir si une proie est tombée dans son piège ou d'être avertie de tout événement pouvant intervenir sur sa toile.
- La notion de colonie d'araignées, même si elle existe dans la nature, est un cas relativement rare, l'araignée étant en général un insecte vivant de manière solitaire. Toutefois, les *araignées sociales* [REF] ont un comportement de colonie, regroupant jusqu'à 5000 individus sur une même toile, coopérant dans la chasse ou l'éducation et allant même jusqu'à aider un congénère qui se serait fait piéger dans la toile d'une araignée d'une autre espèce.

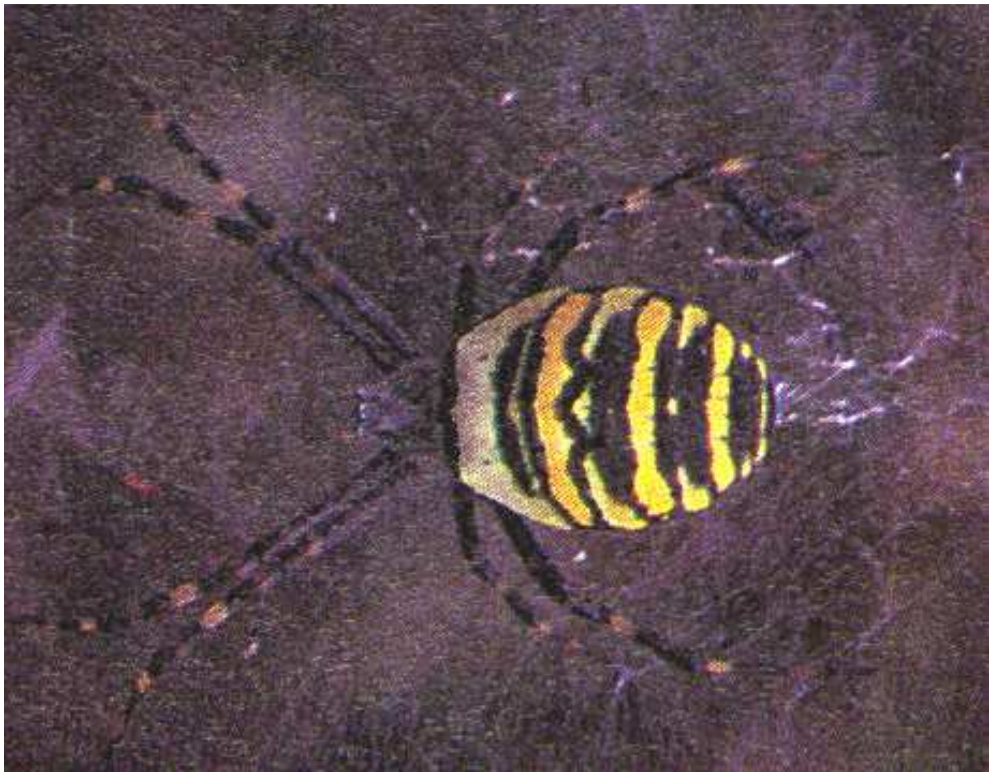


Figure 3.7 Une *Argiope bruennichi* [Jones83, p. 267]

Il devient alors possible de tisser des liens métaphoriques avec les programmes et leur comportement :

- Chaque araignée symbolisera chacune des fonctions d'un programme, pas uniquement en tant que prédateur[92], mais également dans le fait que les données que les araignées manipulent de manière interne sont équivalentes[93] aux éléments les constituant.

  De plus, les araignées sont des êtres autonomes qui, pour la plupart, ne vivent pas en groupes structurés : il en va de même des fonctions d'un programme, liées entre elles mais capables également d'exister indépendamment en tant que membres d'une librairie utilisée dans de multiples programmes.

- La toile, substrat de travail des araignées, sera considérée comme une représentation métaphorique de la *mémoire de travail*, retrouvant ainsi l'équivalence entre données manipulées et éléments constitutifs.

- Le fil comme moyen de communication symbolisera la concrétisation d'un échange de données entre deux fonctions, entre deux araignées.

[92] On parle souvent, au sujet des effets de bords, d'opérations destructrices.

[93] Dans leur représentation interne.

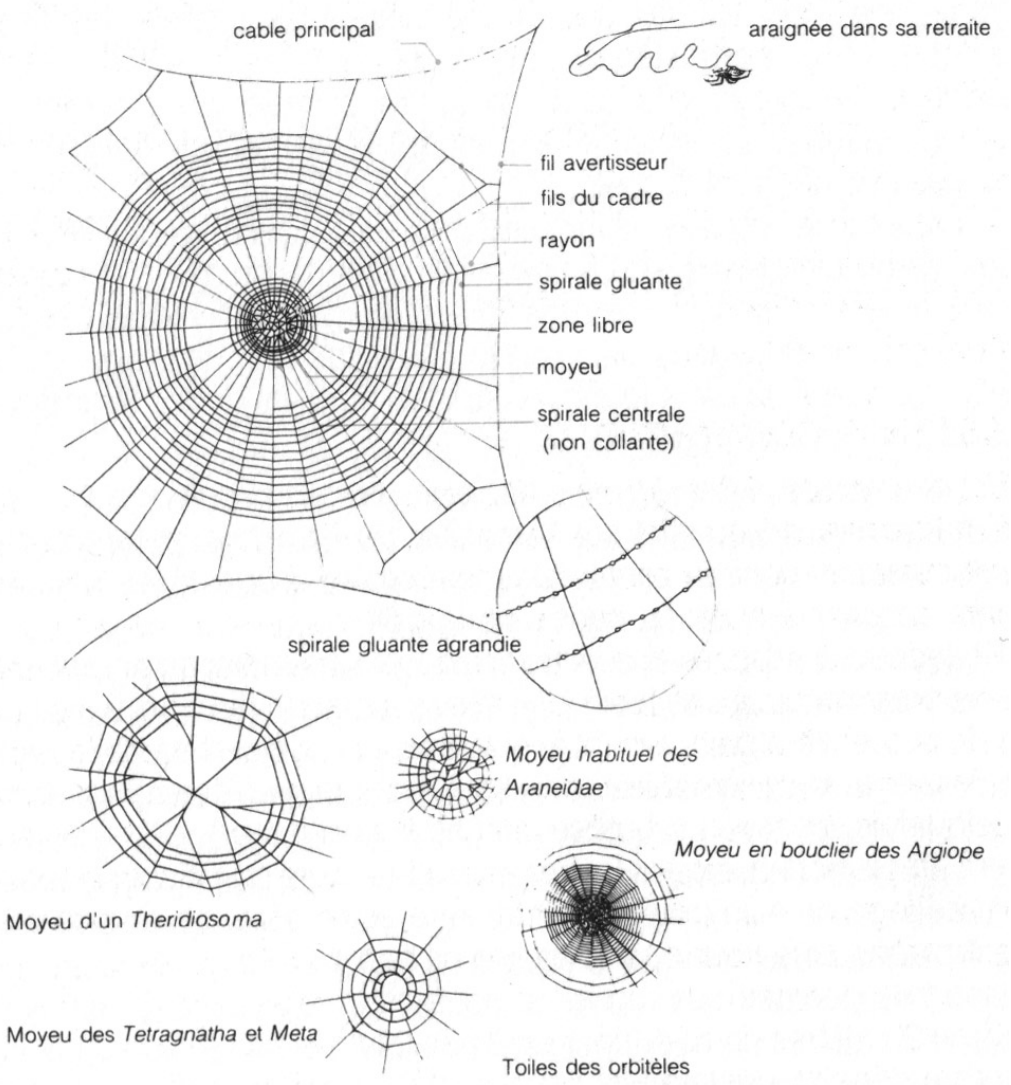


Figure 3.8 Description des toiles construites par les araignées orbitèles [Jones83, p.237]

- Le but d'une araignée est de survivre en se nourrissant. De même, le *but* de survie d'une fonction peut être considéré comme le fait d'*exister*, d'être activé.

Ainsi, cette représentation analogique de programmes s'attachera particulièrement à illustrer le comportement des différentes fonctions d'un programme pendant son exécution. La notion de comportement induit les questions suivantes :

- Quels sont les types de données échangées entre les fonctions ?
- Existe-t'il des relations particulières entre certaines fonctions du programme, relations d'exclusivité (telle fonction n'est appelée que par telle autre fonction), de regroupement (un sous-ensemble de plusieurs fonctions semble former un groupe fonctionnel, s'appeler mutuellement et être appelées par le reste du programme par un unique point d'entrée), ou encore d'inutilité (une, ou plusieurs fonctions ne semblent pas être utilisées) ?

Du fait du caractère animé de la représentation analogique (exprimant ainsi l'observation du comportement d'un programme à partir de son exécution), les conclusions quant à son comportement seront relatives à une exécution particulière et aux données utilisées pour celle-ci. Afin de pouvoir conclure de manière certaine sur les observations effectuées avec notre représentation analogique, il est nécessaire d'effectuer plusieurs observations résultant d'exécutions dont les données utilisées en entrée du programme couvrent l'ensemble de ses possibilités.

II.3.3.2 Présentation de la colonie d'araignées

II.3.3.2.1 Organisation spatiale

La figure 3.9 présente une colonie d'araignées de notre représentation analogique à deux étapes de sa vie. La première image représente la colonie telle qu'elle apparaît au moment de son *installation* sur la toile, c'est-à-dire avant que la représentation analogique ne s'anime, que le programme ne s'exécute. Avant de décrire en détail les caractéristiques des araignées, diverses remarques peuvent être mises en évidence :

- les araignées sont disposées en cercle autour de la toile, leur organisation ne semble pas être le fruit du hasard mais bien plutôt répondre à un ordre précis,
- les araignées regardent toutes vers le centre de la toile.

De même, sur l'image présentant la colonie à la fin de son animation (et donc à la fin de l'exécution du programme), en dehors du fait que l'angle de

prise de vue de l'image est différent de celui utilisé pour l'image initiale, on peut faire les remarques suivantes :

- les araignées (mais pas toutes) ont changé de place,
- les araignées ayant changé de place sont reliées aux autres par des liens colorés.

Ces observations visuelles sont à mettre en parallèle avec les liens unissant les programmes et la représentation analogique :

a) organisation spatiale des araignées sur la toile au moment de leur installation

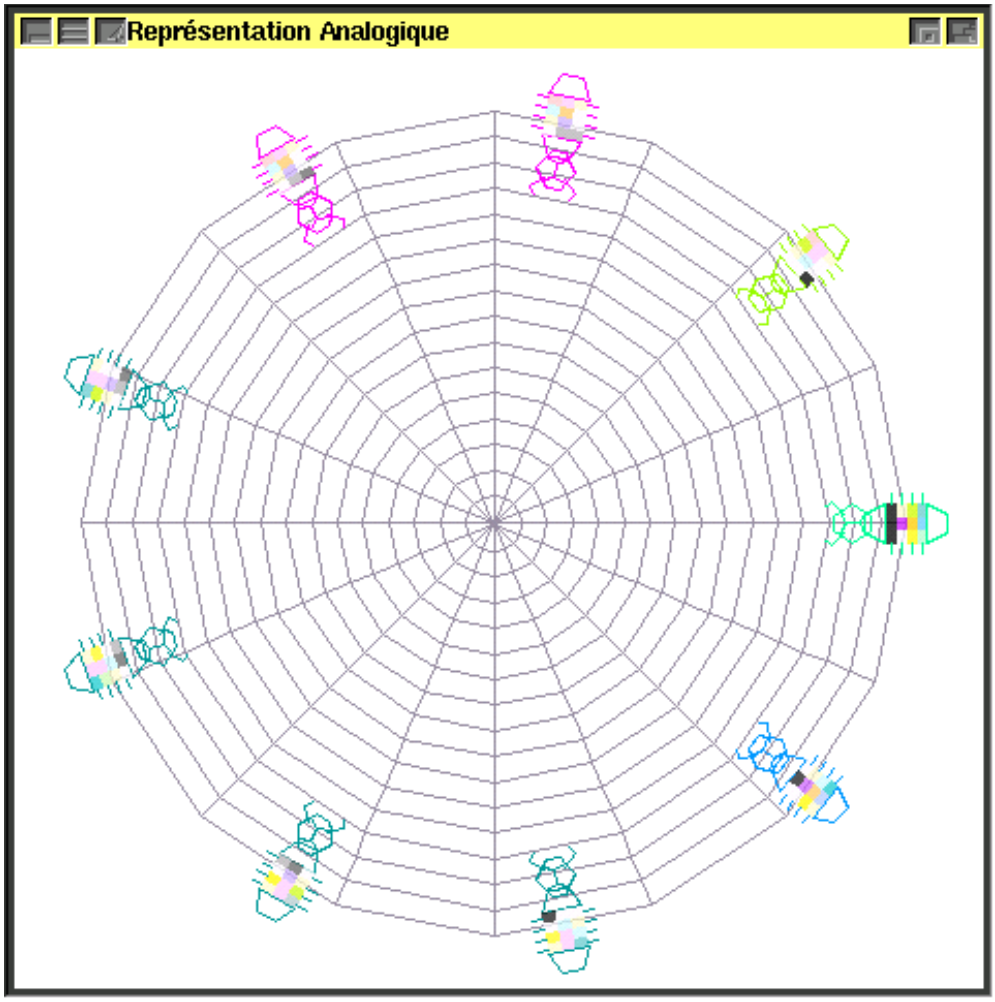
Comme nous l'avons dit précédemment, chaque araignée de la représentation analogique représente une fonction du programme. Ainsi, la disposition initiale des araignées sur la toile répond à un *parcours*[94] d'un point de vue regroupant l'ensemble des fonctions du programme. Dans le cas présent, cet ensemble de fonctions est structuré suivant la classification des fonctions par rapport au type d'instruction utilisé, classification décrite dans la représentation analogique de programmes par des cités (au paragraphe 4.2.3). Outre un regroupement des fonctions (elles sont disposées suivant le sens trigonométrique, la première araignée se situant à l'est de la toile), les couleurs qui caractérisent les quartiers des cités sont à nouveau utilisées dans cette représentation. Ainsi, l'utilisation de ces deux représentations en parallèle permet d'effectuer un lien visuel direct entre un élément de l'une et un élément de l'autre.

L'orientation initiale des araignées répond à un double sens :

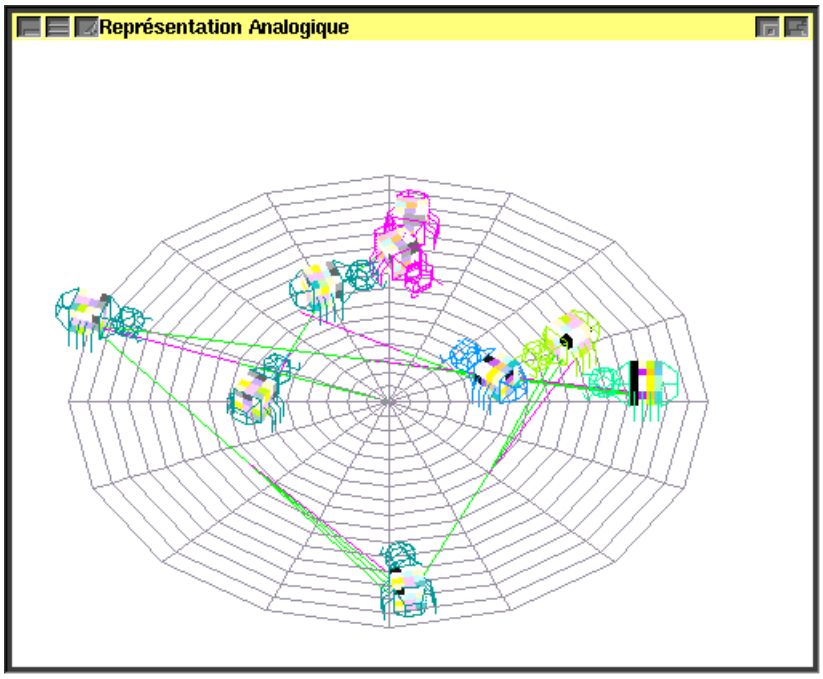
- le sens métaphorique avec les araignées : elles utilisent leur toile pour chasser et gardent ainsi en permanence un contact avec elle. Ce contact est, dans la nature, un contact physique direct (positionnement d'un membre sur un élément de la toile) ou indirect (construction d'un fil reliant l'araignée à la toile).
- le sens métaphorique du lien avec la représentation d'un programme : chaque fonction représentée appartient au programme et va (peut-être) être activée au cours de son exécution. L'activation d'une fonction sera ainsi le résultat d'un appel[95] d'une autre fonction du programme, d'une autre araignée présente sur la toile.

[94] Nous détaillerons dans le chapitre suivant les différents parcours possibles de l'ensemble des fonctions permettant de générer une représentation analogique.

[95] Excepté pour le point d'entrée du programme étant appelé directement par l'interprète.



(a) Une colonie d'araignées au moment de leur installation sur la toile



(b) La colonie à l a fin de l'animation

Figure 3.9 Une colonie d'araignées sur une toile

b) Organisation spatiale des araignées à la fin de l'animation

A la fin de l'exécution du programme, comme le montre la figure 3.9.a, la représentation analogique laisse les araignées à des emplacements différents et des liens apparaissent entre elles. Nous expliciterons en détail la signification des déplacements et des liens reliant les araignées au cours de l'exécution au paragraphe 1.3.3 mais nous pouvons analyser ici les déductions sur le comportement du programme qu'entraînent ces modifications graphiques.

- Une araignée n'ayant pas changé de place au cours de l'exécution, et se trouvant à la même position qu'à l'initialisation de la représentation graphique, indique que la fonction qu'elle représente n'a pas été activée au cours de l'exécution. Comme l'information du lien métaphorique entre un objet graphique et un élément du programme est accessible directement par sa sélection avec la souris, cette représentation analogique permet d'identifier les parties du programme inutilisées dans un contexte particulier (une exécution particulière du programme) bien qu'elles appartiennent à son flot de contrôle et qu'elles peuvent ainsi théoriquement être activées.
- En fait, les liens unissant deux araignées symbolisent l'échange de données entre les fonctions qu'elles représentent au moment de l'appel comme au moment du retour de l'appel. Ainsi, les liens restant à la fin indiquent le dernier chemin d'appel existant entre les différentes fonctions du programme.

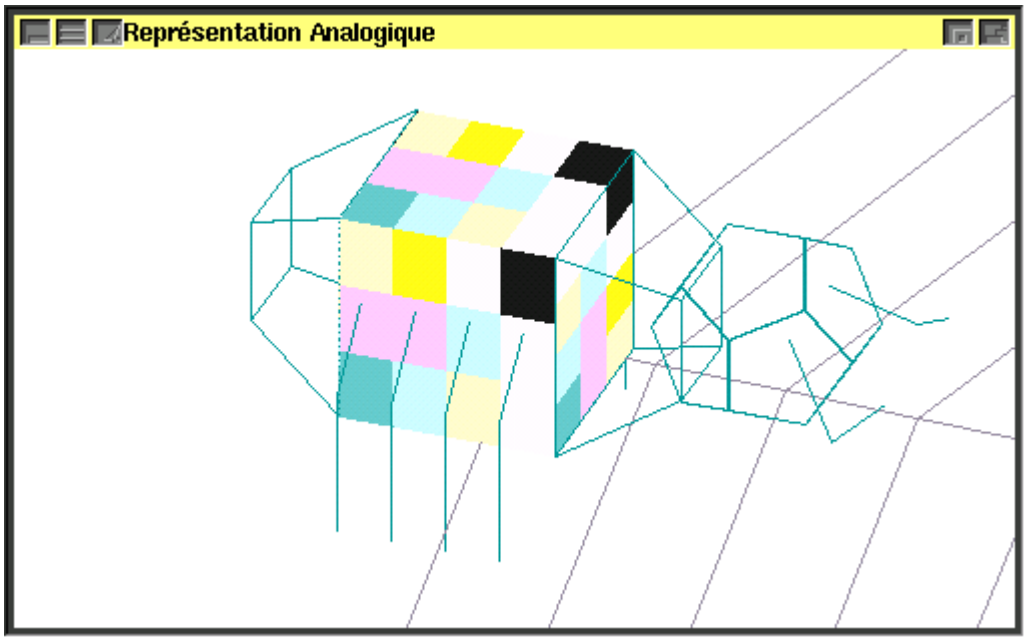


Figure 3.10 Zoom sur une araignée de la figure 3.9

II.3.3.2.2 Caractéristiques d'une araignée

La figure 3.10 présente un zoom sur une araignée de la figure 3.9. Reconnaissable comme appartenant à la famille des araignées du fait de la présence de huit pattes et de mandibules, on peut toutefois caractériser comme schématique cette représentation graphique d'insecte. Ce choix de simplicité dans la représentation graphique est principalement dû au souci d'efficacité de l'animation de la représentation : des objets graphiques trop complexes auraient abouti à un temps d'affichage trop important.

Nous avons vu précédemment que la couleur des araignées était relative à leur appartenance à une classe de fonctions tel que celles-ci ont été présentées dans le paragraphe 3.2.3. Cette couleur apparaît dans tous les éléments de l'araignée, mis à part la partie centrale de son corps. Afin d'augmenter les éléments graphiques servant à distinguer visuellement les différentes araignées, nous avons choisi d'utiliser comme motif de texture du corps un pavage coloré basé sur une énumération des primitives utilisées dans les fonctions. Ce pavage, correspondant aux tuiles des maisons de la représentation analogique de programme par des cités, afin de conserver un élément graphique supplémentaire ayant pour fonction d'effectuer un lien visuel direct entre les deux représentations analogiques.

II.3.3.3 Animation de la représentation analogique

L'objet principal de cette représentation analogique est la visualisation du comportement des programmes à travers l'animation d'araignées sur une toile. Cette animation se compose de deux éléments :

- le déplacement des araignées
- la mise en place de liens unissant les araignées

Ces deux éléments graphiques, que nous allons maintenant décrire, permettent de visualiser d'une part la formation de groupes fonctionnels dans le programme et d'autre part les échanges de données entre les différentes fonctions. Ainsi, les descriptions des éléments actifs lors des animations graphiques qui vont suivre se basent sur une exécution d'un programme particulier. Ce programme, dont les sources se trouvent à l'annexe II.4 de ce mémoire, résout le problème des tours de Hanoi en incluant la visualisation graphique du mouvement des disques au cours de la résolution (la figure 3.11 présente cette visualisation à deux étapes de la résolution).

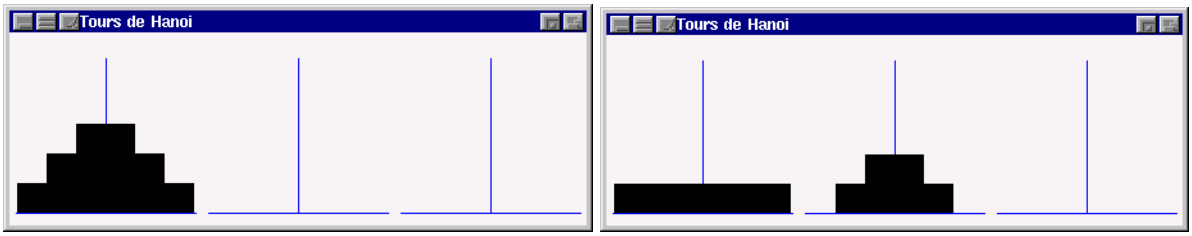


Figure 3.11 Visualisation graphique du déroulement de la résolution des tours de Hanoi

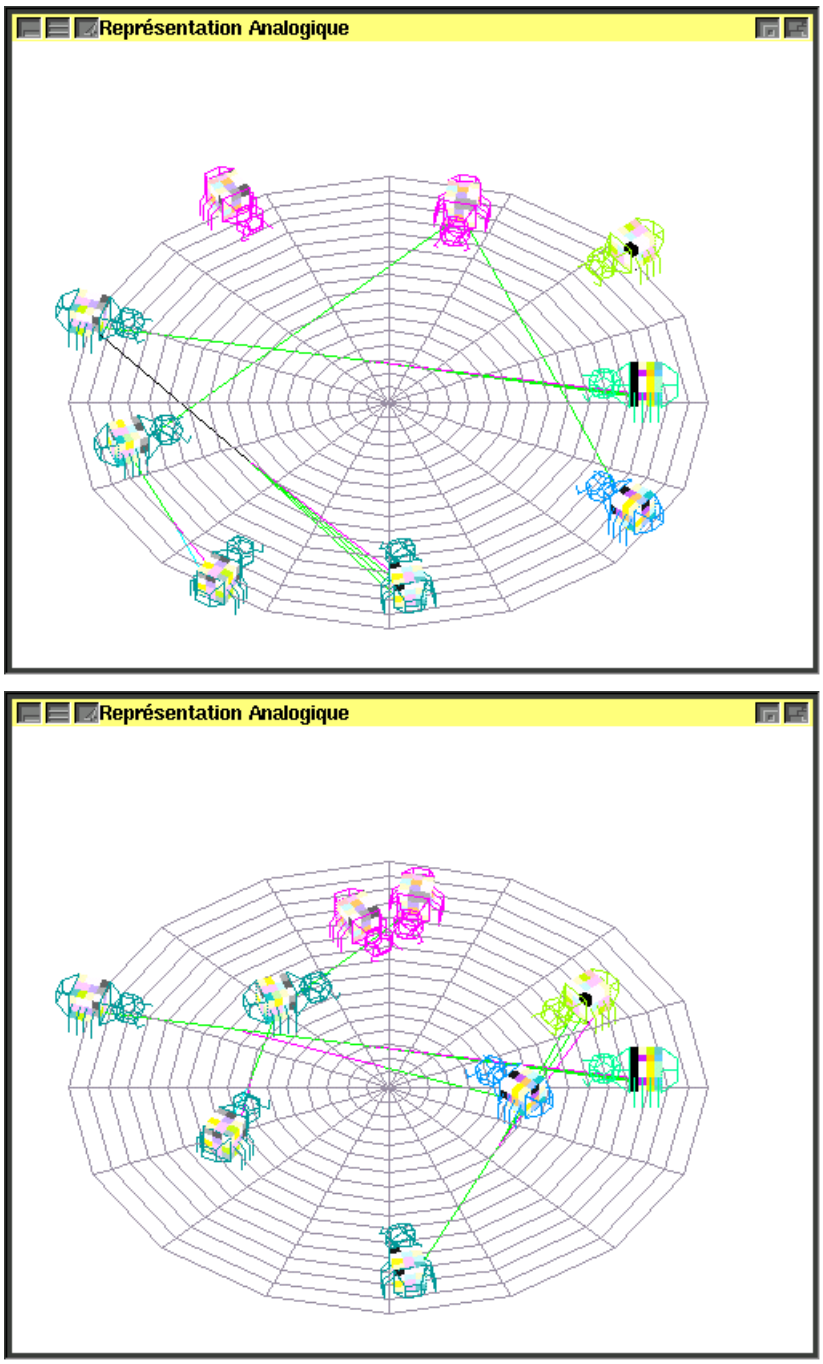


Figure 3.12 Etat de la représentation analogique aux deux étapes du programme Hanoi visualisé dans la figure 3.11

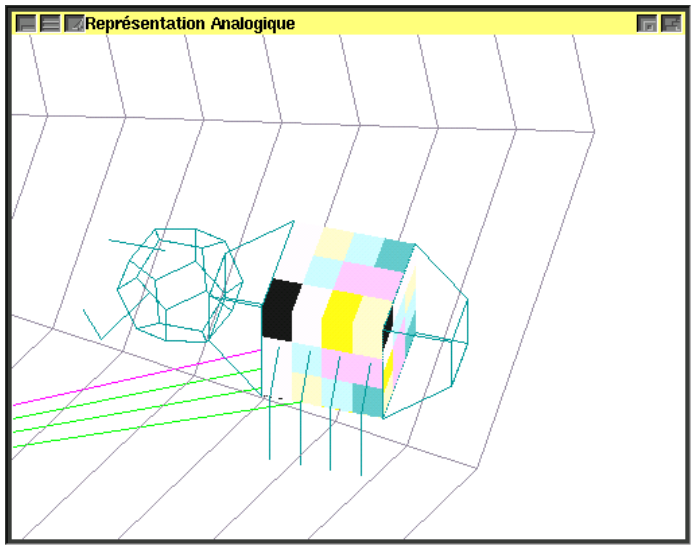
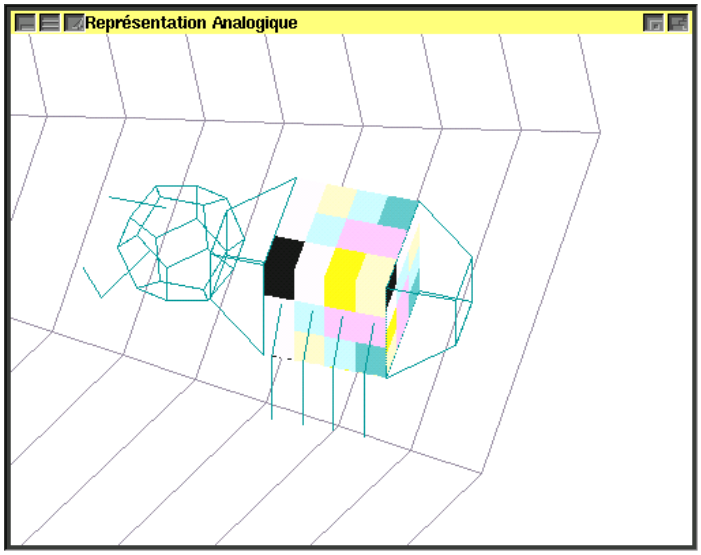


Figure 3.13 Zooms sur une araignée à la première étape



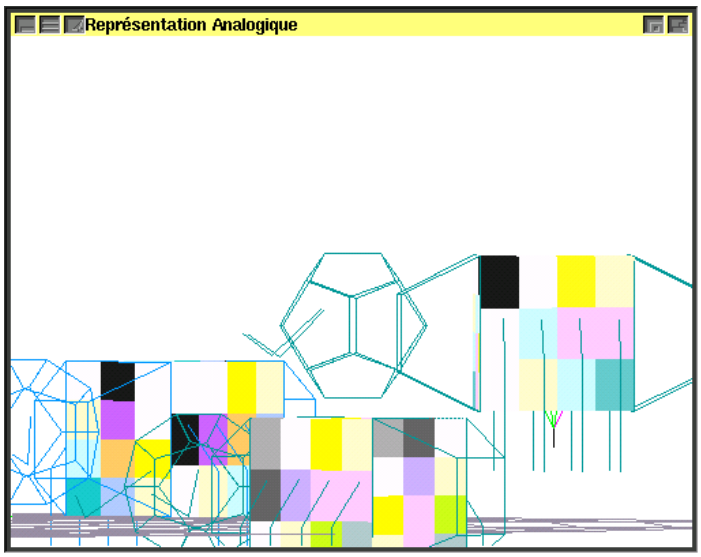


Figure 3.14 Zooms sur une araignée à la seconde étape (vue sous deux angles différents)

II.3.3.3.1 Déplacement des araignées

Comme nous pouvons le voir sur les images des figures 3.12, 3.13 et 3.14, les araignées peuvent se déplacer de deux manières différentes :

1) un déplacement horizontal vers une autre araignée,
2) un déplacement vertical.

Ces déplacements correspondent à deux types d'événements de l'exécution du programme :

1) l'appel d'une fonction depuis une autre fonction. Dans ce cas, le déplacement est irrémédiable : à la fin de l'exécution de la fonction, l'araignée ne reprendra pas sa position initiale. Ainsi, plus une fonction en appelle une autre et plus les araignées qui les représentent se rapprochent. Ce premier type de déplacement permet de voir se construire des groupes de fonctions par attirance mutuelle et ce, pour deux fonctions ou plus. A la fin de l'exécution, des groupes d'araignées apparaissent. Ils représentent un ensemble de fonctions fortement liées les unes aux autres par de nombreux appels mutuels pendant l'exécution : indication de la présence de groupes fonctionnels. Par exemple, la figure 3.15 montre un zoom effectué sur ce qui apparaissait comme un regroupement sur la configuration finale des araignées. En fait, les araignées de ce groupe représentent les fonctions du programme Hanoi permettant la mise à jour de la représentation graphique des tours.
2) Un appel récursif. En effet, dans le cas d'un appel récursif, il n'est pas possible de rapprocher deux araignées puisqu'une seule est en jeu. Afin de mettre en évidence visuellement ces appels, les araignées se déplacent verticalement vers le haut à l'entrée de l'appel récursif, redescendant au moment de la sortie de la fonction (la figure 3.14 montre deux vues d'une araignée représentant une fonction pendant un appel récursif). Ce dernier déplacement, s'il est temporaire (la fonction reprenant sa position initiale à la sortie de l'appel récursif), permet toutefois de distinguer clairement au cours de l'exécution les fonctions récursives des autres.

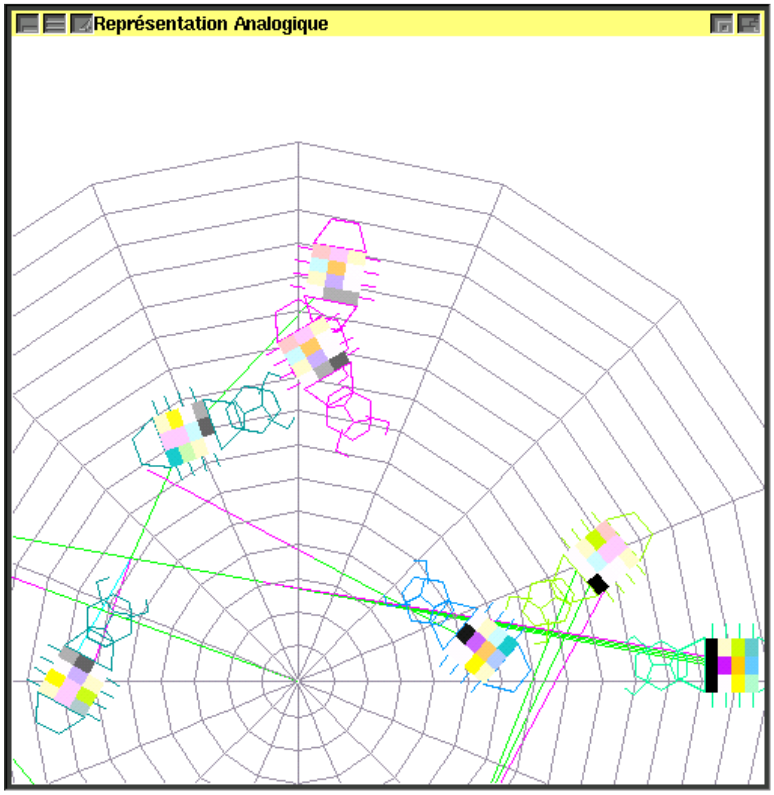


Figure 3.15 Détection visuelle d'un groupe de fonctions

II.3.3.3.2 Echanges entre les araignées

En parallèle du déplacement attirant deux araignées l'une vers l'autre, un lien se tisse entre les protagonistes. Ce lien est composé de deux parties distinctes (cf. figure 3.12 page 81) :

1) une première partie peut être composée de plusieurs brins de couleurs différentes. Cette partie représente graphiquement l'envoi des données de la fonction appelante vers la fonction appelée. Le nombre de brins désigne le nombre de paramètres attendus par la fonction et leur couleur signifie le type de données envoyées (la classification des couleurs s'établit suivant celle décrite au paragraphe 3.2.3 page 61).

2) une seconde partie qui est, au moment de l'appel, de couleur blanche et qui change de couleur au moment de la sortie de la fonction. Cette partie signifie la valeur retournée par la fonction appelée dont le type n'est pas connu avant la sortie de celle-ci.

Cette utilisation des fils comme moyen de communication entre araignées est une extrapolation de ce qui ce passe dans la nature : les araignées ne communiquent en effet pas directement en utilisant un fil entre deux individus mais, dans le cas d'une colonie comme pour les individus solitaires, le moyen de communication, d'information est bien soit un fil relié à la toile,

soit la toile elle-même[96].

Au niveau de l'effet visuel de la représentation analogique, l'utilisation de tels liens permet de suivre la progression de l'exécution (l'affichage, la modification et le déplacement de ces liens entre individus indiquent la progression dans le flot de contrôle) ou, par exemple, l'imbrication des appels de fonctions (la couleur blanche du fil de retour indiquant une fonction en cours d'exécution).

II.3.3.4 Conclusion

Le but annoncé de cette représentation analogique, qui fut en fait élaborée pour tendre à sa réalisation, était de permettre la visualisation des liens unissant les différentes parties d'un programme et d'aider à la description de ces liens. Comme nous l'avons vu, ce type de représentation permet de distinguer, au cours de l'exécution comme a posteriori, la formation de groupes de fonctions, l'enchaînement effectif des fonctions entre elles et, finalement, de distinguer les fonctions n'entrant pas en jeu dans le cours d'une exécution.

L'apport de l'utilisation de représentations analogiques, outre le côté esthétique, est de pouvoir suivre l'évolution de l'ensemble des fonctions du programme sur une unique représentation graphique. Toutefois, leur application sur des programmes de grande taille (mettant ainsi en jeu de grands nombres de fonctions), semble limitée du fait de la complexité des liens pouvant apparaître ou des combinaisons de déplacements rendant difficile la détection de groupes fonctionnels. Pour pouvoir traiter de tels cas, il serait nécessaire d'utiliser un pré-traitement sur le programme pour parvenir à représenter plusieurs fonctions par une unique araignée et à visualiser ainsi le dialogue entre groupes fonctionnels.

Du point de vue de la métaphore utilisée, il devrait être possible d'étendre l'utilisation d'insectes comme éléments de visualisation des programmes dans des environnements de programmation analogiques du fait, comme nous l'avons dit en introduction de ce chapitre, de la possibilité d'attachement d'un petit nombre de règles de vie suffisant à la conservation d'un lien analogique *simple*, de mémorisation aisée. De plus, ce type d'objet semble être un bon candidat dans l'élaboration d'une interface de programmation à partir d'analogies en prenant, par exemple, pour base le lien entre fonctionnalité et espèce, spécificité dans la fonctionnalité et différentes races d'une espèce.

[96] Comme c'est le cas pour les *Araignées Sociales*, dans leur approche d'une proie sur leur toile : elles marchent de manière synchronisée afin de ne pas masquer les vibrations de leur but.

II.3.4 Animation d'algorithmes avec Zeugma

II.3.4.1 Animation analogique d'algorithmes

L'animation d'algorithmes est un sujet particulièrement étudié dans le domaine de la visualisation de programmes et il est la source de nombreux systèmes existant aujourd'hui. Dans le chapitre IV de ce mémoire, nous comparons notre système avec des systèmes d'animations d'algorithmes existant aujourd'hui. Comme nous le précisons dans cette comparaison, notre critique principale envers ces systèmes et les représentations qu'ils proposent, quelle que soit la manière dont les animations graphiques sont spécifiées, est une sorte de glissement de l'objet fondamental de cette discipline. La recherche de représentations animées illustrant toutes les étapes d'un algorithme est déconsidérée par rapport à des aspects plus techniques comme :

- le moyen d'attacher une représentation ou une animation avec l'implémentation d'un algorithme,
- la construction d'outils sophistiqués permettant de décrire des représentations graphiques.

Comme nous le montrons dans notre critique de ces systèmes, ils aboutissent en général à des systèmes de visualisation des données qui ne se sont pas penchés sur la question de savoir si tous les différents aspects d'un algorithme, toutes les étapes qu'il décrit, sont bien représentés. Souvent, bien au contraire, les visualisations d'algorithmes ne montrent pas leur fonctionnement mais ses effets.

La représentation analogique d'algorithmes que nous allons détailler ici s'attache à montrer les différentes étapes constituant des algorithmes de tris de listes de nombres. Nous proposons une représentation unique, basée sur la conjonction de l'animation de l'évolution de la résolution et de la persistance de son historique. En effet, la plupart des visualisations d'algorithmes de tris proposées aujourd'hui ne prennent en compte que l'un ou l'autre de ces aspects alors qu'une visualisation efficace se doit, selon nous, d'inclure ces deux aspects simultanément. La figure 3.16, que nous expliciterons dans ce chapitre, montre deux vues de la représentation finale du tri par fusion de notre système.

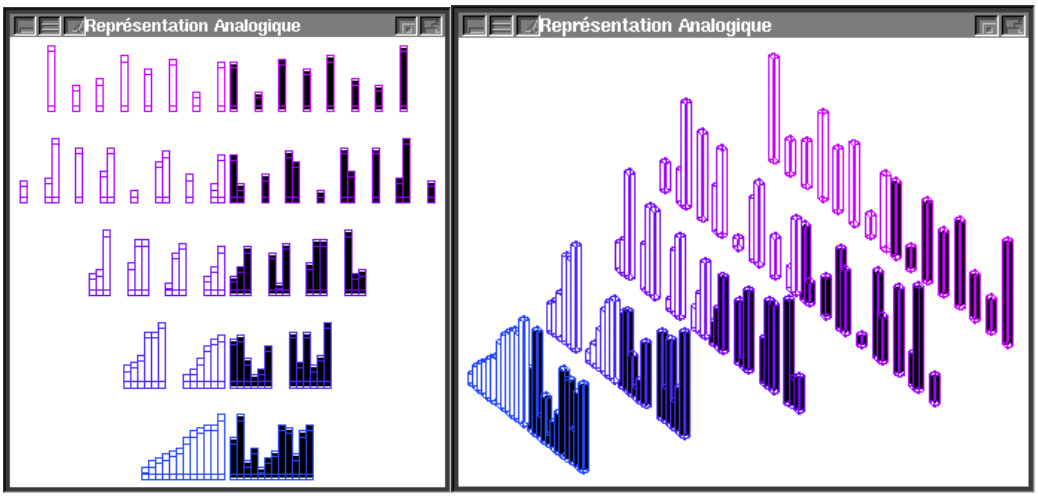


Figure 3.16 Deux vues du tri par fusion

II.3.4.2 Description de la représentation analogique

La représentation analogique que nous avons implémentée pour illustrer le fonctionnement de différents algorithmes de tri se base sur l'utilisation d'histogrammes, représentation communément admise depuis Tufte [Tufte83] comme la plus efficiente pour visualiser des listes de nombres. Afin de visualiser la progression dans la résolution et de présenter son historique dans une même image, nous utilisons les trois dimensions de l'espace suivant le schéma présenté dans la figure 3.17. Le niveau de récursion apparaît du bas vers le haut de la fenêtre de visualisation. Les différentes données sont représentées horizontalement et la hauteur de leur représentation indique leur valeur.

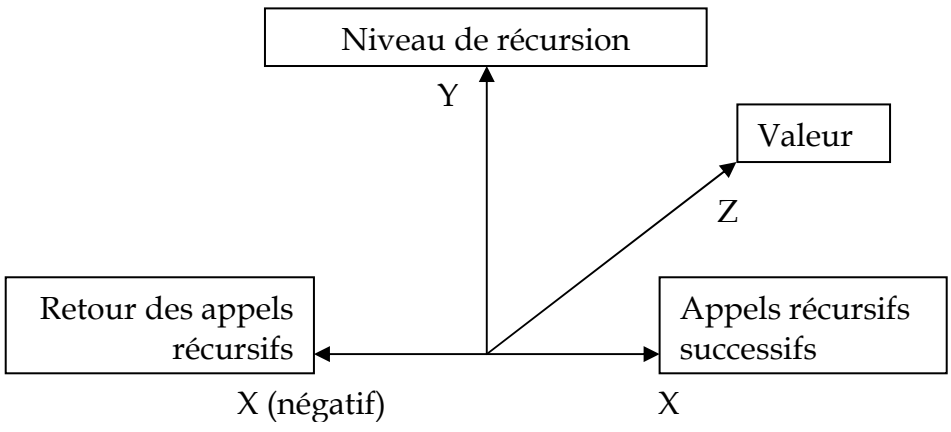
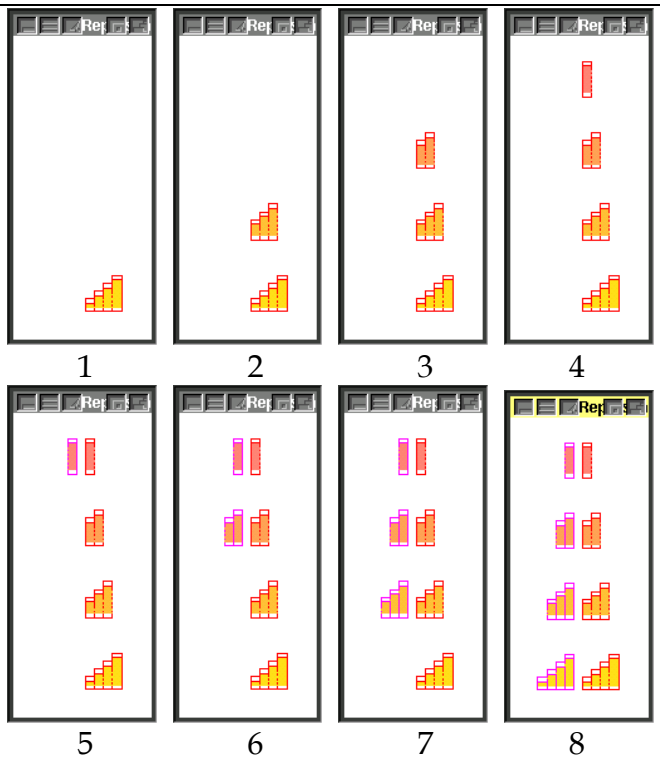


Figure 3.17 Schéma explicatif de l'animation analogique



(1) appel initial

(2) à (4) appels récursifs

(3) à (8) retour des appels

Figure 3.18 Vues successives de l'animation de la copie d'une liste de quatre nombres

```
(de copie (l)
  (if (null l) nil
    (cons (car l) (copie (cdr l)))))
```

Figure 3.19 Code source de la copie d'une liste

Pour illustrer cette représentation sur un programme simple, la figure 3.18 présente l'application de notre animation sur la copie d'une liste de quatre nombres. Chacune des étapes de la résolution apparaît successivement, permettant une distinction visuelle entre les appels récursifs en entrée de leur sortie ainsi que les valeurs manipulées à chacune des étapes. Nous pouvons déduire de cette animation que le programme de copie de liste construit le résultat final après avoir totalement parcouru la liste et que son implémentation doit ainsi commencer par un appel récursif avec la liste privée de son premier élément puis, lorsque la liste est vide, retourner la concaténation du premier élément avec la copie du reste. De fait, ces déductions visuelles correspondent à l'exécution du programme présenté dans la figure 3.19.

Pour construire cette représentation analogique, notre système Zeugma a joué le rôle d'interface entre les programmes et la construction des représentations graphiques des étapes successives. En effet, nous avons voulu

construire une animation générique d'algorithmes pouvant être appliquée à une série de programmes, sans avoir à modifier ni les programmes, ni l'animation, pour l'adapter à telle ou telle implémentation. Le code source de la définition de cette analogie est présenté dans l'annexe I.3 de ce mémoire. Notons néanmoins que cette animation peut être sujet aux mêmes critiques que celles énoncées page 86.

```
; définition du tri rapide
(de tri_rapide (l)
  (if l
     (append (tri_rapide (partition (car l) (cdr l) 'le))
                (cons (car l)
                      (tri_rapide (partition (car l) (cdr l) '>))))))

; partition d'une liste (lst) par rapport à un pivot (val) et un opérateur (op).
(de partition (val lst op)
  (cond
   ((null lst) nil)
   ((op val (car lst)) (cons (car lst) (part val (cdr lst) op)))
   (t (part val (cdr lst) op))))
```

Figure 3.20 Code source de l'implémentation du tri rapide

II.3.4.3 Application au tri rapide

Le premier exemple d'application de notre animation analogique à des algorithmes de tri est le tri rapide. L'implémentation de cet algorithme utilisé pour cette animation est présentée dans la figure 3.20. Nous avons choisi de lier l'animation de l'algorithme avec l'accès (en entrée ou en sortie) de la fonction *tri_rapide*. La figure 3.21 présente le film d'une partie de l'animation et la figure 3.22 deux vues de l'image finale de l'exécution.

Les différentes étapes de l'algorithme, que notre animation doit mettre en évidence, sont :

1) la construction, à chaque étape, de deux partitions de la liste donnée en argument. La première contiendra les éléments plus petits que le premier élément de la liste et la seconde les éléments plus grands ou égaux à celui-ci.
2) Chaque nouvel appel récursif travaillera sur la partition reçue.

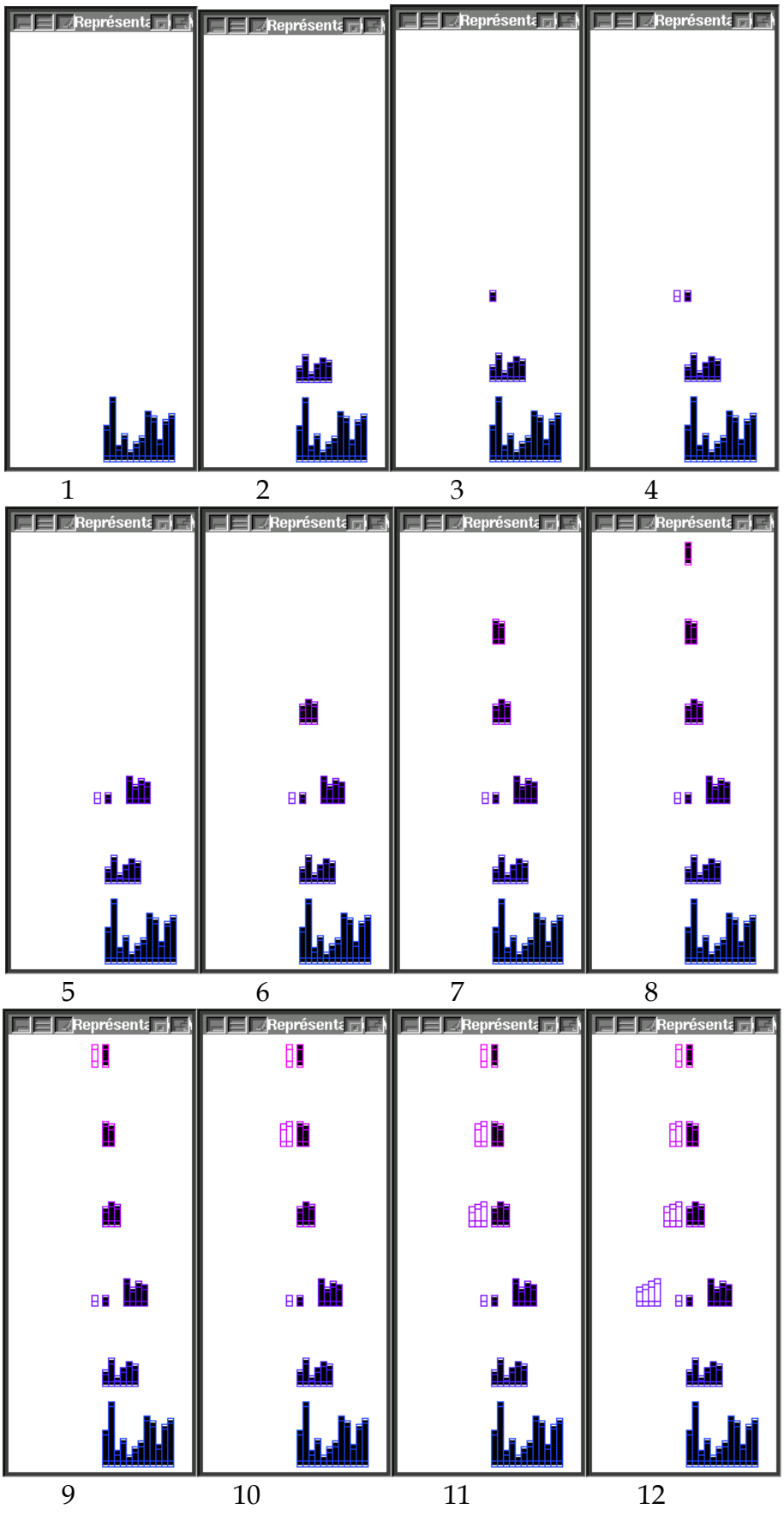
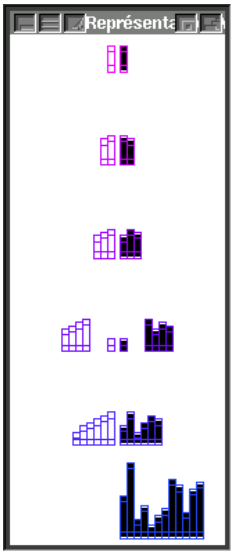


Figure 3.21 Film d'une partie du tri rapide (début)



13

Figure 3.21 Image finale du film d'une partie du tri rapide

Détaillons maintenant le déroulement du film :

- les proportions étant conservées d'un niveau de récursion à l'autre, nous pouvons voir que l'élément présent au troisième niveau de l'étape 3 est plus petit que le premier du second niveau de l'étape 2. De plus, les nouveaux éléments apparaissant à ce même niveau à l'étape 5 sont tous plus grands que ce dernier. Le programme prend visiblement en considération les plus petits éléments avant de traiter les plus grands.
- Les éléments présents dans la partie *entrée* (droite) du troisième niveau de l'étape 5 désignent les deux partitions des éléments du second niveau. On peut alors distinguer uniquement cinq éléments alors que le niveau précédent en contenait six. Il apparaît visuellement que l'élément manquant est le premier, utilisé comme pivot. Cet élément réapparaît toutefois en sortie du tri (partie gauche) des éléments de ce niveau, à l'étape 13.
- Le positionnement selon l'axe des abscisses, relatif au nombre d'appels récursifs ayant eu lieu à un niveau donné, justifie l'utilité de l'animation par rapport au simple affichage de l'historique complet : elle permet de distinguer la provenance des éléments. Par exemple, sur l'image de l'étape 13, les éléments du quatrième niveau ne sont pas issus des éléments centraux du troisième mais, comme le montrent les étapes 5, 6, 11 et 12, du travail de l'algorithme sur la seconde partition des éléments du deuxième niveau.
- La construction du résultat, une liste de nombres visiblement triée, est effectuée après que la liste argument ait été réduite à un seul élément (visible aux étapes 4 et 8). Puis, par sorties successives,

l'élément pivot est placé entre les deux partitions (étapes 9 à 13).

Ainsi, de la combinaison d'une animation et de la persistance de l'historique nous avons pu déduire que :

1) les éléments d'un niveau de récursion sont successivement partitionés en *plus petits* et *plus grands* par rapport à l'élément présent en tête,
2) la partition a lieu tant que la liste contient plus d'un élément,
3) la construction de la liste triée est le résultat de la concaténation du tri des éléments plus petits, du pivot et du tri des éléments plus grands.

Cette description correspond bien à la description du tri rapide ainsi qu'à son implémentation.

Ces deux vues de la représentation analogique, présentées dans la figure 3.22, prises à la fin de l'exécution du programme, permettent de distinguer visuellement l'utilisation des trois dimensions dans la construction de la représentation. De plus, le tri de la seconde partition des éléments du premier niveau (absents dans la figure 3.21) illustre bien le fonctionnement de l'algorithme (par le passage du quatrième au cinquième niveau en entrée et en sortie par exemple).

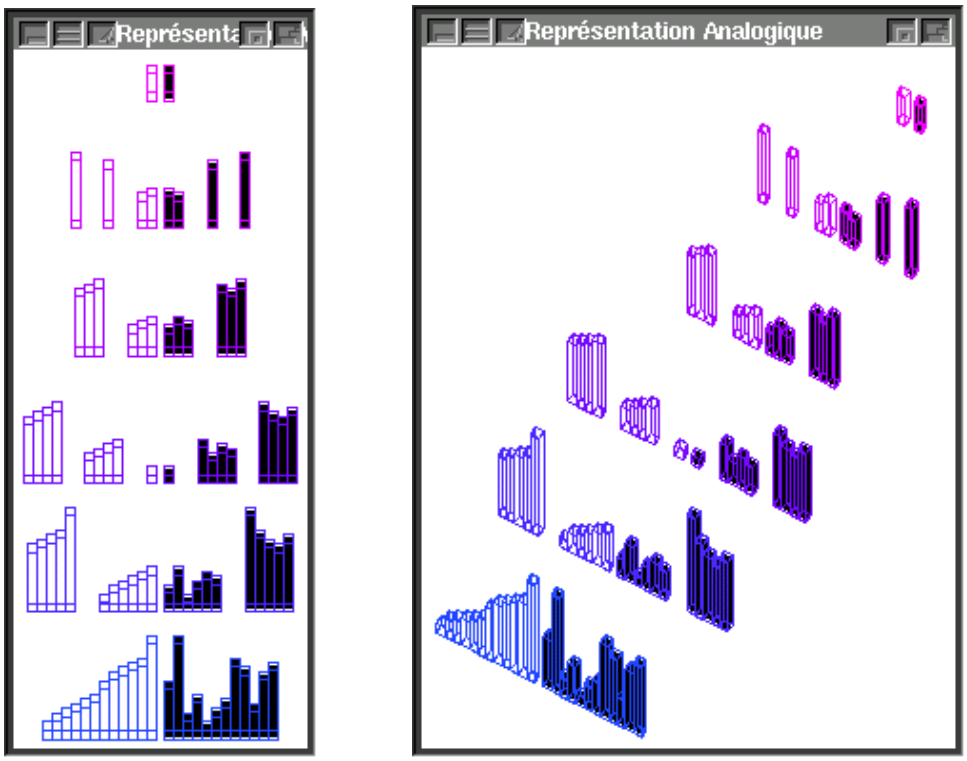


Figure 3.22 Deux vues de l'image finale du tri rapide

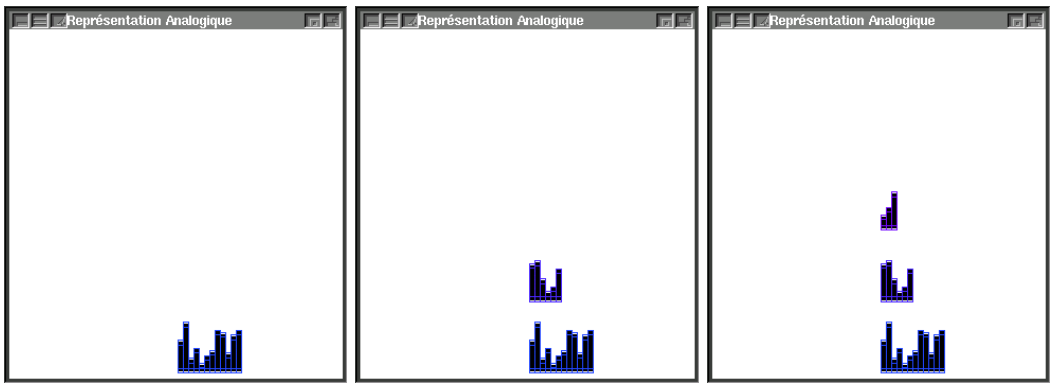
II.3.4.4 Application au tri par fusion

```
; définition du tri par fusion
(defun tri_fusion (l)
  (if (null (cdr l)) l
    (let (l1 (separe l)) ; l1 contiendra les deux parties de la séparation de l
      (fusion (tri_fusion (car l1))
       (tri_fusion (cdr l1))))))
; construit la séparation en deux listes égales d'une liste l
(defun separe (l l1 l2)
  (if (null l) (cons l1 l2)
    (separe1 (cdr l) (cons (car l) l1) l2)))

(defun separe1 (l l1 l2)
  (if (null l) (cons l1 l2)
    (separe (cdr l) l1 (cons (car l) l2))))

; fusionne, de manière ordonnée, deux listes
(defun fusion (l1 l2)
  (cond
   ((or (null l1) (null l2)) (or l1 l2))
   ((< (car l1) (car l2)) (cons (car l1) (fusion (cdr l1) l2)))
   (t (cons (car l2) (fusion l1 (cdr l2))))))
```

Figure 3.23 Code source du tri par fusion

Le second tri pris en exemple est le tri par fusion. L'intérêt de ce choix réside dans le fait que cet algorithme n'est en soi pas très différent du tri rapide. L'implémentation utilisée est présentée dans la figure 3.23, une partie du film de son exécution dans la figure 3.24 et la figure 3.16 présente deux vues de l'image finale de son exécution.

De la même manière que pour le tri rapide, voyons si notre visualisation permet de comprendre l'algorithme sous-jacent au tri par fusion.



1 2 3

Figure 3.24 Partie du film du tri par fusion (début)

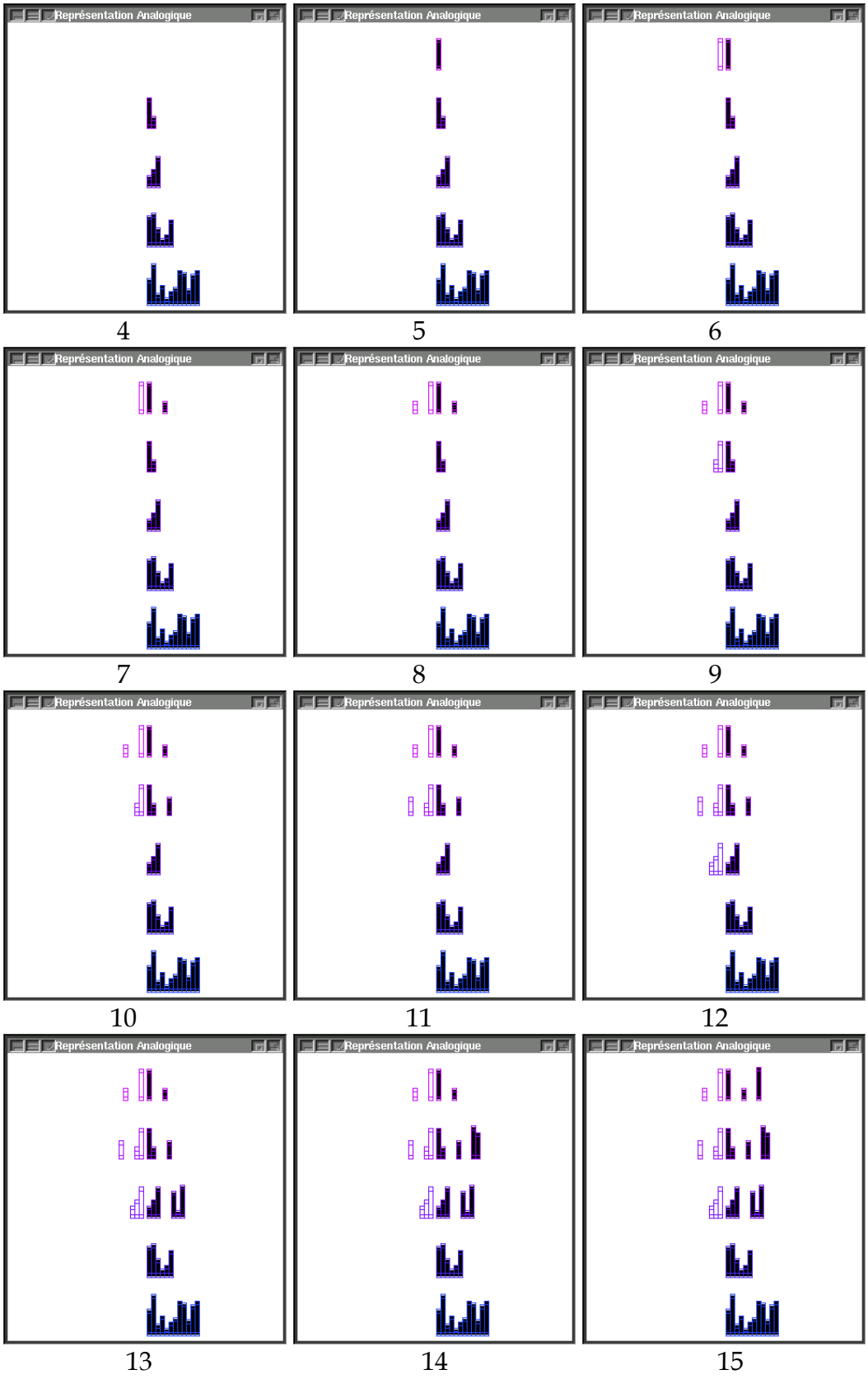


Figure 3.25 Suite du film du tri par fusion

Figure 3.25 Fin du film du tri par fusion

En remarque préalable à l'examen de ce film de l'animation du tri par fusion, on peut remarquer que, même s'il fonctionne en apparence comme le tri rapide vu précédemment (chaque niveau de récursion effectue une partition sur les éléments manipulés par le niveau inférieur), le nombre d'étapes nécessaires pour aboutir au même résultat (le tri de la moitié de la liste de départ) est nettement plus important que pour le tri rapide (24 par rapport à 13). Examinons ce que nous pouvons déduire visuellement de ce film :

- Puisque, comme dans le tri rapide, une partition est effectuée sur la liste de nombres triée, voyons si la représentation graphique animée nous donne des indications par rapport au moyen mis en

œuvre pour effectuer cette partition. L'image numéro 2, par exemple, semble montrer que la partition ne s'effectue pas selon un critère lié à une valeur particulière mais par rapport à la position dans la liste : le dernier élément de la liste du second niveau est visiblement le premier de la liste du premier niveau. De plus, l'élément qui le précède ne semble pas suivre le premier du premier niveau mais se trouve être en troisième position. Ceci est confirmé par la présence, dans l'image 24, du même nombre d'éléments au premier niveau et au second : les éléments sont pris à raison d'un sur deux et, de plus, transmis en ordre inverse. Cette inversion apparaît également dans le passage du troisième au quatrième niveau des images 3 et 4, alors que l'image 10 illustre, pour ce même passage de niveau, la prise en compte d'un élément sur deux dans la partition des listes.

- Le second point est la construction du résultat : de la liste triée. Dans le cas où l'une des listes ne contiendrait qu'un élément, la construction du résultat semble être effectuée par simple concaténation (par exemple, le résultat construit dans l'image 12 est visiblement la concaténation des éléments du niveau supérieur de l'image 11). Dans le cas où les listes en jeu dans la construction du résultat contiendraient plus d'un élément, comme dans la construction du résultat de l'image 23, il semblerait que la concaténation soit *intelligente* dans le sens où les listes ne sont pas seulement mises bout à bout : la liste finale place à la bonne position les éléments des deux listes.

Les éléments déduits de l'observation du film de l'animation illustrent bien les points suivants de l'algorithme de tri par fusion :

1) le tri commence par deux appels récursifs[97] avec chacun la moitié des éléments de la liste donnée en argument. Ces deux listes sont construites[98] en prenant en compte un élément sur deux et en construisant des listes inversées par rapport à la liste originale.
2) La liste triée est le résultat d'une fusion[99] ordonnée du résultat de ces appels récursifs.

Par contre, le point que cette représentation semble décrire de manière moins détaillée est le processus en jeu dans la fusion des listes afin de construire le résultat. Ceci est dû au fait que cette animation se base sur l'entrée et la sortie de la fonction *tri_fusion* du programme et qu'il faudrait s'attacher à visualiser le comportement de la fonction *fusion*, responsable de la fusion des listes, afin de voir clairement le processus alors en jeu. Mais, même avec la visualisation présentée, il est possible de déduire une concaténation prenant

[97] Par la fonction lisp *tri_fusion* de l'implémentation présentée à la figure 3.24.

[98] Par la fonction lisp *separe*.

[99] Par la fonction lisp *fusion*.

en compte chaque élément présent dans les listes, ce qui est synonyme de fusion.

II.3.4.5 Application au tri par insertion

```
; définition du tri par insertion
(defun tris_ins (l)
  (if (null l) nil
    (insert (car l) (tris_ins (cdr l)))))

; définition de l'insertion d'un élément à la bonne place dans une liste ordonnée.
(defun insert (e l)
  (cond
   ((null l) [e])
   ((< e (car l)) (cons e l))
   (t (cons (car l) (insert e (cdr l))))))
```

Figure 3.26 Code source du tri par insertion

Le dernier algorithme de tri présenté ici est le tri par insertion. Le code source de l'implémentation étudiée est présenté dans la figure 3.26 et une partie du film de l'animation dans la figure 3.27.

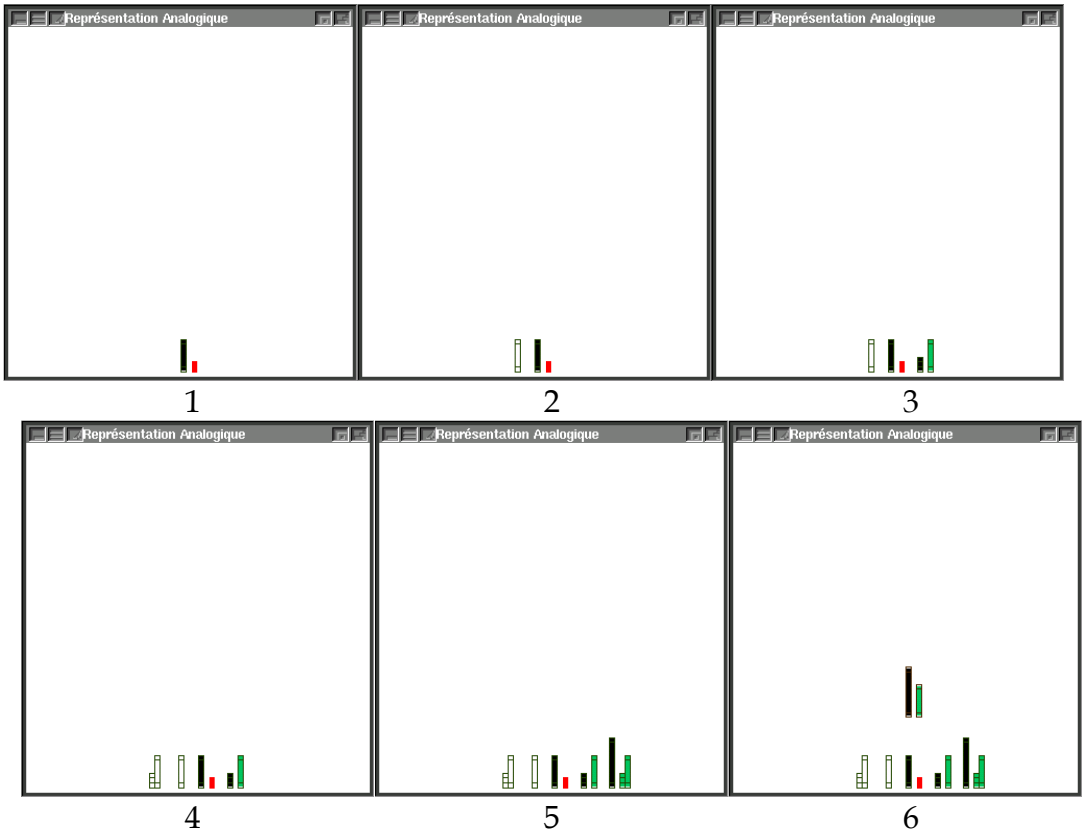


Figure 3.27 Film du tri par insertion

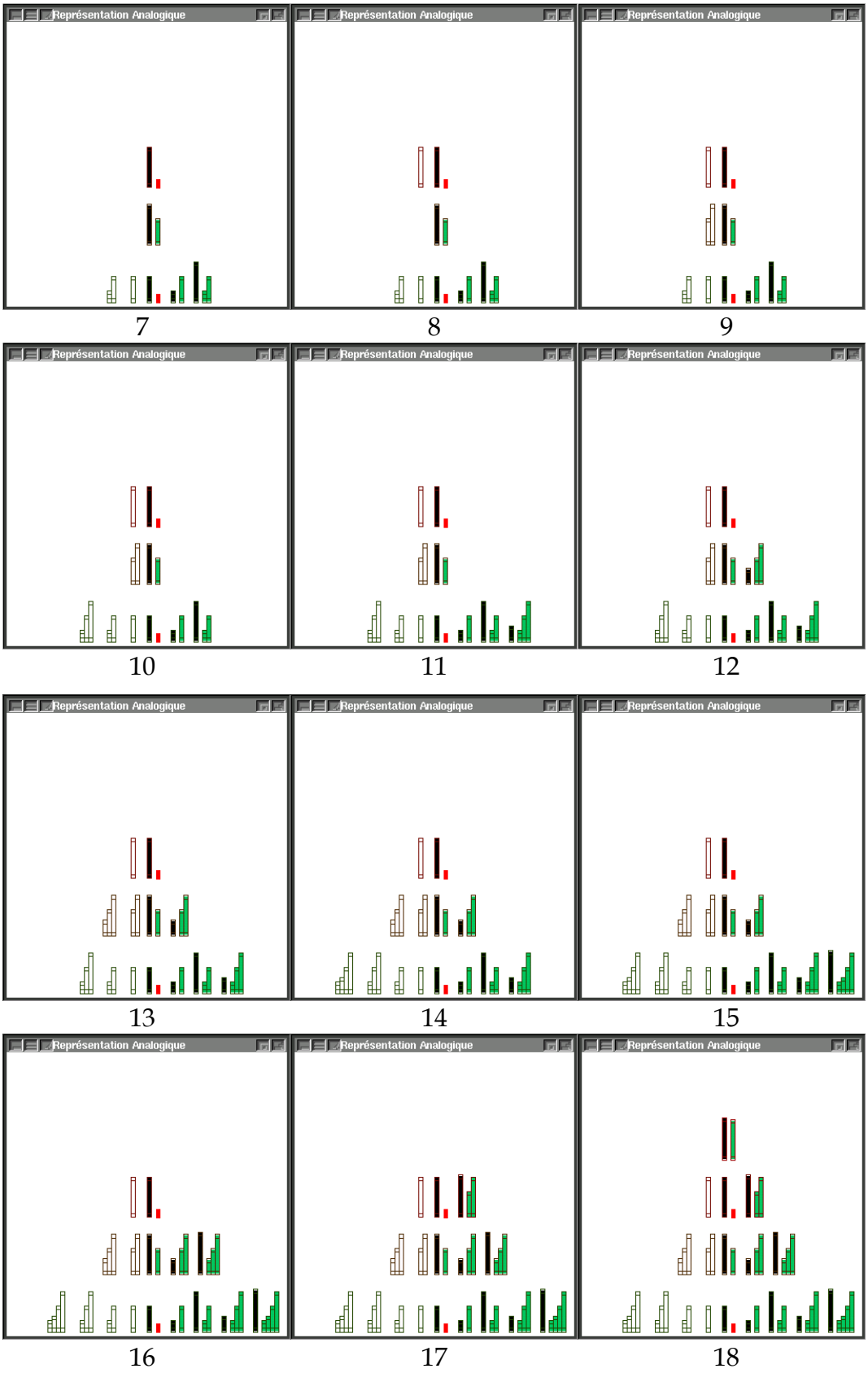


Figure 3.27 Film du tri par insertion (suite)

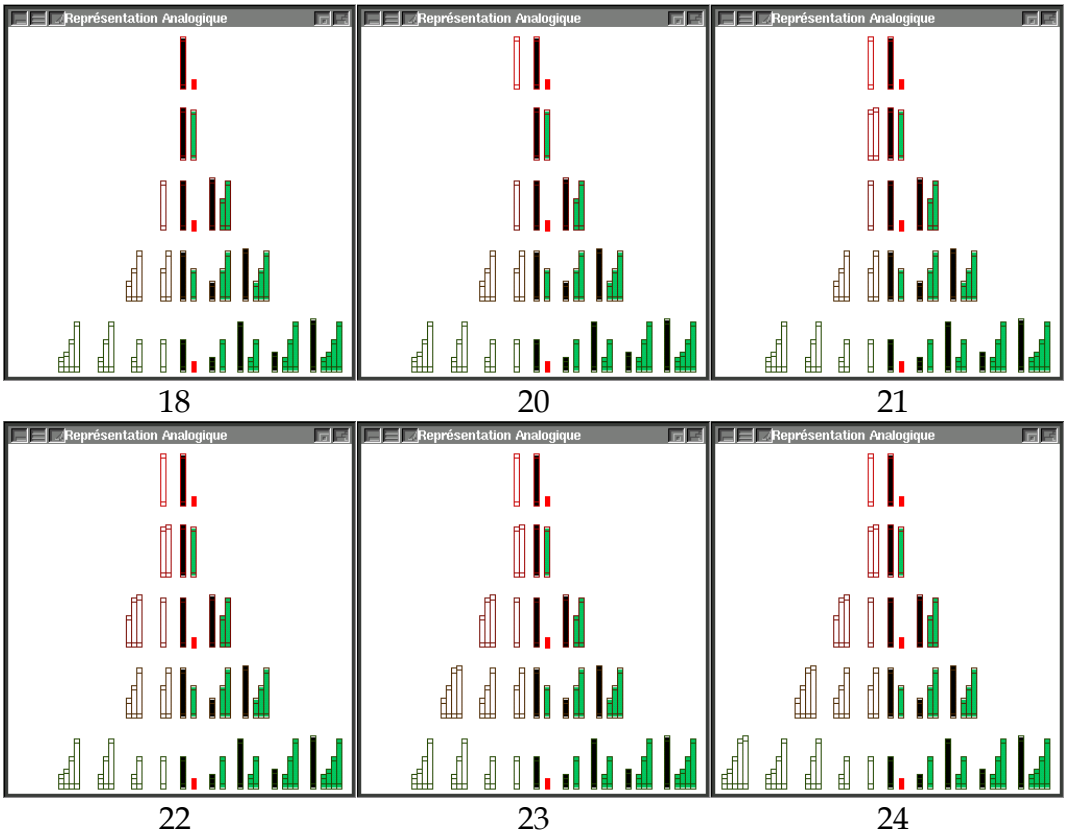


Figure 3.27 Film du tri par insertion (fin)

Il est important de noter ici que notre visualisation s'attache à illustrer le mécanisme en jeu dans l'insertion d'un élément *à la bonne place* mais, comme nous allons le montrer, elle permet également de visualiser la totalité de l'algorithme du tri par insertion. Examinons maintenant le film de l'animation construit par notre système. La fonction de l'implémentation à laquelle est attachée cette représentation est la fonction *insert*.

La première remarque visuelle est la présence, dans les données en entrée du programme, d'éléments de couleurs différentes. Ainsi, outre les éléments noirs déjà présents dans les exemples de visualisation des algorithmes de tri rapide et de tri par fusion, des éléments rouges et des éléments verts apparaissent dans la partie droite (correspondant à l'entrée) de l'animation.

Trois couples d'images nous donnent des indications par rapport à la signification des éléments de couleur rouge : ce sont les couples d'images 1 – 2, 7 – 8 et 19 – 20. Dans le couple 1 – 2, l'élément rouge, présent dans la partie entrée, disparaît dans la construction du résultat : il n'est pas pris en compte. On en déduit alors que cette couleur désigne des éléments neutres, sans valeur. Le couple 7 – 8 arrive après deux niveaux de récursion, les éléments verts à droite de chaque niveau ont successivement disparu et à leur place est apparu un élément rouge n'existant pas auparavant. Cette couleur semble ainsi désigner un élément présent une fois que les autres ont disparu, indiquant alors un espace vide. Cette dernière remarque s'applique également au couple 19 – 20, apparaissant, quant à lui après cinq niveaux de récursion.

Les éléments de couleur verte, comme ceux de couleur rouge, ne sont pas toujours présents en entrée du programme, contrairement aux éléments de couleur noire. De plus, dans toutes les images de l'animation, les éléments de couleur noire et rouge sont toujours isolés alors que les éléments de couleur verte peuvent constituer des groupes, visiblement toujours ordonnés. On peut aussi remarquer que les éléments verts qui apparaissent au premier niveau de récursion sont la copie conforme des éléments de sortie du même niveau à l'étape précédant leur apparition (cf. couples d'images 2 – 3, 4 – 5, 10 – 11 et 14 – 15).

Les éléments de couleur noire, toujours présents et isolés en entrée comme nous l'avons déjà remarqué, se retrouvent également toujours présents en sortie, dans le résultat. Ceci montre que ces éléments sont destinés à être ajoutés, dans la construction du résultat, à l'élément de couleur différente qui les accompagne. Par conséquent, le programme visualisé semble travailler avec deux données : une, de couleur noire, désignant un élément à placer dans l'autre, de couleur rouge ou verte.

Le placement des éléments de couleur noire dans ceux de couleur rouge ou verte semble alors suivre la démarche suivante :

- Si l'élément de droite est rouge, le résultat consiste en l'élément noir (à placer).
- Si l'élément de droite est vert, le programme progresse par appels récursifs successifs jusqu'au moment où le premier élément vert est plus grand que l'élément à placer, le résultat apparaissant alors à l'étape suivante (cf. couples d'images 3 – 4 et 12 – 13). Si cela n'arrive pas (l'élément noir est plus grand que tous les éléments verts), un élément rouge apparaît alors, ce que nous avons déjà remarqué.

On peut déduire des observations précédentes que le programme fonctionne comme suit :

1) Le programme trie une série d'éléments qui apparaissent successivement de couleur noire au premier niveau de récursion,
2) Les éléments sont placés, les uns après les autres, dans le résultat du tri des précédents,
3) Le placement des éléments s'effectue par la recherche de la *bonne place* dans les éléments déjà triés, par progression successive avec comparaison entre l'élément à placer et le premier élément de la liste où le placer.

Les deux premières étapes déduites de l'observation concernent des appels successifs au programme visualisé. Il apparaît ainsi que ce programme est une partie d'un autre programme qui l'utilise pour placer des éléments dans une liste de nombres. De fait, cet autre programme, que nous pouvons désigner comme méta-programme, fait appel au programme visualisé en conservant la valeur qu'il retourne puisqu'elle est présente à l'appel

suivant. On peut alors déduire que la liste triée, à l'extrême gauche du premier niveau de l'image 24, est composée de tous les éléments noirs de la droite de ce même niveau. Sans autre information sur ce méta-programme, trois conjectures sont alors possibles :

1) il reçoit les données d'un autre programme et construit une liste triée au fur et à mesure en partant d'une liste vide[100],
2) il parcourt une liste de nombres et construit à chaque étape une liste d'éléments triée (s'il fonctionne avec deux variables),
3) il parcourt la liste de nombres à trier jusqu'à la fin puis place successivement les éléments (du dernier au premier) à la bonne place (s'il fonctionne par construction en retour d'appel récursif comme les algorithmes déjà présentés).

En conclusion, les observations visuelles de cette animation nous ont permis de déduire le fonctionnement du programme *insert* présenté dans la figure 3.26 mais aussi de conjecturer le fonctionnement du programme *tri_insert* présenté dans la même figure. En effet, la première conjecture, si la visualisation est replacée dans le contexte de l'animation d'algorithmes de tri, perd alors de sa substance.

II.3.4.6 Conclusion

Notre animation analogique d'algorithmes de tris de listes de nombres, tout en se basant sur des éléments graphiques simples et communément utilisés pour ce type de visualisation, permet d'en déduire le fonctionnement par simple observation visuelle. La différence principale entre notre représentation analogique et celles que l'on trouve habituellement dans la littérature est la cohabitation des opérations effectuées sur les données avec la progression dans la résolution. De plus, il apparaît clairement que la persistance visuelle de ces informations est l'élément déterminant dans l'étude visuelle des animations. Dans notre critique par rapport aux autres systèmes de visualisation d'algorithmes présentée dans le chapitre IV de ce mémoire, nous insistons sur le fait que les visualisations proposées ne permettent pas de déduire toutes les étapes présentes dans les algorithmes de tri.

Notre visualisation, même si elle n'a pas été construite pour un algorithme précis, peut en fait s'appliquer à tout programme de manipulation de listes de nombres. Elle apporte l'élément manquant à une analyse visuelle complète du fonctionnement des algorithmes : *un historique animé des différentes étapes aboutissant à la construction du résultat*. De fait, ces éléments, appartiennent plus au fonctionnement des programmes qu'à la description habituelle des algorithmes. Ils peuvent être comparés à une trace graphique, et

[100] Peut-être en utilisant des *socket* ou tout autre mécanisme de communication inter-processus, ou bien en conservant la liste des données triées dans une variable globale et en y plaçant des données…

permettent de capturer la partie essentielle des algorithmes sous-jacents aux programmes représentés. C'est ce point qui illustre la partie analogique de la représentation : elle relie effectivement les différents points de l'exécution d'un programme de manipulation de listes de nombres à des objets graphiques, des choix de couleur, des animations et des dispositions spatiales dont la combinaison permet de déduire un comportement sous-jacent au programme.

Nous allons maintenant présenter notre système Zeugma, basé sur notre méthode de construction des représentations analogiques de programmes, avec lequel ont été construites toutes les représentations analogiques présentées dans ce chapitre.

Chapitre III

« […] le plus important de beaucoup, c'est de savoir faire des métaphores ; car cela seul ne peut être repris d'un autre, et c'est le signe d'une nature bien douée. Bien faire des métaphores, c'est voir le semblable. »

Aristote, Poétique
[Aristote, p. 117]

III Zeugma

Résumé

Les contributions principales de Zeugma sont :

- La création par l'utilisateur de liens analogiques entre des programmes et des représentations sous forme d'images programmables.
- L'indépendance syntactique des liens analogiques de tout contexte programmatoire, c'est-à-dire indépendant de techniques particulières de programmation et indépendant des structures langagières utilisées. Ces liens sont également indépendants du domaine d'application des programmes et des représentations graphiques choisies. Bien entendu, une dépendance de ces divers contextes peut être introduite ultérieurement à un niveau sémantique librement choisi par l'utilisateur.
- L'observation de ces liens, tant au niveau statique de la représentation que dans leurs aspects évolutifs liés à l'exécution des programmes.

Ce chapitre présente notre système Zeugma. Dans un premier temps nous allons décrire les principes sous-jacents aux différentes parties le composant et, dans un deuxième temps, nous décrirons en détail son utilisation. Les analogies présentées dans le chapitre précédent ont été construites en utilisant ce système et son code source complet se trouve en annexe de ce mémoire.

III.1 Eléments constitutifs du système Zeugma

III.1.1 Introduction

III.1.1.1 Problématiques

Le but principal de notre système Zeugma est d'aider à la création, l'exploration et l'expérimentation de représentations analogiques de programmes Lisp. Zeugma est composé d'outils permettant la création de liens analogiques[101] entre des objets textuels (les programmes) et des objets gra-

[101] Nous rappelons que les *liens analogiques* entre deux objets mettent en correspondance un ensemble de relations définissant une structure de relations existante dans ~~l'~~un ~~des~~ objet~~s~~ représenté~~s~~ avec un autre ensemble de relations couvrant la

phiques (les images). Ces outils prennent en compte la triple nature des programmes : celle de textes structurés répondant à une syntaxe et une grammaire, le fait que ces textes décrivent des processus, et, enfin, le fait que ces processus puissent être activés.

Nos représentations graphiques, imagées et animées, visent à rendre intuitivement compréhensibles aussi bien ces aspects disjoints que leurs interdépendances et interactions : comment la description d'un processus change-t-elle en fonction d'une modification textuelle et comment le processus lui-même est-il modifié. C'est à de telles questions que nous essayons en premier de répondre. Mais – ne le perdons pas de vue – nous essayons également de répondre aux questions inverses : quels sont les changements engendrés par une modification du comportement du programme dans le texte de la description du processus sous-jacent ?

De plus, nous devons apporter des éléments de réponse aux questions difficiles que sont :

- la nécessité de descriptions textuelles, analogiques, graphiques, etc. ?
- leurs apports ?
- leurs limitations ?
- ce qu'elles montrent, cachent, rendent inaccessibles ?
- ce qu'elles améliorent ?

Techniquement, Zeugma répond à un double défi :

1) L'implémentation de liens utiles et utilisables entre des objets de nature différente (par exemple : textes et images, descriptions et processus),
2) L'établissement de descriptions comportementales de ces objets de natures différentes, de manière à pouvoir lire (observer, parcourir, etc.) les deux objets en parallèle, cela avec des critères de lectures différents, tout en maintenant une relation entre les deux.

Pour répondre à ces défis, Zeugma permet la définition de liens analogiques indépendants d'un contexte applicatif particulier et d'un programme spécifique en se basant sur des *aspects des programmes* : des résultats d'analyse de programmes Lisp basés sur les différents flux de contrôle ou de données, l'utilisation de telle ou telle expression ou encore des statistiques effectuées sur la composition des programmes. Ces différentes caractéristiques définissent des points de vue sur les différentes natures d'un programme, permettant alors d'élaborer des représentations spécifiques à chacune et de les combiner dans une même image, dans un même scénario. L'ensemble « programme plus résultat des analyses » est mis en relation avec des représentations graphiques dont les objets, la structure et les animations sont librement

même structure dans un ~~l'~~autre objet.

définissables par l'utilisateur. De plus, notre système offre des outils d'observation et d'aide à la lecture des analogies dans les diverses dimensions abordables (représentation de textes structurés, de descriptions d'un processus ou du comportement des processus).

III.1.1.2 Choix techniques

Nous allons maintenant présenter les différents choix *techniques* à partir desquels fut construit notre système Zeugma. Ces choix sont :

- le choix du langage de programmation étudié (Lisp),
- le choix de l'interprète de ce langage (Xbvl),
- le choix de la librairie graphique utilisée pour construire les représentations (OpenGL).

Suite à l'explication de ces choix, nous présenterons le choix méthodologique qui est à la base de la construction des représentations analogiques de programmes dans Zeugma : les *Objets de Relations Structurels*.

III.1.1.2.1 Choix de Lisp

Les caractéristiques suivantes du langage Lisp nous semblent particulièrement adaptées à l'implémentation de notre méthode de création de représentations analogiques. En effet il possède :

- Une équivalence entre la représentation interne et externe des programmes et les données qu'ils manipulent. Ce point nous a permis de construire un système autorisant autant la construction de visualisation de la composition des programmes (comme les cités), que de leur comportement (les araignées) ou la visualisation d'algorithmes (les tours de Hanoi ou le tri de listes). En effet, Zeugma n'effectue pas de distinction entre programmes et données dans la construction des représentations. La seule partie du système spécifiquement liée aux programmes Lisp est l'ensemble des mécanismes permettant l'observation de leur exécution.
- La structure des programmes (les listes) permettent le traitement du code source des programmes comme structure prédicative, conduisant à définir des critères de parcours communs entre les différents niveaux d'abstraction, qu'ils soient littéraux (le code source), structurels (les flots de contrôle et de données), ou abstraits (classification des fonctions, …). En effet, les différents critères d'activation des ORS utilisent les notions de profondeur (CAR), de largeur (CDR) et d'élément (ATOM) issues de la représentation interne des abstractions sous forme de listes.

Ces deux caractéristiques particulières au langage Lisp pourraient se retrouver directement dans d'autres langages (par exemple Prolog en considé-

rant les listes manipulées dans ce langage sous leur forme prédicative) ou même se construire dans des langages donnant accès de manière non contraignante (ne demandant pas la modification des programmes) à la composition ou au comportement des programmes comme SmallTalk.

Nous avons choisi de ne pas utiliser le langage Prolog comme base de notre étude sur la visualisation de programmes pour deux raisons : le caractère particulier du paradigme sous-jacent et, bien que Prolog soit un excellent outil d'enseignement de la programmation, le langage Lisp paraît plus adapté à la construction de programmes d'importance, d'où son statu de langage adapté à la construction de prototypes [Wertz83] ainsi qu'à la construction d'abstractions sur les programmes[102].

Le problème avec le langage Smalltalk est d'une nature différente. Ce langage définit en effet un paradigme de programmation basé sur les notions de classes et d'objets (les instances de classes). Toutefois, la composition même des classes utilise des éléments qui n'appartiennent pas à ce paradigme (les méthodes) ni à l'exécution des programmes (les messages). Ainsi, pour pouvoir appliquer notre méthode de construction de représentations analogiques de programmes, il nous aurait fallu définir trois types d'éléments cognitifs : un pour les classes et les instances de classes, un second pour les méthodes et un troisième pour les messages.

III.1.1.2.2 Choix de Xbvl

Figure 1.1 Interface de Xbvl

La seule partie de notre système Zeugma intimement liée au langage Lisp nécessitant une interaction avec l'interprète sous-jacent est celle qui est liée à l'observation de l'exécution des programmes. Notre choix de présenter un système permettant de traiter de manière efficace ce point nous a conduit à choisir le système Xbvl comme base de construction de notre environnement de programmation. En effet, Xbvl [Sendoya92], issu des systèmes

[102] Les travaux de Fançoise Balmas, que nous avons évoqués plus haut, sont construits à partir d'une étude du langage Lisp.

VLISP [Greussay76, 77, 82] et bVLISP [Wertz83] répond aux exigences posées par notre méthode :

- Il est aujourd'hui portable sur la plupart des systèmes Unix disponibles et une version destinée au système Windows95 est actuellement en voie de réalisation. Ce point nous a permis de développer Zeugma et de l'évaluer en laboratoire ainsi que dans un contexte pédagogique sur les ordinateurs destinés aux étudiants de premier et second cycle.
- Il propose un mécanisme d'ajout d'activité supplémentaire à l'évaluation des différents objets Lisp. Ce mécanisme ayant comme propriété principale de n'affecter en rien le déroulement normal des programmes nous a permis de mettre en place des mécanismes d'observation du comportement des programmes de manière automatique, transparente pour l'utilisateur du système[103].
- Il propose une interface avec X Windows qui nous a permis de construire l'interface utilisateur de notre système, interface dont nous détaillerons l'implémentation dans la section III.4.2 de ce chapitre.
- Il inclut un interprète Prolog qui nous permet actuellement d'étendre notre méthode aux programmes écrits dans ce langage.
- Son implémentation répond totalement au souci d'ouverture évoqué plus haut et nous a permis de réaliser une interface entre Xbvl et la librairie graphique Open GL.
- Il propose un interprète Lisp optimisant *à la volée* les récursions terminales procurant une optimisation transparente des programmes. Il permet l'écriture de programmes automodifiants ce qui ne restreint en rien l'écriture de programmes dans le langage Lisp.

De plus, Xbvl est issu de travaux de recherche sur les environnements de programmation[104], la compréhension et la correction automatique de programmes[105] ainsi que sur la construction d'interfaces entre le langage Lisp et des fonctionnalités appartenant au système Unix telles que la communication

[103] Ceci permet, par exemple, de capter de manière sélective les types d'instructions évaluées sans avoir à écrire un méta-evaluateur comme celui développé afin de construire des visualisations du comportement des programmes par Hideki Koike et Manabu Aida [Koike95b].

[104] Il intègre, depuis bVLISP, des outils de documentation, d'édition d'annotation et de vérification des programmes Lisp.

[105] PHENARETE, système de correction et d'amélioration automatique de programmes Lisp [Wertz76, 78, 79], peut être automatiquement mis en œuvre par l'utilisateur de Xbvl en cas d'erreur pendant l'exécution d'un programme.

inter-processus[106]. Il s'inscrit donc entièrement dans notre démarche d'élaboration d'un environnement de programmation intégrant des visualisations analogiques de programmes.

III.1.1.2.3 Choix de Open Gl

Notre méthode de construction de représentations analogiques de programmes fait principalement correspondre des structures appartenant à des abstractions de programmes et à des objets graphiques. Ainsi, nous avons porté notre choix sur la librairie graphique 3D Open GL [] comme interface de construction des représentations graphiques car elle offre les caractéristiques suivantes :

- Le rendu des objets graphiques est calculé à partir d'une série de transformations (projection, position et échelle, texture) gérées par des matrices placées sur des piles indépendantes. La gestion de ces piles donne à l'utilisateur de cette librairie la possibilité de construire des représentations dont les objets graphiques partagent ou influencent les propriétés d'autres objets. Comme nous l'avons illustré dans le second chapitre de cette thèse avec nos exemples de représentations analogiques, son utilisation permet de positionner des objets selon des séries de transformations successives déterminant ce que nous nommons la *structure* de la représentation graphique.
- L'implémentation de l'interface entre Xbvl et OpenGL est basée sur l'utilisation de *listes graphiques* regroupant des séries d'opérations et gérées par la librairie elle-même. La modification dynamique de ces listes nous a permis de construire les animations des représentations (en plaçant, par exemple, des translations dans des listes dont la modification n'affectera que l'élément graphique animé) et nous a également fourni le moyen d'effectuer des sauvegardes de parties d'images, facilité que nous utilisons dans la construction de l'historique d'une animation.
- Il existe une version *freeware*, MESAGL, compatible avec la version commerciale de cette librairie et disponible sur de nombreuses plates-formes et systèmes d'exploitation. Ceci permet à Xbvl de proposer d'importantes capacités de modelage de graphiques 3D tout en conservant sa dimension d'ouverture, d'évolutivité et de portabilité.

Ainsi, nous avons choisi d'implémenter une interface entre Xbvl et OpenGL, interface détaillée dans l'annexe 3 de ce mémoire, et de l'utiliser

[106] Il possède aujourd'hui une interface avec les *sockets* permettant de construire des applications Lisp distribuées ou la communication entre des programmes écrits dans d'autres langages de programmation et Xbvl. Cette fonctionnalité nous permettra, dans un développement futur de Zeugma, d'intégrer l'étude de multiples langages de programmations.

comme outil de construction des représentations analogiques.

III.1.2 Eléments constitutifs de Zeugma

Notre système Zeugma permet la création de liens analogiques entre des programmes Lisp et des représentations graphiques. Les liens analogiques, reliant les programmes et leurs représentations, se basent sur la mise en relation d'aspects des programmes et des représentations graphiques.

Nous allons détailler ici les notions liens entre programmes et représentations, d'aspects des programmes et représentations graphiques, tels qu'ils sont utilisés dans ce système, ainsi que les objets conçus pour mettre en relation ces deux types d'objets.

III.1.2.1 Les liens entre programmes Lisp et représentations analogiques

La définition des relations entre les programmes Lisp et leurs représentations graphiques s'effectue par l'établissement de *schèmes de génération de représentations analogiques*. Ces schèmes sont constitués d'*Objets de Relation Structurelle* (ORS) décrivant les relations entre des composantes d'objets de nature différente. Par exemple, s'il s'agit de mettre en relation un programme Lisp et une représentation graphique sous forme d'une ville abstraite, les ORS établissent :

- La relation entre les voiries de la ville et une organisation des différentes parties du programme. Cette organisation pouvant être construite à partir des flux de contrôle et de données ou, comme dans notre exemple précédent, à partir d'une répartition des fonctions Lisp suivant le type d'instruction utilisé.
- La relation entre les quartiers, les maisons individuelles et les fonctions du programme.
- La relation entre les différentes pièces et les différentes instructions composant les fonctions du programme.
- La relation entre les objets graphiques présents dans les jardins et les données manipulées par les fonctions.
- La relation entre un objet en mouvement (personnage, voiture, etc.) et la progression dans l'exécution du programme.

Ce sont donc les ORS qui ont à charge à la fois le contrôle de la construction de la représentation analogique, la description des relations particulières et celle des parcours analogiques dans les deux objets.

La définition de tels objets donne au système Zeugma une indépendance par rapport à un contexte applicatif ou programmatoire. En effet, le choix d'objets abstraits regroupant les propriétés des liens entre les programmes et leurs représentations entraîne une abstraction par rapport à un

programme particulier, ce qui conduit alors à des représentations analogiques propres à chaque programme. L'ensemble des ORS mis en œuvre dans la construction d'une représentation définit alors le schème de génération de la représentation analogique.

La figure 1.2 présente un résumé de la méthode utilisée dans notre système pour construire des représentations analogiques de programmes. Elle est ici considérée comme l'ensemble des choix d'aspects des programmes et de représentation graphique liée entre eux par des schèmes de génération de représentation composés d'Objets de Relation Structurelle.

III.1.2.2 Aspects des programmes

L'élaboration des aspects des programmes a pour fonction de permettre une plus grande lisibilité dans la représentation, pour une lecture et une observation des programmes à partir d'analogies. Si l'utilisateur est confronté à un problème programmatoire du style *goal – clobbering* [Sussman72], il est clair que la recherche d'une solution se situe plutôt sur un niveau d'interaction entre modules (fonctions) qu'à l'intérieur d'une fonction isolée. Les aspects choisis afin de construire une analogie appliquée à ce type de problème doivent conduire à faire abstraction du comportement individuel des fonctions. Ils doivent, également, renforcer les vues possibles sur les échanges et interactions globales entre les différentes parties du programme. Nous définissons ainsi les aspects des programmes comme étant :

1) Des *parcours* des différents niveaux structurels présents dans les programmes. Ces niveaux structurels pouvant être décrits comme les flots de contrôle et de données pour les relations entre parties du programme, les différents types de composition d'instructions (branchements conditionnels, répétitions, …), ou encore les différents types d'instructions présents. Ces niveaux structurels sont, en

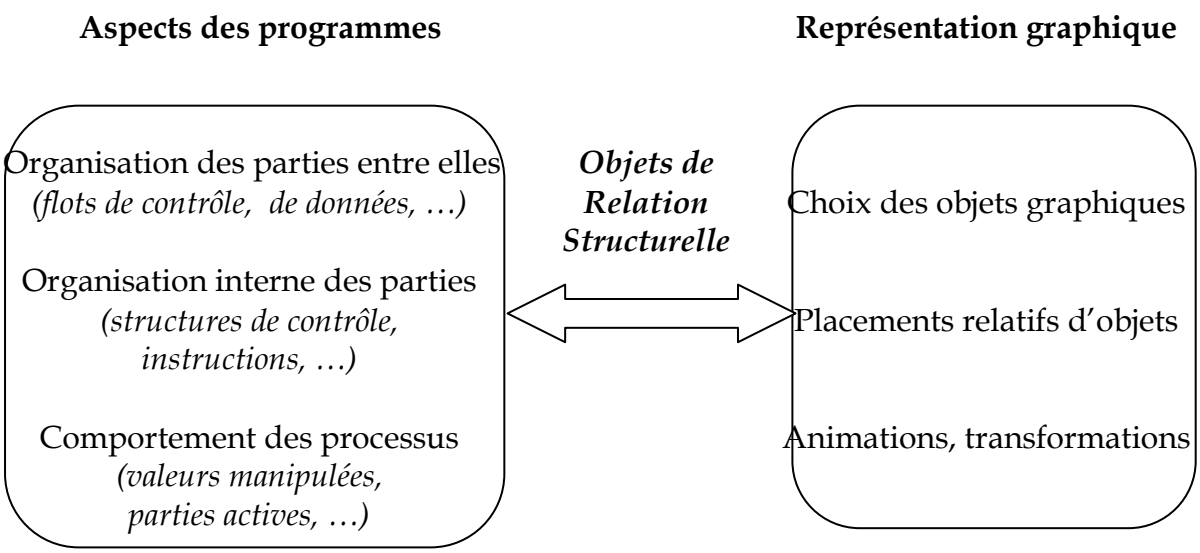


Figure 1.2 Schéma général des représentations analogiques de programmes

eux-mêmes, le thème de multiples travaux de recherche que ce soit à un niveau organisationnel des parties entre elles (comme les travaux sur les plans ou clichés [Rich81]), ou au niveau de la composition interne des parties (signatures [Balmas95b]). La notion de parcours s'applique alors au processus de génération de la représentation analogique qui met en relation le calcul d'une structure présente dans un programme particulier à un schème de génération de représentations analogiques.

Bien entendu, les niveaux structurels sont un ensemble *ouvert*, appelant de nouvelles définitions suivant le domaine étudié, bref, configurable par l'utilisateur du système.

2) Des discriminations entre les différentes parties actives lors de l'exécution du processus décrit par le programme. Ceci afin de permettre la liaison entre le rôle des différents éléments du programme et des animations ou des transformations de la représentation graphique.

Ce deuxième point, lié à la considération des programmes comme processus actifs est, lui, figé dans Zeugma. Les aspects qui le définissent sont les différents aspects observables dans l'exécution d'un programme : la mise en œuvre des fonctions (à l'accès au point d'entrée et/ou à sa sortie), la mise en œuvre des expressions (au début et/ou à la fin de leur évaluation), l'accès au variables (en distinguant les effets de bords des simples consultations) et aux valeurs qu'elles contiennent.

Nous allons maintenant détailler ces deux types d'aspects des programmes en prenant comme base les caractéristiques définies actuellement dans Zeugma.

III.1.2.2.1 Les niveaux structurels comme aspects des programmes

Les premières structures prises en compte en tant qu'aspects, et présentes dans tout programme, sont les différents flots de contrôle et de données, ainsi que la structure purement syntaxique de chaque unité composant le programme (fonction, module, expression, etc.). Rappelons que nos représentations analogiques doivent capter à la fois les singularités de chaque expression, de chaque fonction, voire de chaque module, qu'elles doivent permettre d'exprimer les interdépendances (sous forme de flux par exemple) entre entités et groupes de fonctions (et programmes). Notre choix porte sur la structure arborescente de la *liste programme* (voire *liste instruction* ou *liste fonction*) déterminant la représentation des objets individuels ; d'autres caractéristiques seront à la base de la représentation de leurs interdépendances.

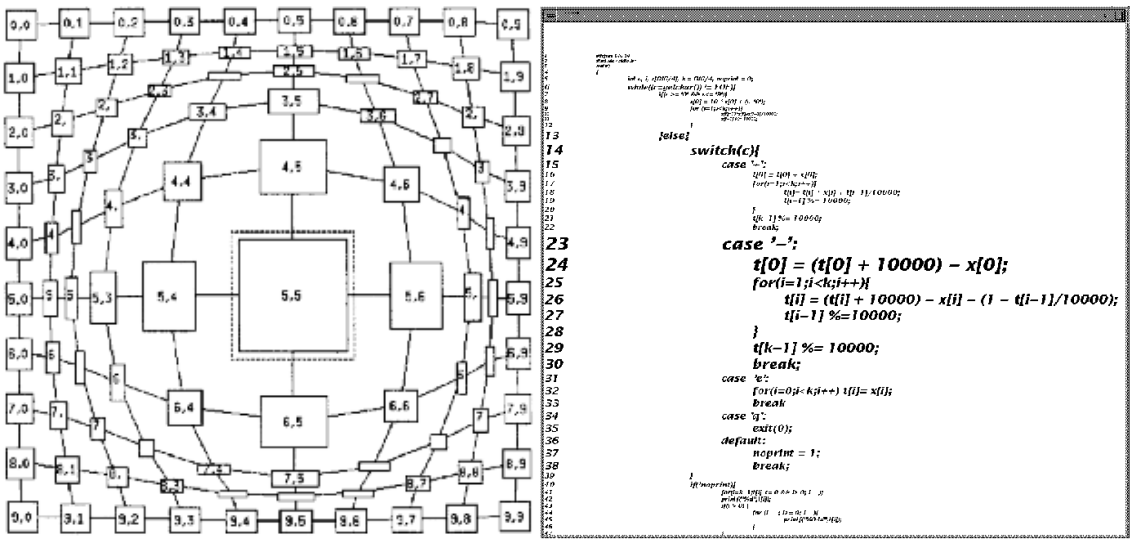


Figure 1.3 Zoom sur un graphe selon la méthode du Fisheye [Sakar93] et selon la méthode Fractale [Koike95a]

Pour revenir sur notre exemple de représentation analogique sous forme de cités, la forme de chacun des lieux à l'intérieur d'une maison est déterminée par la structure de l'expression correspondante. Les voiries représentent les communications possibles, les interactions entre les maisons, groupes de maisons, etc.

C'est cette distinction dans le choix qui détermine la construction de la représentation : niveau syntactique pour les unités et résultats d'analyses plus poussées pour l'interaction. Ceci nous permet de changer aisément de point de vue, par exemple pour cacher les détails de pièces d'une maison quand le point de vue est distant (par exemple au niveau d'un quartier), voire global. Mais, pour un examen ou un suivi local, le point de vue doit changer, de manière à rendre les détails à nouveau visibles.

Ceci pose également le problème de la *distorsion* à l'intérieur de représentations qui devraient à la fois capter un comportement global et quelques comportements très localisés. Des solutions comme la représentation en Fisheye [Sakar93] ou fractale [Koike95a] (cf. Figure 1.3) sont des propositions originales allant dans ce sens. Nous reviendrons sur ce problème et sur l'approche offerte par notre système dans la section 1.4.1.

Détaillons maintenant les méthodes utilisées actuellement dans Zeugma pour obtenir ces différentes structures de programmes.

a) Les flots de contrôle

Les flots de contrôles sont obtenus suivant deux méthodes de calcul choisies par l'utilisateur :

1. Un flot purement syntaxique, calculé à l'instant de la lecture du programme par Xbvl[107], se basant sur la détection d'appels à des

[107] Ces aspects sont issus d'outils appartenant à l'environnement de programmation

fonctions (par défaut ne sont pris en compte que les fonctions utilisateur),

2. Un flot décrivant une exécution du programme, Xbvl gardant les informations de transfert de contrôle d'une fonction à une autre au moment de l'exécution.

Ainsi, la figure 1.4 présente les flots de contrôle issus des deux méthodes de calcul pour un même programme de parcours de graphe[108].

b) Les flots de données

Les flots de données sont calculés de manière syntaxique en vérifiant l'influence d'une variable appartenant à une fonction sur une autre variable de cette même fonction ou d'une autre fonction du programme, soit par le passage comme paramètre, soit par une opération à effet de bords (affectation ou modification). La structure résultante sera ainsi un graphe dont chaque nœud est composé d'un couple « variable – fonction » mettant en relation le nom d'une variable et le contexte de son utilisation. La figure 1.5 présente une partie des flots de données pour le même programme de parcours de graphe. Cette figure, dans laquelle est également présent l'affichage des informations sur l'animation de la représentation de l'analogie « des programmes comme des cités », présente l'influence de la variable *mem* définie dans la fonction *rechercher*. Cette variable influence la variable *mem* définie dans la fonction *rech* et cette dernière va influencer les différentes variables présentées dans l'arborescence. Ainsi, ce flot de données montre les influences directes comme indirectes (par passage de paramètres successifs) d'une donnée sur l'ensemble du programme. Nous détaillerons dans le paragraphe 2.1.5.2 (page 147) les différents attachements possibles entre une variable et l'animation d'une représentation analogique.

c) Structures internes des programmes

La structure des parties de programmes (ou fonctions) est issue de la structure même de la programmation Lisp qui représente les expressions sous forme de liste, permettant ainsi de considérer le code source comme une structure arborescente. Cette structure arborescente, équivalente dans sa représentation interne et sous forme d'arbre graphique, peut également être utilisée pour modifier la fonction. La figure 1.6 présente ainsi une fonction Lisp sous sa forme textuelle et dans une représentation structurelle arborescente. En sus de ces structures, le système permet l'utilisation de tests sur les

Xbvl [Wertz83, p. 54]. En effet, Xbvl procède automatiquement, lors de la définition ou de l'utilisation d'une fonction Lisp, à l'extraction du type de la fonction, de la liste des variables locales et globales utilisées ou modifiées ainsi que de la liste des fonctions utilisateur appelées ou appelant cette fonction.

[108] Ce programme est celui à partir duquel a été construite la cité *Graphe* de la représentation analogique de programmes comme des villes présentée dans la partie précédente de ce mémoire.

particularités des expressions. Ces particularités entraînent la possibilité de placer des conditions sur les instructions utilisées (test sur les instructions elles-mêmes ou sur des groupes d'instructions classés par rapport à leur fonctionnalité) ou d'utiliser le résultat d'énumération par rapport à l'utilisation de telle ou telle classe d'instructions.

L'utilisation de telles structures permet de répondre au problème du point de vue adopté pour la construction de la représentation. Par exemple, les représentations analogiques par des cités ou par des araignées sur une toile adressent des niveaux totalement différents dans leur complexité (très haut niveau de détail pour les cités, minimal pour les araignées) tout en étant applicables à un même programme. Par le choix de la ou des aspects de programmes pris en compte lors de la génération de la représentation analogique l'utilisateur détermine le niveau de détail de la représentation finale.

III.1.2.2.2 Définition d'aspects structurels dans Zeugma

Les aspects structurels sont considérés, dans notre système, comme les données décrivant le domaine de départ de construction des analogies ; le domaine cible étant les représentations graphiques. Afin de répondre au souci d'ouverture, leurs définitions peuvent être modifiées par l'utilisateur de notre système. Nous allons décrire ici la définition d'un de ces aspects : le flot de contrôle. Dans un second temps, nous montrerons, par des exemples, l'utilisation de ces aspects des programmes dans l'élaboration des schèmes de génération de représentations analogiques.

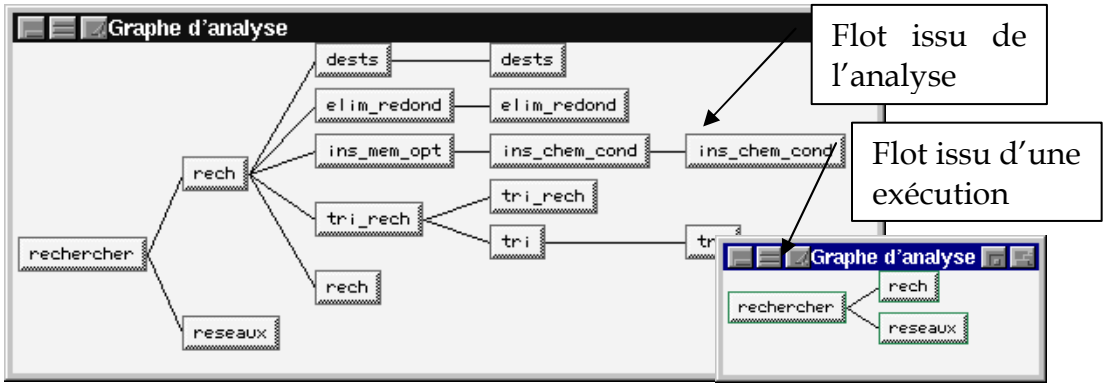


Figure 1.4 Deux flux de contrôle d'un programme

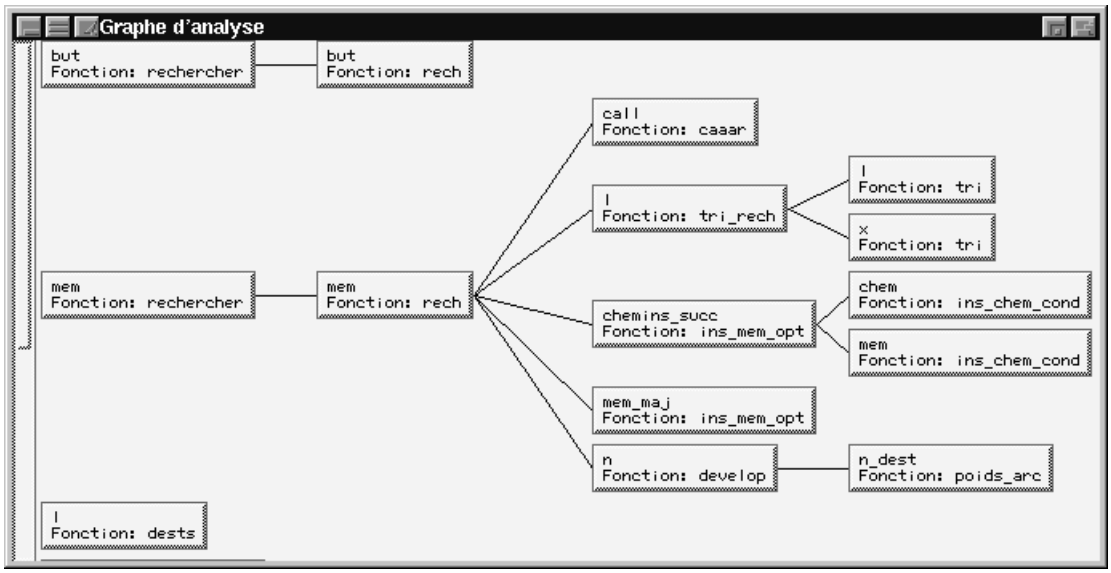


Figure 1.5 Flot de données d'un programme

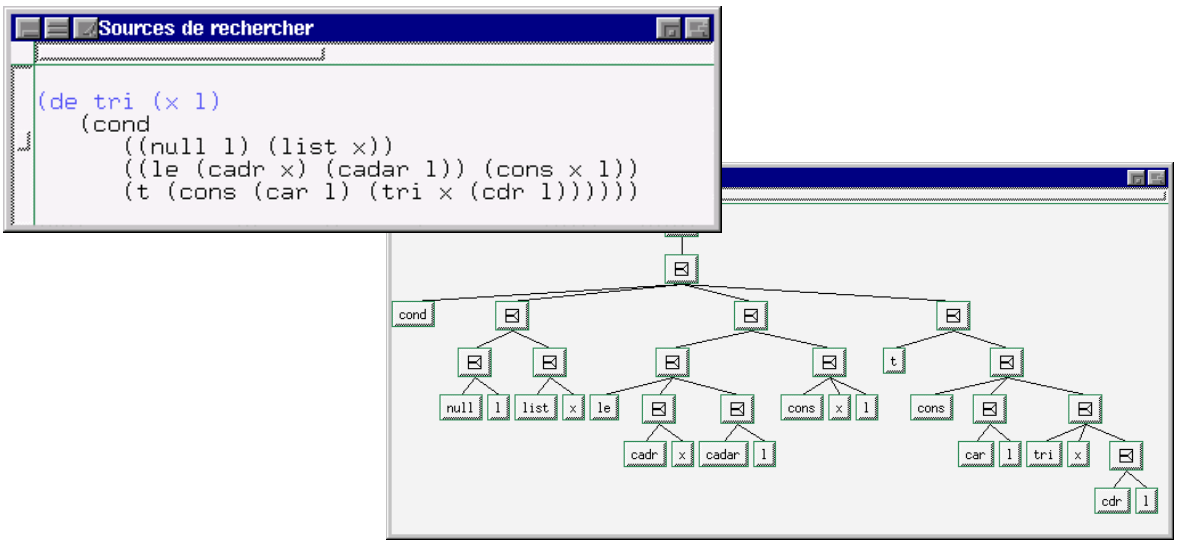


Figure 1.6 Deux représentations d'une fonction Lisp

La figure 1.7 présente la définition, dans la syntaxe de Zeugma, du flot

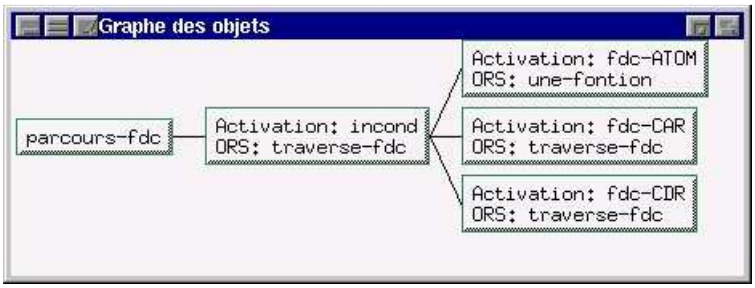
de contrôle comme aspect des programmes. Pour comprendre ce code source, notons tout d'abord qu'il débute par une déclaration de la spécificité du parcours d'un flot de contrôle, par rapport aux connaissances générales de Zeugma sur ce qu'est un aspect. Si nous savons que Zeugma est implémenté – en partie – comme un système orienté objet, *define-PDV Flot-de-contrôle* peut être considéré comme la définition d'une sous-classe de la classe PDV (racine des aspects de programmes). Cette sous-classe précise donc son nom et titre (qui seront utilisés par la suite dans les menus d'interface de cette classe), la (ou les) fonction(s) d'analyse permettant de construire à partir d'un programme son flot de contrôle, l'ensemble des méthodes *Test* définissant la manière de parcourir ce flot de contrôle. La suite de la figure 1.7 donne la définition de chacune de ces méthodes (toujours, bien entendu, écrites en Zeugma). Cette figure vise à donner une première idée sur l'écriture d'objets Zeugma et, afin de ne pas alourdir trop cet extrait, nous avons volontairement omis la définition de la fonction d'analyse de programmes *analyse-FDC* qui se trouve à l'annexe II.6 de ce mémoire.

La définition de tels aspects des programmes dans Zeugma permet leur utilisation comme *moyen* pour définir les parcours des programmes. Ces parcours ne seront donc pas basés sur un programme ou un groupe général de programmes mais selon un aspect des programmes applicable à différents programmes, sans nécessairement en avoir connaissance préalablement. Regardons alors quelques exemples d'utilisation des aspects *flot de contrôle*, *flot de données* et *composition interne des fonctions* qui peuvent respectivement mettre en évidence les échanges entres les fonctions (au niveau du contrôle et des données) et baser la construction d'une analogie sur l'étude de l'organisation des expressions dans les fonctions constitutives d'un programme.

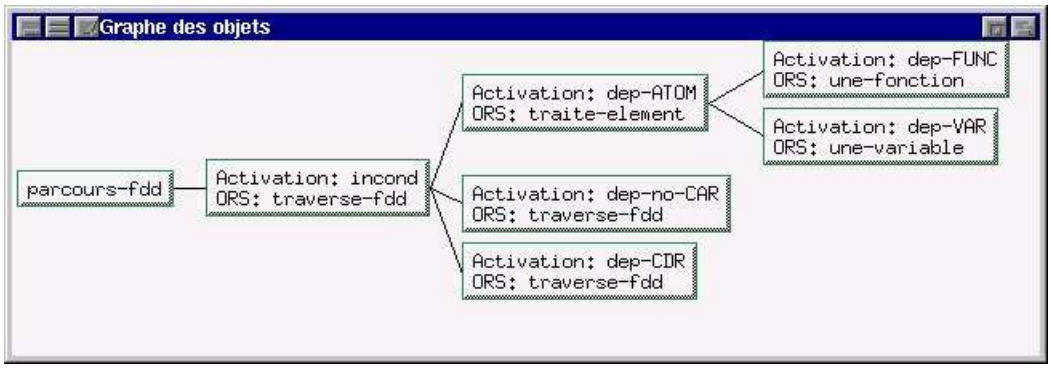
La figure 1.8 présente sous forme arborescente les schèmes générateurs de représentations analogiques. Ces schèmes sont considérés comme *source* car ils décrivent la manière de considérer les programmes, mais ils ne décrivent pas les opérations graphiques qui leurs sont liées. Chaque élément des arborescences présent dans ces figures est formé d'un couple « condition d'activation – ORS » indiquant, d'une part, dans quelle condition (liée au parcours d'un aspect des programmes) l'élément est activé et, d'autre part, l'ORS correspondant à cet élément.

```
; Définition du flot de contrôle comme aspect des programmes
; Ce flot est construit à partir des informations d'appel présentes dans l'i-val des fonctions, ainsi
; le flot d'un programme se trouve lié à sa structure syntaxique ou à une exécution particulière
;
; Les compteurs liés à ce parcours sont :
; fdc-ipr : profondeur dans la structure
; fdc-ilg : largeur dans la structure
; fdc-iilg : largeur dans le sous arbre courant
; fdc-nlig : nombre d'éléments du sous arbre courant
; Les tests liés à ce parcours sont :
; fdc-CAR : parcours en profondeur
; fdc-CDR ; parcours en largeur
; fdc-ATOM : traitement d'un élément

(define-PDV Flot-de-controle
  (nom FDC)  (titre "Flot de contrôle")          ; nom et titre pour les menus de l'interface
  (analyse (analyse-FDC . FDC)) ; fonction de l'analyse et champ dans lequel est placé le flot à traiter
  (compteurs fdc-ipr fdc-ilg fdc-iilg fdc-nilg) ; compteurs indiquant la progression dans la structure
  (tests fdc-CAR fdc-CDR fdc-ATOM))          ; tests liés au parcours de la structure

; Dans la définition des tests, flow désigne la structure parcourue et car-flow le premier élément
; de cette structure.

(define-Test fdc-CAR
  (nom fdc-CAR)  (titre "Flot de contrôle - Profondeur")
  (test-eval (listp car-flow))        ; test sur la structure parcourue
  (next-flow car-flow)                ; effet sur la structure parcourue
  (fdc-ipr (1+ fdc-ipr))  (fdc-iilg 1)  (fdc-nilg (length car-flow))) ; effet sur les compteurs

(define-Test fdc-CDR
  (nom fdc-CDR) (titre "Flot de contrôle - Largeur")
  (test-eval (cdr flow))
  (next-flow (cdr flow))
  (fdc-ilg (1+ fdc-ilg)) (fdc-iilg (1+ fdc-iilg)))

; le test fdc-ATOM vérifie, si Zeugma à été configuré pour ne pas afficher plusieurs fois le même
; élément (Flow-Uniq), que l'élément n'a pas déjà été rencontré lors du parcours de la structure.
; Pour cela  il utilise une variable locale (Fdc-Constructed), mise à jours à chaque fois que le test a
; été validé, qui va contenir les éléments rencontrés.
(define-Test fdc-ATOM
  (nom fdc-ATOM)  (titre "Flot de contrôle - Elément")
  (local-var Fdc-Constructed (progn (newl Fdc-Constructed car-flow) (GA-Set-Composition car-flow)))
  (test-eval (and (litatom car-flow) (not (and Flow-Uniq (member car-flow Fdc-Constructed)))))
  (next-flow car-flow)
  (fdc-iilg 1) (fdc-nilg 0))
```

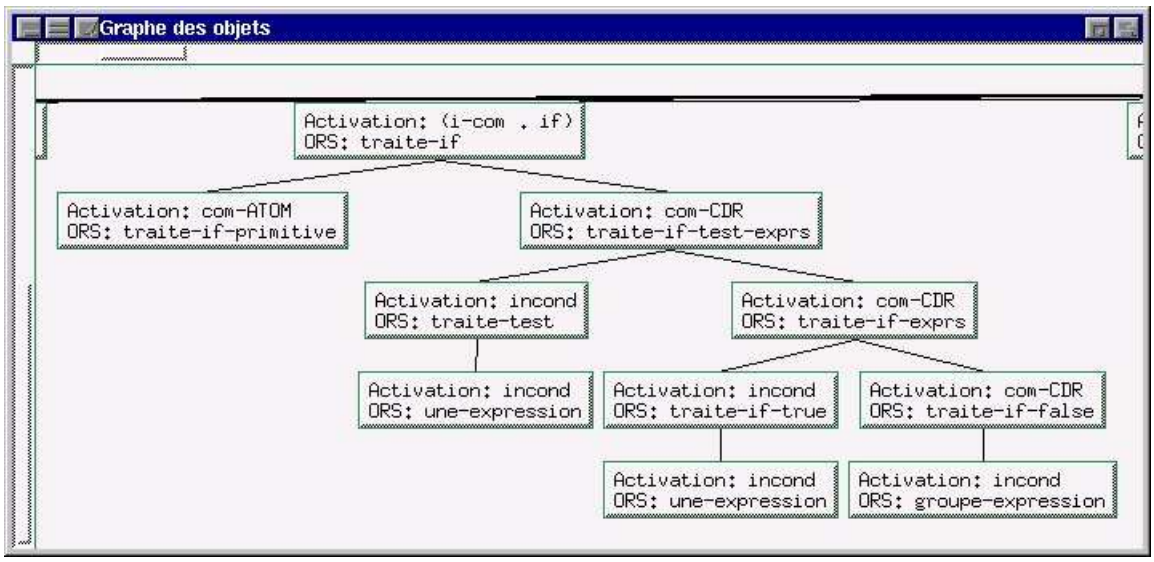
Figure 1.7 Définition du flot de contrôle comme aspect des programmes.



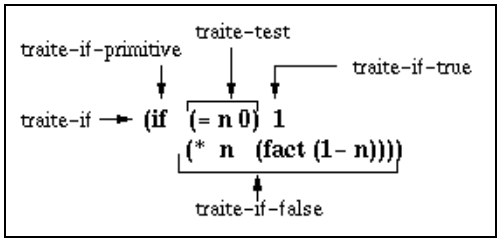
(a) Schème parcourant le flot de contrôle



(b) Schème parcourant le flot de données



(c) Partie d'un schème parcourant la structure interne d'une fonction Lisp



(d) Exemple de l'application du schème (c) sur une expression Lisp

Figure 1.8 Exemples de schèmes de générations de représentations analogiques

Détaillons maintenant les différentes arborescences présentées :

a) Ce premier exemple de schème de génération de représentations analogiques se base sur l'ensemble des définitions d'un parcours de flot de contrôle (que nous venons de voir). Son point d'entrée (*parcours-fdc*) est un ORS d'initialisation dont l'activité graphique associée pourrait être, comme dans la représentation analogique par des cités, de dessiner le sol sur lequel vont se placer différents objets. Cet ORS donne de manière inconditionnelle (indicateur *in-cond*) la main à l'ORS *traverse-fdc* qui a pour fonction de parcourir récursivement le flot de contrôle – par les conditions d'activation *fdc-CAR* (parcours en profondeur) et *fdc-CDR* (parcours en largeur). Il possède aussi un lien vers l'ORS *une-fonction* qui sera activé pour chaque élément atomique de la structure parcourue, dans ce cas pour chaque fonction du flot de contrôle.

b) Le second schème décrit le parcours du flot de données des programmes. La structure de ce dernier flot diffère de celle du flot de contrôle par la présence d'éléments atomiques complexes (en fait les couples « variables – fonctions »). En effet, comme nous l'avons décrit, le flot de données est considéré, dans notre système, comme un graphe dont chaque nœud est composé d'un couple « variable – fonction » indiquant la variable en jeu et son contexte. Ceci nécessite la présence de deux ORS particuliers pour le dessin de chaque élément des couples : *une-variable* et *une-fonction* respectivement liés aux indicateurs *dep-VAR* pour l'élément « variable » du nœud et *dep-FUNC* pour l'élément « fonction ».

c) Le dernier exemple de schème montre un extrait du parcours de la structure interne d'une fonction Lisp, plus particulièrement d'une expression de type *(if test expressions)*. Afin de décrire ce dernier schème, nous présentons, dans la figure 1.8.d un exemple d'expression Lisp (correspondant à une partie d'une fonction calculant le factoriel d'un nombre) mis en parallèle avec les ORS. Ainsi, le schème détecte la présence d'une expression *if* par la condition *(i-com . if)* entraînant l'activation de l'ORS *traite-if*. Les conditions d'activation du reste du schème permettent de sélectionner la partie de l'expression prise en compte par tel ou tel ORS. Par exemple, la condition d'activation *com-CDR* fait progresser le parcours de l'expression vers l'élément suivant : *traite-if-test-exprs* reçoit l'expression sans le *if*, puis *traite-if-exprs* sans le test et finalement, *traite-if-false* reçoit l'expression sans l'élément suivant le test et correspondant, en Lisp, à l'expression évaluée dans le cas où le test se solde par un succès. Les ORS traite-test et traite-if-true, de par leur condition d'activation et celle de l'ORS qui les suivent n'affectent pas le parcours de l'expression. En fait, ces ORS ont comme fonction de permettre l'ajout d'opérations graphiques (tant

le dessin d'un objet qu'une transformation) par rapport à des points précis d'une expression. Par exemple, le traitement du test d'une telle expression (ORS *traite-test*) entraîne, dans la représentation de programmes par des villes, un changement de couleur et la construction d'un cube délimitant l'espace occupé par le dessin des expressions contenues dans le test.

Les exemples d'aspects des programmes et de schèmes générateurs de représentations que nous venons de présenter s'appliquent à l'étude de la composition des programmes. La série d'aspects que nous allons maintenant présenter s'applique à l'étude du comportement des programmes.

III.1.2.2.3 Aspects des programmes liés à l'exécution

Les aspects des programmes liés à l'étude de leur comportement ont pour fin de permettre l'observation de la progression dans l'exécution d'un programme :

- La progression dans le flot de contrôle : les événements pris en compte sont l'entrée ou la sortie d'une fonction ou des instructions,
- Les différents accès aux variables manipulées par le programme : les événements sont l'accès en lecture ou l'accès en écriture aux variables.

De la même manière que pour les aspects structurels précédemment étudiés, ces aspects ne sont pas liés à un programme particulier mais peuvent s'appliquer à toutes les fonctions ou à toutes les variables d'un programme, sans que l'utilisateur ait à le spécifier pour chaque élément pris en compte. Toutefois, il demeure possible de définir, comme aspect du comportement des programmes, des caractéristiques particulières (fonction, expression ou variable d'un programme particulier) afin de restreindre l'observation à ce sous-ensemble du programme.

L'utilisation de ces aspects, dans l'élaboration d'une représentation analogique de programme, permet de spécifier des liens entre des événements liés à l'exécution d'un programme et des événements graphiques.

Dans la représentation analogique de programmes par des villes, par exemple, un ORS particulier (dont le graphique résultant est un personnage) a comme caractéristique de s'appliquer à la fois à la progression dans le flot de contrôle et à la progression dans l'exécution des expressions composant les fonctions. L'événement graphique lié à cet objet dans le contexte de son activation au cours de l'exécution d'un programme sera alors le déplacement de l'objet graphique lui correspondant dans la représentation.

Dans cette même représentation, l'accès à une variable présente dans le programme entraîne la reconstruction d'un objet graphique particulier lié au couple « variable – fonction », qui indique son emplacement spatial, et au type de la valeur contenue par cette variable, qui détermine son aspect gra-

phique. Ceci est effectué par un unique ORS dont la particularité est que l'objet graphique qu'il désigne est dépendant du contexte de son activation.

Les aspects des programmes, qu'ils s'appliquent à l'étude de leur composition ou de leur comportement, visent à créer des schèmes de génération de représentations analogiques basés sur telle ou telle facette d'un programme. Nous allons maintenant voir la manière dont s'articule le pendant graphique de ces parcours de structures.

III.1.2.3 Objets graphiques structurés

Comme nous l'avons vu dans la figure 1.2, les représentations analogiques de programmes construites en utilisant notre système résultent de la mise en relation des aspects des programmes avec des représentations graphiques. Nous allons maintenant présenter les représentations graphiques, les objets graphiques qu'elles permettent de manipuler ainsi que la manière dont elles sont structurées.

La construction des structures de représentations graphiques utilisées dans notre système afin d'illustrer les structures extraites des programmes, se base sur l'utilisation des fonctionnalités de placements relatifs d'objets graphiques par la gestion d'une pile de matrices de transformations qui agissent sur la position, sur l'échelle ou sur d'autres propriétés de dessin (couleur…) de la librairie de programmation graphique OpenGL[109]. Ces fonctionnalités conduisent à une définition relative des objets graphiques dans une même représentation.

Nous utilisons ces fonctionnalités de transformation comme ossature de la structure des représentations graphiques générées.

III.1.2.3.1 Structure d'une représentation graphique

Prenons l'exemple de la construction de la représentation du contenu d'une maison dans la représentation analogique de programmes par des cités. La figure 1.9 présente quelques-unes des étapes de la construction de cette représentation pour la fonction *tri* déjà présentée sous sa forme textuelle et arborescente dans la figure 1.4 (page 116). La première étape consiste dans le dessin d'une maison vide, composée d'un chemin d'accès, d'un jardin, d'un cube vide et d'un toît. La représentation graphique de l'ensemble des instructions composant la fonction représentée sera alors placée à l'intérieur du cube vide. L'organisation spatiale de ces instructions dépend alors de leurs types :

- Les représentations graphiques des composants d'une expression de type *condition* seront organisées horizontalement en largeur sui-

[109] L'annexe III de ce mémoire décrit notre implémentation de l'interface entre Xbvl et la librairie graphique Open GL.

vant l'axe des abscisses (par exemple, la figure (b) présente le placement du début de la condition présente dans la fonction *tri*),

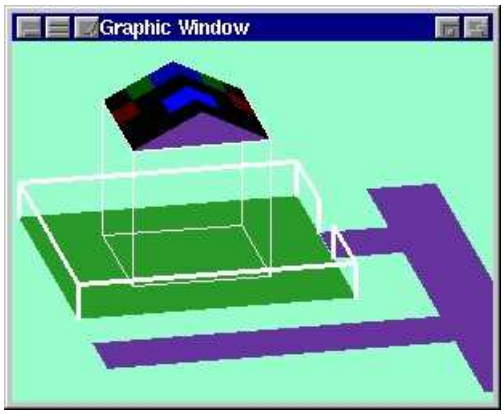
- Les représentations graphiques des composants d'une expression de type *lambda* ou *boucles* seront organisées horizontalement en profondeur,
- Les représentations graphiques des composants de toute autre expression seront organisées suivant l'axe des ordonnées.

Chacune de ces organisations aboutit à une subdivision de l'espace qu'elle divise en autant de sous-parties que le nombre d'expressions qu'elle contient. Ainsi, par exemple, l'expression *cond* contenue dans la fonction *tri* contient trois clauses. L'espace utilisé pour la représentation graphique de cette instruction sera alors subdivisé en quatre (une partie pour l'affichage de l'objet graphique correspondant à l'instruction *cond* plus trois pour les clauses contenues) suivant l'axe des abscisses.

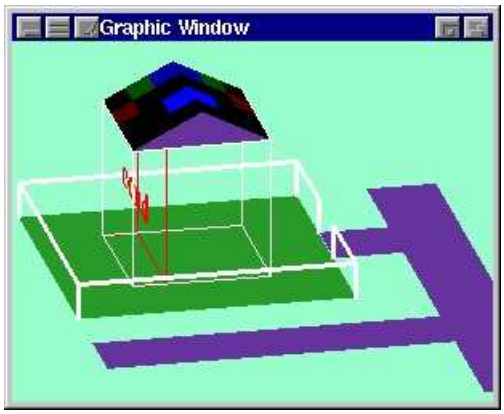
Le placement des différents objets graphiques représentant les expressions nécessite donc la possibilité d'un changement de repère tant au niveau de la position que de l'échelle. De plus, ces changements doivent être relatifs : un changement d'échelle ou de position ne doit s'appliquer qu'aux éléments inclus. On peut comparer ces changements successifs à des changements de repères dans lesquels non seulement la position et l'échelle peuvent changer mais aussi l'orientation ou certaines propriétés de dessin comme l'épaisseur des traits ou la couleur. Ce sont ces changements successifs de repères qui constituent la structure de la représentation graphique. Chaque élément graphique dépend alors, dans sa taille, sa position ou son orientation d'une série de transformations graphiques (les informations sur ces modifications successives ayant abouti au placement d'un objet graphique sont disponibles par la simple sélection de cet objet avec la souris (la figure (c) présente la figure finale de la représentation de la fonction et la figure (d) montre les informations déterminant le placement du début de la condition présente dans la fonction *tri*)).

III.1.2.3.2 Actions sur la structure d'une représentation graphique

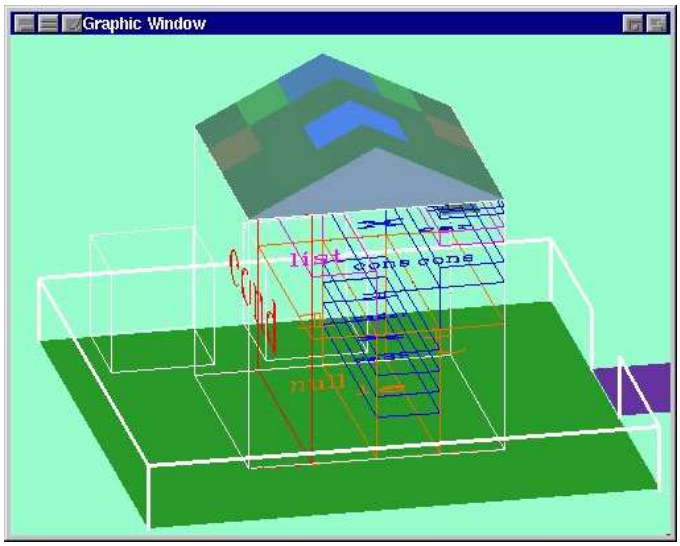
Le système Zeugma utilise les facilités de transformation pour la construction des représentations. Il les utilise également pour effectuer les animations des représentations. En effet, il est possible, dans notre implémentation de la librairie OpenGL dans Xbvl, de modifier dynamiquement les valeurs d'une transformation graphique appliquée à un objet. Ceci permet de construire des représentations dont chacune des parties pourra s'animer, ou de déplacer un nouvel objet graphique à l'intérieur d'une représentation tout en conservant la notion de structure de représentation.



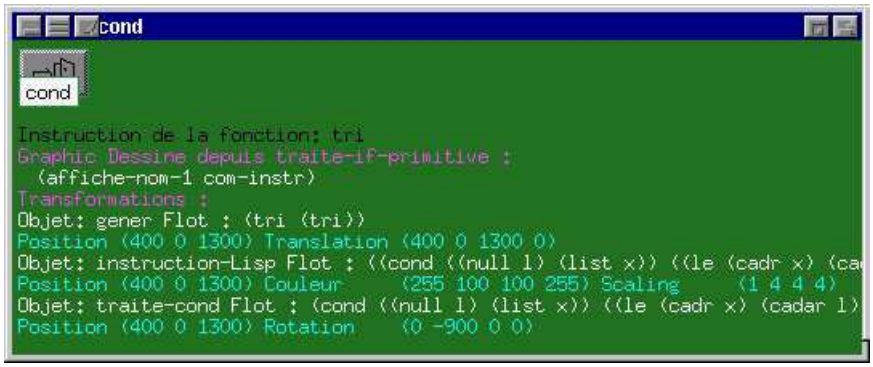
(a) Maison vide



(b) Placement de l'instruction de condition



(c) Maison finale



(d) Information sur la suite des transformations graphiques pour le placement de l'instruction de condition fournies par le système Zeugma

Figure 1.87 Etapes dans la construction de la représentation d'une fonction

III.1.2.4 Liens programmes – représentations : caractéristiques des ORS

Nous avons élaboré une méthode de mise en relation des deux types de structure que nous avons décrit précédemment (cf. section 2 du chapitre II). Cette méthode se rapproche de la Structure Mapping Theory de Gentner et de la théorie des analogies de Hofstadter par le fait qu'elle a pour finalité de construire des relations entre des structures caractérisant des objets appartenants à des domaines différents. La Structure Mapping Theory cherche à mettre en relation des structures logiques, décrivant des phénomènes appartenant à des domaines distincts, alors que notre méthode permet la construction d'une structure (graphique) par rapport à une autre (extraite des programmes). Ainsi, Zeugma ne cherche pas à faire correspondre deux structures appartenant à des domaines différents mais, se basant sur l'utilisation des Objets de Relation Structurelle (ORS) dont nous allons décrire ici les principales caractéristiques, il créé une représentation graphique analogue à un programme Lisp.

III.1.2.4.1 Parcours d'aspects des programmes

Comme nous l'avons vu dans le paragraphe 1.2.2.1 (page 112), les schèmes générateurs de représentations analogiques sont composés d'un ensemble d'ORS liés entre eux par des conditions d'activation. Ainsi, les propriétés principales des ORS sont les spécifications d'enchaînement vers d'autres ORS, définissant un parcours d'un aspect sur les programmes. La description des propriétés de ces enchaînements entre ORS peut se résumer ainsi :

- Chaque ORS est issu d'une condition d'un aspect des programmes, constituant sa condition d'activation ou son *contexte*.
- Le nombre des enchaînements à partir d'un ORS n'est pas limité et il peut être nul.
- L'enchaînement effectif, au moment de la génération de la représentation analogique, entre un ORS et un autre est lié à l'application d'une condition relative aux aspects des programmes sur un programme particulier.

Ces trois caractéristiques permettent de définir des ensembles d'ORS (ou schème de génération de représentations analogiques). En considérant les différents aspects des programmes comme autant de graphes calculés à partir d'un programme particulier, les schèmes de générations de représentations analogiques peuvent alors êtres considérés comme des automates décrivant le parcours de ces graphes.

III.1.2.4.2 Liens avec la génération de représentation graphique

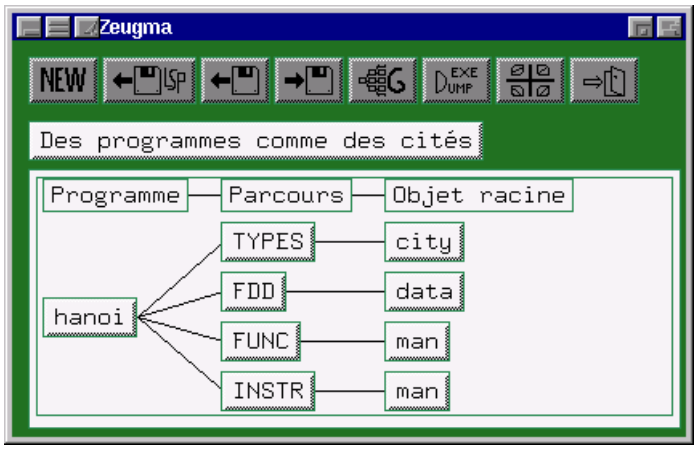
La définition d'un ensemble d'ORS permet la création d'un schème mettant en évidence un aspect des programmes. De plus, les ORS possèdent des propriétés graphiques permettant la génération d'une représentation analogique de l'aspect spécifié. Ces propriétés graphiques sont :

- La génération d'un objet graphique au moment de l'activation d'un ORS. Cette activation étant liée à la vérification d'une condition liée à un aspect des programmes, l'objet graphique généré peut alors être mis en relation directe avec celle-ci. Ainsi, par exemple, dans la représentation de programmes par des cités, chaque maison est liée à une fonction particulière du programme.
- L'enchaînement d'un ORS vers un autre peut entraîner la génération d'une transformation graphique. Cette transformation, utilisant les piles de transformations, sera alors liée avec la condition d'activation. Dans la représentation de programmes par des cités, ces transformations sont utilisées pour spécifier l'organisation des maisons ou l'organisation des pièces dans les maisons suivant le type d'expression rencontré.
- Si une condition d'activation liée à l'observation du comportement des programmes a été spécifiée pour un ORS, son avènement au cours de l'exécution du programme entraînera l'activation de l'ORS. L'action graphique produira alors une transformation de la représentation (modification d'un objet existant dans son aspect ou de sa position, comme pour le personnage en déplacement dans la cité) ou la création de nouveaux objets graphiques (comme pour l'affichage des valeurs manipulées dans les jardins des maisons).

En résumé, pour construire une représentation analogique de programme, on doit, en premier lieu, décider de la partie ou de l'ensemble de parties des programmes à traiter, ensuite, du schème de génération à appliquer à ces aspects, et enfin, des actions graphiques générées à chaque étape du parcours des caractéristiques par le schème.

Nous allons maintenant présenter l'interface proposée par notre système afin d'aider la création, la mise au point et l'expérimentation des représentations analogiques de programmes.

III.2 Zeugma comme interface de création d'une représentation analogique



Description des boutons de commande :

Création d'une nouvelle analogie (§ 2.3.2)


Chargement d'un fichier définissant un programme Lisp


Chargement d'un fichier de définition d'une analogie (Annexe)


Sauvegarde de la définition de l'analogie courante (Annexe)


Installation de l'éditeur de représentation (§ 2.4.2.1)


Création / Consultation d'un historique (§ 2.4.3.3)


Installation des points de vue multiples (§ 2.4.3.2)


Sortie du système

Figure 2.1 Interface principale du système Zeugma

III.2.1 Une interface pour la création de représentations analogiques de programmes

III.2.1.1 Etapes de la construction d'une représentation analogique

Dans notre système, les représentations analogiques de programmes sont élaborées à partir de schèmes de génération de représentations. Ces schèmes permettent la création de liens analogiques entre caractéristiques de programmes et représentations graphiques.

Les différentes étapes dans la construction d'une représentation analogique sont :

1) Le choix du domaine de départ de l'analogie : les aspects des programmes qui sont pris en compte par l'analogie. Ces aspects distin-

guent les observations de la composition des programmes (en se basant par exemple sur le flot de contrôle ou de données) des observations du comportement du programme (pour lesquels l'objet de l'observation sera par exemple les modifications successives de telle ou telle variable). Ce premier choix décide de la base de construction des représentations graphiques. En effet, pour des représentations analogiques pouvant combiner différents aspects (comme dans notre exemple de représentation analogique de programmes par des cités), un seul constituera toutefois la base de construction de la représentation graphique. Dans notre exemple, l'organisation du plan de la ville, construit à partir de la classification des fonctions suivant le type d'instruction utilisée.

2) Préciser, pour chacun des aspects choisis, l'ensemble des ORS actifs lors de son parcours. Ainsi, des représentations analogiques de la composition des programmes pourront mettre en évidence des particularités syntaxiques tout en ignorant d'autre.

3) Enfin, pour chaque ORS du schème, indiquer les opérations graphiques qui lui sont associées.

Pour détailler chacune de ces étapes nous allons décrire la construction de la représentation analogique de programme par des araignées sur une toile présentée dans le second chapitre de ce mémoire.

III.2.1.2 Choix des domaines d'application de l'analogie

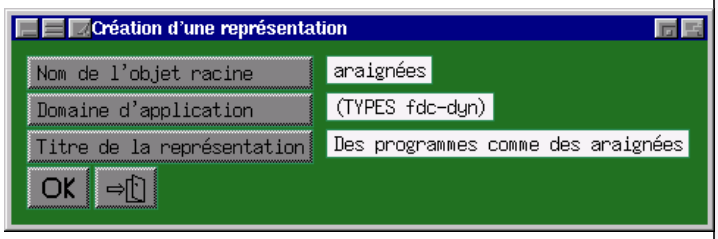
Le premier choix dans la construction d'une nouvelle représentation analogique de programme (par la commande NEW de la fenêtre d'interaction principale de Zeugma) est celui du domaine d'application de l'analogie, qui déterminera le ou les aspects des programmes auxquels elle s'intéresse. La figure 2.2.b présente la liste des domaines actuellement disponibles dans Zeugma. Cette liste décrit en fait les aspects définis actuellement dans le système.

Les aspects liés à l'étude du comportement des programmes sont, comme nous l'avons précisé précédemment, figés. Ils permettent l'étude du *comportement* :

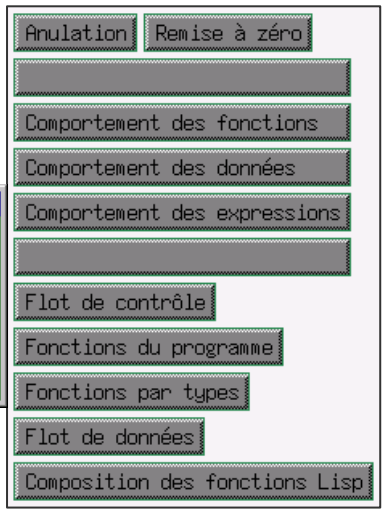
1) *Des fonctions* : la représentation sera générée à partir d'un attachement placé à l'entrée et à la sortie des fonctions formant le programme.

2) *Des données* : la génération de la représentation se fait à partir de l'accès aux différentes variables utilisées dans le programme étudié.

3) *Des expressions* : dans ce dernier cas, c'est l'entrée et la sortie des différentes expressions du programme qui aboutira à la génération d'une représentation.



(a) Spécifications de la nouvelle analogie



(b) Liste des domaines

Figure 2.2 Création d'une nouvelle représentation analogique

Les aspects liés à l'étude de la composition des programmes sont générés à partir de leurs définitions. Ainsi, les choix apparaissant dans la liste de la figure 2.2.b résultent des aspects définis dans Zeugma et ils seront augmentés automatiquement si de nouveaux sont définis. Ces choix sont actuellement :

1) Le *Flot de contrôle*, la représentation sera construite à partir du parcours du flot de contrôle des programmes étudiés.
2) Les *Fonctions du programme*, la représentation sera alors construite à partir du parcours de la liste des fonctions utilisées dans le programme.
3) Les *Fonctions par types*. La différence avec le choix précédent est que dans ce cas la liste des fonctions sera triée suivant le type de primitive qu'elle utilise[110].
4) Le *Flot de données*, la génération de la représentation sera effectuée à partir du parcours du flux de données des programmes étudiés. Pour ce dernier parcours, il sera possible de spécifier la ou les données initiatrices du flux de données déterminant alors si la représentation concerne toutes les variables utilisées dans le programme ou uniquement un sous-ensemble de celles-ci.
5) La *Composition des fonctions Lisp*, les éléments parcourus seront les expressions composant les fonctions du programme étudié.

Les analogies que nous pouvons construire avec notre système Zeugma permettent la combinaison sélective des différents domaines. Il est possible

[110] C'est ce domaine qui est utilisé comme base de construction de la représentation analogique de programmes par des cités.

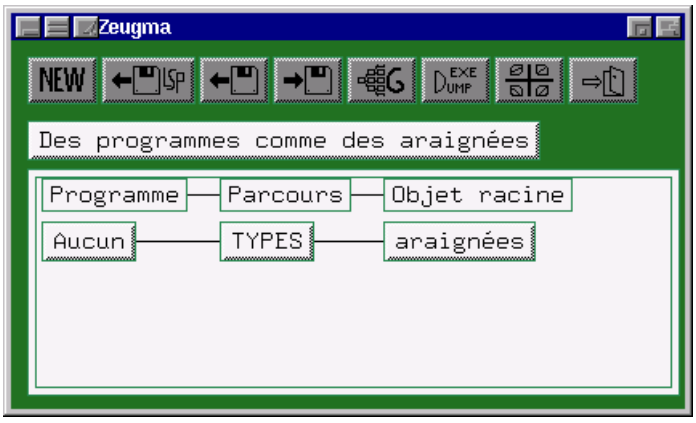
de choisir comme domaine de base la combinaison des parcours des flux de contrôle et l'accès aux entrées et sorties de fonctions (comme dans la représentation de programmes par des araignées sur une toile) ou encore une combinaison des flots de contrôle et de données, de la composition et des différents aspects dynamiques lors de l'observation de l'exécution du programme (comme dans la représentation de programmes par des villes).

La représentation analogique que nous allons construire (la représentation du comportement des programmes par des araignées en mouvement sur une toile), combine deux aspects des programmes :

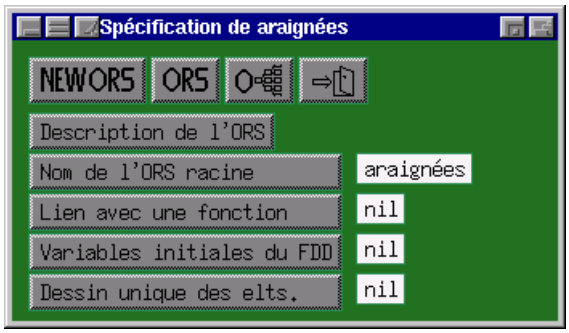
1) Le parcours de la liste des fonctions organisées suivant leur type (choix *Fonctions par types* apparaissant par l'indicateur *TYPES* dans la spécification de la nouvelle analogie) permettant de construire la représentation initiale de toutes les fonctions du programme comme autant d'araignées positionnées sur une toile.
2) Le *Comportement des fonctions* (indicateur *fdc-dyn*) permettant de lier la représentation initiale des fonctions avec leur comportement pendant l'exécution et ainsi d'animer les araignées et de dessiner les liens symbolisant les échanges de données.

Une fois le domaine d'origine choisi, le système génère automatiquement les objets de relation structurelle (ORS) correspondant au parcours des aspects de celui-ci. En effet, le processus mis en œuvre pour aider à la construction de représentation analogique utilise des schèmes de générations analogiques *à compléter*, des sortes de *patterns* ou *filtres* décrivant des parcours types des domaines concernés et laissant à l'utilisateur à la fois le soin de préciser le contenu *graphique* de l'analogie, et celui d'adapter le parcours aux critères particuliers qu'il désire observer.

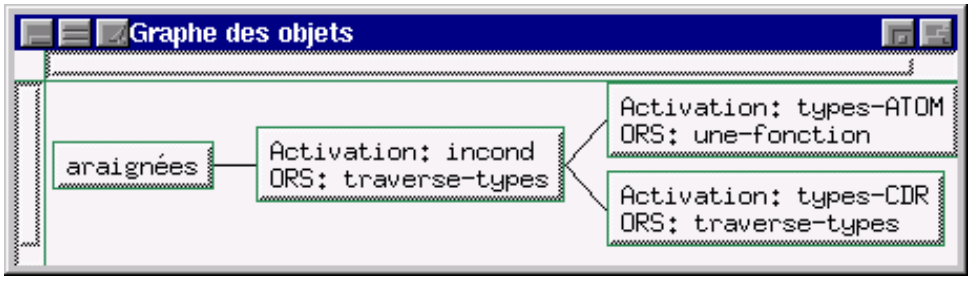
Ainsi, la figure 2.3 présente les fenêtres du système Zeugma correspondantes aux objets générés pour notre nouvelle représentation analogique. La fenêtre de la figure 2.3.a indique la présence d'un parcours de l'ensemble des fonctions composant le programme, triées suivant le type de primitive utilisée et qui n'est pour l'instant lié à aucun programme. Celle de la figure 2.3.b présente les spécifications de l'objet de relation structurelle initiateur du parcours (ou *ORS racine*) et celle de la figure 2.3.c le schème de génération type sous la forme d'une structure arborescente.



(a) Interface principale prenant en compte la nouvelle représentation analogique



(b) Spécifications de l'ORS *racine* de la nouvelle représentation



(c) Schème de génération de la nouvelle représentation

Figure 2.3 Fenêtres générées automatiquement par Zeugma

Nous allons maintenant détailler les différents choix présents dans les menus de notre système permettant d'une part de préciser le contenu graphique de l'analogie et d'autre part de modifier les schèmes de génération eux-mêmes.

III.2.1.3 Les Objets de Relation Structurelle

Après cette première étape, la définition d'une représentation analogique de programme consiste en l'affectation des propriétés graphiques des différents ORS définissant les étapes du parcours des domaines sélectionnés. Les ORS sont structurés sous forme de graphes (présentés dans la figure 2.2.c sous la forme d'un arbre). La racine de ces graphes, ou point d'entrée du processus de génération de l'analogie, est constituée par l'ORS dit *racine*. C'est à partir de cet ORS que le parcours des caractéristiques liées au domaine spécifié et extraites d'un programme particulier vas être initié. Ainsi, la propriété principale de cet ORS (nommé *araignées* dans la figure 2.3) est qu'il est indépendant d'une condition d'activation appliquée au programme et qu'il sera activé par la mise en œuvre de la génération de la représentation analogique.

III.2.1.3.1 L'O.R.S. racine

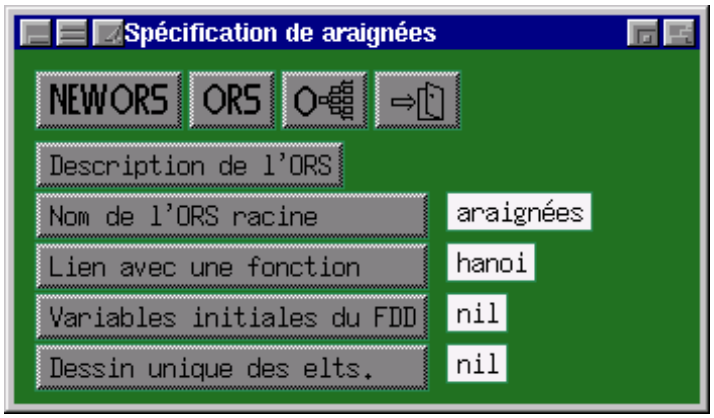


Figure 2.4 Fenêtre de spécification des ORS racine

L'ORS racine, outre la position de point d'entré du graphe des ORS définissant le schéma générateur d'une représentation analogique, contient la spécification des propriétés suivantes :

- Le point d'entrés du programme étudié (champ *Lien avec une fonction*).
- Dans le cas d'une génération basée sur le parcours du flux de données, l'ORS racine contient la liste des variables appartenant au programme étudié à prendre en compte comme points d'entrée. Une simple commande permet en outre, si l'on veut un flot de données complet, de spécifier la prise en compte de toutes les variables du programme.

- La dernière spécification de l'objet racine (*Dessin unique des elts.*) concerne la possibilité d'indiquer à Zeugma que les éléments des différents aspects pris en compte doivent chacun être représenté par un objet graphique unique. Ainsi, par exemple, dans la figure 1.2 présentant le flot de contrôle d'un programme de parcours de graphes, il est possible de spécifier si les éléments apparaissant plusieurs fois (comme par exemple *elim_rebond*, *tri*, …) doivent apparaître plusieurs fois dans la représentation analogique ou, du fait qu'ils désignent le même élément du programme, une seule fois.

De plus, la fenêtre de spécification des objets racine donne accès aux outils de création (

), de spécification (

) ou de visualisation de l'arborescence () des ORS dont ils sont racine.

Ainsi, les propriétés liés aux ORS racine concernent directement la construction de la représentation analogique. Ces propriétés permettent en fait de préciser d'une part le programme étudié lors de la génération d'une représentation ainsi que les données visualisées, et, d'autre part, d'avoir accès à la création et la consultation de l'ensemble des ORS.

La figure 2.4 présente les caractéristiques de l'ORS racine de notre nouvelle représentation. Par rapport à la fenêtre de spécification originale (figure 2.3.b), seul le champ *Lien avec une fonction* à été modifié, indiquant que nous allons tester la représentation analogique sur le programme, dont le point d'entrée est la fonction *hanoi*, visant à résoudre le problème des tours de Hanoï. Ce programme intègre, de plus, une représentation graphique du déroulement de l'algorithme.

Les autres champs des propriétés de l'ORS n'ont pas changé du fait que notre représentation ne se base pas sur un parcours des flots de données et que l'aspect choisi (*Fonctions par types*) ne contient pas plusieurs fois le même élément.

L'étape suivante est de déterminer les liens graphiques entre les aspects des programmes et leur représentation analogique graphique ; en d'autres termes, il faut préciser les propriétés de chaque ORS ainsi que celles des liens unissant les ORS entres eux.

III.2.1.3.2 Propriétés des ORS

Tous les ORS possèdent les propriétés décrites en détail ci-dessous. Elles précisent d'une part la manière dont les ORS vont s'enchaîner au cours de la génération de la représentation analogique et, d'autre part, les actions graphiques liées autant à l'accès aux ORS qu'à leur enchaînement.

Ces propriétés, accessibles via la fenêtre présentée dans la figure 2.5, sont :

- La spécification des fonctions correspondant aux actions gra-

phiques exécutées au moment de l'activation de l'ORS (champ *Données graphiques*). Ce champ permet de spécifier le lien direct en caractéristique de programme (présente dans la condition d'activation de l'ORS) et un objet graphique particulier. Dans notre exemple, l'ORS affiché correspond aux fonctions du programme. Ainsi, pour chacune de ces fonctions, le dessin d'une araignée sera effectué (par l'exécution de la fonction Lisp *dessine-une-araignee*).

- Le champ *Reconstruction* permet d'indiquer au système si, lors des différentes utilisations d'un objet graphique, il faut reconstruire sa représentation ou non. Dans notre exemple, la reconstruction est nécessaire du fait du caractère particulier de chaque araignée. Par contre, dans la représentation des programmes par des cités, l'objet graphique correspondant au personnage en mouvement ne nécessitera pas de reconstruction graphique. La présence de ce champ est due à un souci d'optimisation du temps de construction de la représentation graphique finale : si cet indicateur a comme valeur *nil*, un seul objet graphique sera généré et réutilisé pour cet ORS même si il se trouve être activé plusieurs fois.
- La possibilité de générer l'activation d'autres ORS (champ *Objets Générés*). La sélection de ce champ entraîne l'apparition d'une nouvelle fenêtre permettant de spécifier les détails de l'enchaînement. Nous décrirons dans le paragraphe 2.1.4 cette fenêtre ainsi que son fonctionnement.

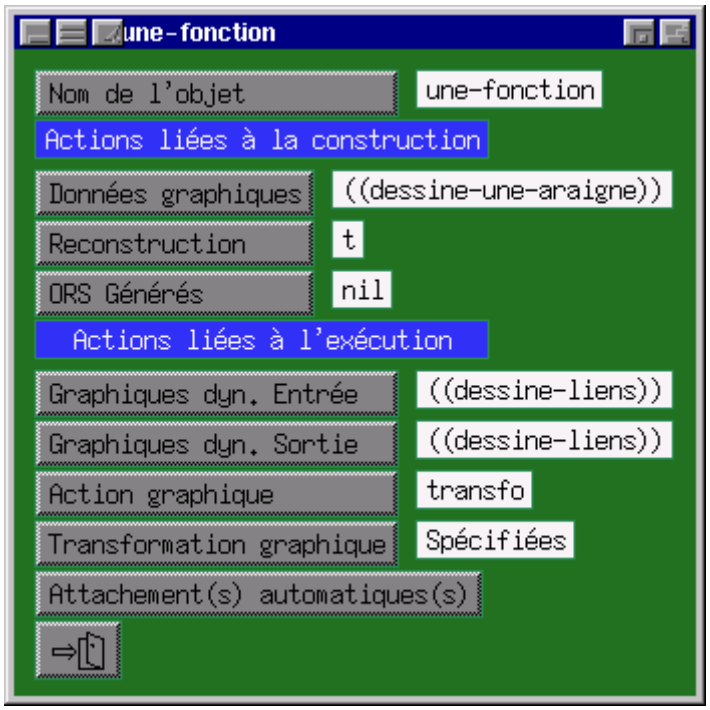


Figure 2.5 Fenêtre de spécification des ORS

Une seconde série de propriétés, plus particulièrement liées au comportement de l'ORS lors de son activation pendant l'exécution du programme, sont aussi présentes dans cette fenêtre de spécification des ORS :

- La spécification des opérations graphiques effectuées dans le cas d'une activation au moment de l'exécution d'un programme (le champ *Graphiques dyn. Entrée* pour l'entrée dans une fonction, une expression ou pour un accès à une variable et le champ *Graphiques dyn. Sortie* pour la sortie d'une fonction ou la fin de l'exécution d'une expression). La présence de ce champ, définissant un deuxième objet graphique pour un ORS permet d'utiliser les propriétés de l'objet graphique généré par le parcours des caractéristiques de la composition programmes avec des caractéristiques du comportement de programmes. Ainsi, dans notre exemple au moment de l'entrée et de la sortie des fonctions, des liens graphiques unissant les araignées leurs correspondants seront dessinés. Ces liens ne remplacent donc pas le dessin des araignées mais s'ajoutent à celui-ci.
- Dans le cas de la spécification d'actions graphiques liées à l'observation du comportement des programmes, Zeugma définit les types d'actions graphiques applicables aux objets graphiques liés aux l'ORS. L'action liée à un ORS est alors placée dans le champ *Action graphique*. Ces actions appliquées aux objets graphiques peuvent être (cf. figure 2.6) :
  - La reconstruction de l'objet graphique.
  - La reconstruction de l'objet après l'effacement du précédent.
  - Une transformation graphique (translation, rotation, changement d'échelle ou de couleur).
  - Un changement de la position sans la reconstruction de l'objet.
  - Un changement de la position avec la reconstruction de l'objet.
- Les données spécifiant la transformation dynamique de la position (champ *Transformation graphique*).
- La spécification de liens automatiques entre l'ORS et le comportement du programme. En effet, l'affectation de ces liens est en principe lié à un programme particulier mais, comme dans notre représentation de programmes par des araignées, un ORS peut être lié à tous les éléments d'un programme (dans notre cas, l'entrée et la sortie de toutes les fonctions du programme).

La figure 2.7 présente les différentes possibilités d'attachement automatique entre un ORS et les éléments des programmes. Ces attachements seront ainsi répercutés sur tous les objets du programme qu'ils désignent. Le dernier champ de cette fenêtre (*Test attaché aux variables*) conditionnera l'activation de l'ORS sur l'accès à une variable avec l'évaluation d'un test en-

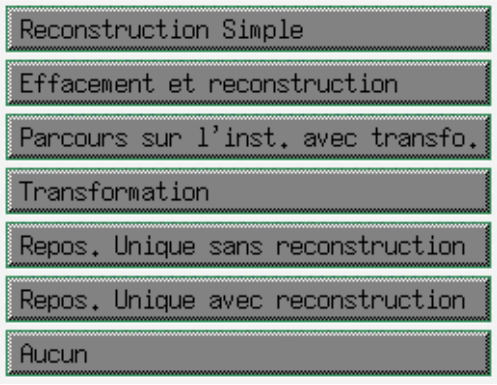


Figure 2.6 Actions sur un objet graphique au cours de l'exécution des programmes.

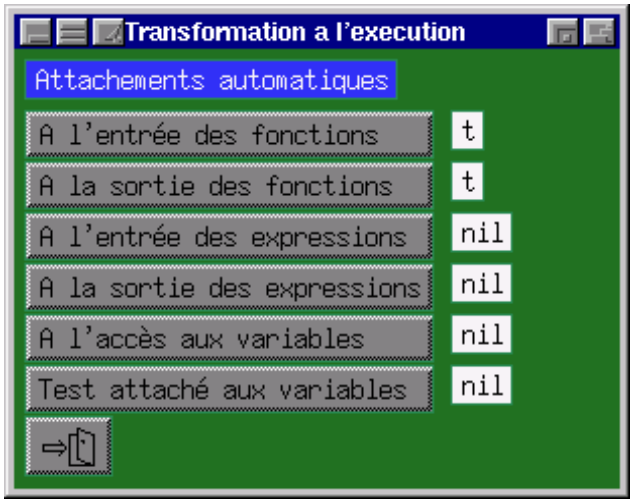


Figure 2.7 Spécification des attachements automatiques entre un ORS et les programmes.

tré par l'utilisateur.

Notre ORS aura ainsi les propriétés suivantes :

- il n'entraîne pas la génération d'autres ORS (champ *Objets Générés* à *nil*),
- à chaque activation, le dessin d'une araignée sera effectué,
- au cours de l'exécution du programme, l'araignée correspondant à la fonction exécutée changera de position et des liens seront dessinés.

En résumé, les ORS spécifient d'une part le parcours générateur de la représentation et d'autre part les objets graphiques générés par ce parcours ainsi que leur comportement dynamique dans le cas de leur utilisation durant l'exécution du programme. Nous allons maintenant détailler la spécification de l'enchaînement d'un ORS vers d'autres ORS.

III.2.1.4 Spécification du parcours des aspects des programmes

Le parcours des aspects des programme définit la manière d'utiliser le résultat des analyses effectuées sur les programmes lors de l'initialisation du processus de génération de la représentation analogique. Ces analyses fournissent des graphes (flots de contrôle, de données, …) et le processus de génération d'une représentation analogique consistera dans le parcours de ces graphes et de l'exécution des opérations graphiques liées à ce parcours. La spécification des objets générés à partir d'un ORS constitue le point central de la spécification de ces parcours.

III.2.1.4.1 Construction de la génération

La construction de la génération de l'activation d'un nouvel ORS *but* à partir d'un ORS, que nous nommerons *d'origine*, est constituée de trois étapes :

1) le choix de l'ORS but,
2) le choix de la condition d'activation de cet ORS,

la spécification (optionnelle) de transformations graphiques ayant lieu au moment du passage de l'ORS d'origine à l'ORS but.

Lors de la sélection du champ *Objets Générés* de la fenêtre des propriétés d'un ORS, Zeugma présente la liste des objets générés à partir celui-ci (figure 2.8). Cette liste indique la condition nécessaire à l'enchaînement vers le prochain ORS, le nom de celui-ci et l'existence ou non d'une transformation graphique au moment de l'enchaînement.

Pour chaque enchaînement, il est possible de spécifier (figure 2.9) :

- La condition d'*activation* liée au parcours,
- L'*ORS* suivant,
- La ou les *transformations* graphiques.

La condition d'enchaînement assume une double fonction :

1) l'indication de *contexte* programmatoire, sa vérification indiquant la présence d'un certain type d'élément (une fonction, un certain type d'instruction, etc.) lié au parcours d'un aspect particulier (nous détaillerons dans la section suivante les différentes conditions d'activations liées aux différents aspects actuellement définis dans Zeugma),

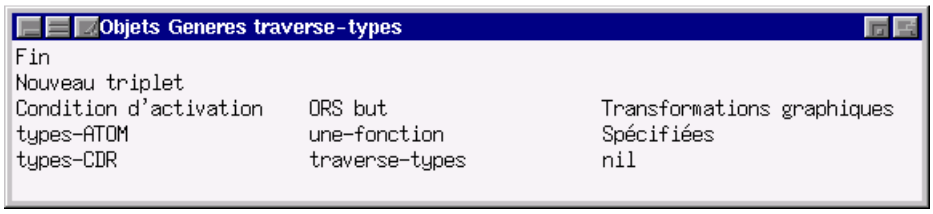


Figure 2.8 Liste des ORS générés à partir de l'ORS traverse-types

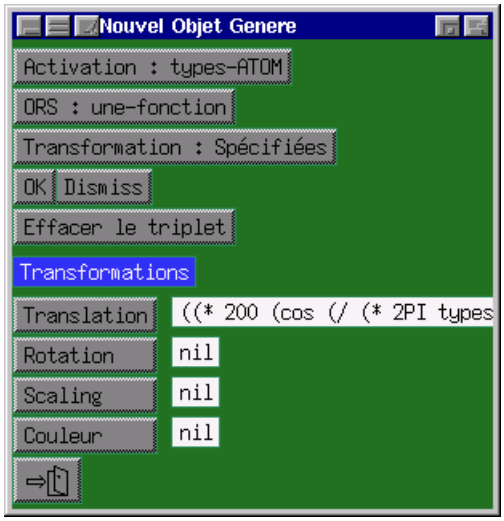


Figure 2.9 Propriétés du lien entre l'ORS traverse-types et l'ORS une-fonction

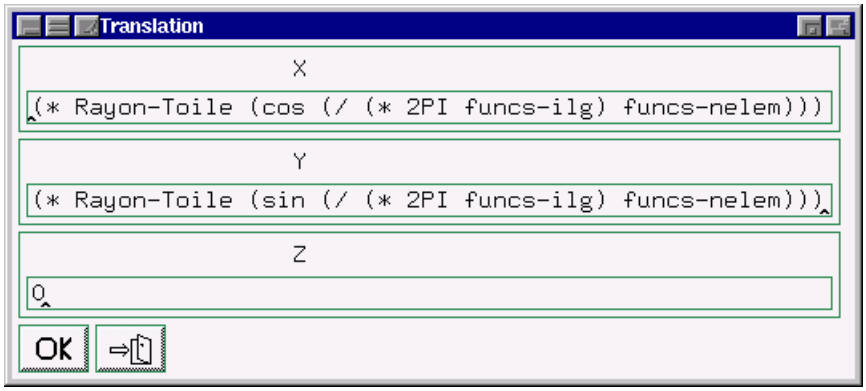


Figure 2.10 Spécifications de la translation effectuée lors de la transition entre les deux ORS

2) la progression dans le parcours d'une structure décrivant le résultat d'une analyse propre à un aspect particulier d'un programme dont la représentation analogique est en cours de construction. En effet, l'ORS but recevra comme structure l'application de l'opération induite par la condition d'activation sur la structure originale. Ainsi, par exemple, les deux conditions d'activations présentées dans la liste de la figure 2.8 auront respectivement comme effet sur la structure la transmission de l'élément rencontré pour types-ATOM et la progression dans la liste des fonctions pour types-CDR.

Le champ *transformation* permet de spécifier les transformations graphiques mises en œuvre au moment du passage de l'ORS source à l'ORS but. Ces transformations graphiques peuvent être une translation (relative à la position de l'objet graphique généré par l'ORS source), une rotation ou un changement d'échelle (dans le repère résultant de l'enchaînement des transformations graphiques précédent l'activation de l'ORS source), ou un changement de couleur.

Les figure 2.8, 2.9 et 2.10 sont extraites de la spécification de l'enchaînement entre l'ORS traverse-type, dont la fonction est de parcourir l'ensemble des fonctions du programme triées selon leur type, et l'ORS une-fonction décrit précédemment.

La liste de la figure 2.8 montre l'enchaînement récursif permettant le parcours et, pour chaque élément de l'ensemble (condition d'activation types-ATOM), l'enchaînement vers une-fonction. Pour ce dernier enchaînement (dont le détail est présenté dans la figure 2.9), il est spécifié une translation spécifiée dans la figure 2.10. Cette translation utilise des variables (*types-ilg* et *types-nelem*) liées au parcours (préfixe *types*) et au contexte de son utilisation (*nilg* indique le rang[111] de la fonction liée à l'activation de l'enchaînement et *nelem* désigne le nombre d'éléments de l'ensemble des fonctions du programme représenté). Nous présenterons dans le paragraphe 2.1.6 une description des variables fournies par les différents parcours et utilisables dans la construction des représentations analogiques.

III.2.1.4.2 Choix des aspects étudiés

Les conditions de parcours des aspects des programmes, utilisées sous la forme des *conditions d'activations* dans la définition des liens entre ORS, sont classifiées par rapport au domaine auquel ils s'appliquent. Comme nous l'avons indiqué au paragraphe 2.1.1, nous distinguons deux types d'aspects :

1) Ceux liés à l'étude de la composition des programmes. Ce premier ensemble n'est pas figé et peut être augmenté par l'utilisateur. Ainsi, la description des conditions d'activations que nous allons pré-

[111] Puisque, pour cet exemple, nous avons choisi comme domaine de départ *fonction-par-type*, le rang indique le placement de la fonction dans la *liste-fonction*.

senter ici ne peut constituer une liste exhaustive de toutes les conditions possibles mais uniquement de celles actuellement disponibles dans Zeugma,

2) Ceux liés à l'étude du comportement des programmes. Contrairement au premier type d'aspects, ceux-ci sont prédéfinis dans Zeugma et ne peuvent être modifiés ou augmentés.

En plus de ces types d'aspects, Zeugma intègre la possibilité de définir des *aspects génériques*, permettant par exemple des enchaînements inconditionnels entre deux ORS ou des enchaînements liés à l'évaluation d'un test défini par l'utilisateur.

III.2.1.4.3 Conditions liées à l'étude de la composition des programmes

Les menus de conditions présentées ici (figure 2.12) sont automatiquement générés par Zeugma au moment de l'intégration des définitions des aspects des programmes[112] (cf. § 1.2.1.2). Les parties (a), (b), (c) et (d) de cette figure présentent les conditions liées aux parcours d'ensembles définissant sous forme de graphes (pour les flots de contrôle et de donnés) ou de listes (pour l'ensemble des fonctions ou l'ensemble des fonctions classées suivant le type d'instruction utilisé). Le parcours de ces ensembles est lié au choix d'une représentation interne commune sous la forme de *liste programme* arborescente pour les graphes ou simple pour les listes. Ainsi, le parcours de telle structure sera possible par l'utilisation de conditions d'enchaînement permettant une progression en largeur et en profondeur.

La partie (e) de la figure présente les conditions liées au parcours de la composition d'une fonction Lisp. Ces conditions, outre le cheminement dans la *liste fonction* définissant la fonction, permettent des parcours liés à des particularismes syntaxiques.

[112] Les définitions de ces aspects sont présentées en Annexe I.5 de ce mémoire.

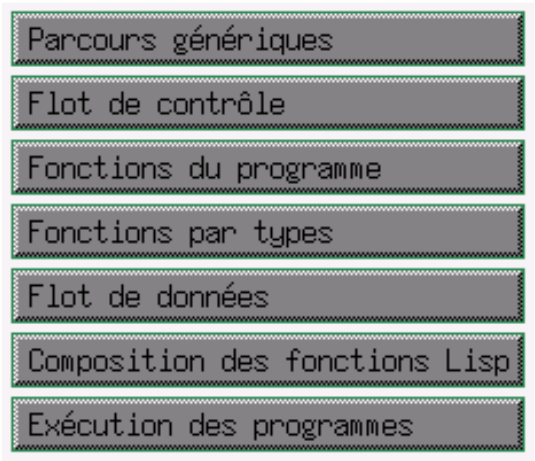
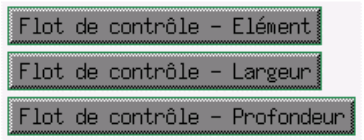
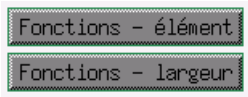


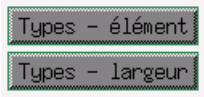
Figure 2.11 : Types de parcours actuellement définis dans Zeugma



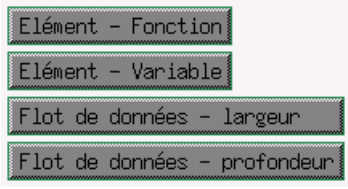
(a) Conditions sur le flot de contrôle



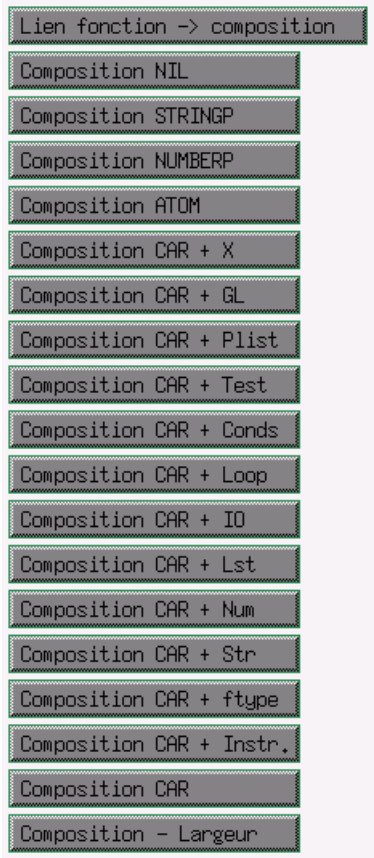
(b) Conditions sur la liste des fonctions



(c) Conditions sur les fonctions classées



(d) Conditions sur le flot de données



(e) Conditions sur la composition

Figure 2.12 Conditions des parcours liés à la composition des programmes.

Ainsi, les conditions nommées *Expression xxx* désignent l'initiation du

parcours d'une expression Lisp (ou parcours en profondeur) suivant notre typage des primitives du langage (cf. Annexe II.2). En effet :

- *Expression - Str* sera valide si l'expression examinée effectue une opération sur les chaînes,
- *Expression - Num* pour une opération arithmétique,
- *Expression - Lst* pour une opération sur les listes,
- *Expression - IO* pour une opération d'impression ou de lecture,
- *Expression - Loop* pour une répétition,
- *Expression - Cond* pour un branchement conditionnel,
- *Expression - Test* pour une opération booléenne,
- *Expression - Plist* pour une opération sur les p-liste,
- *Expression - GL* pour une opération graphique utilisant l'interface avec la librairie OpenGL de Xbvl,
- *Expression - X* pour une opération utilisant l'interface entre Xbvl et X Windows.

Les deux conditions suivantes demandent des données supplémentaires à l'utilisateur :

- *Expression - Instr.* pour une opération sur une instruction précise,
- *Expression - ftype* pour une opération sur une fonction d'un type particulier.

Enfin, *Composition CAR* sera validée si, dans un enchaînement entre deux ORS, aucune des précédentes conditions n'a été validée. Cette dernière correspond, dans les autres parcours, à un parcours en profondeur simple.

Les conditions *Composition NIL, STRINGP, NUMBERP* et *ATOM* seront valides si l'élément testé est atomique et est respectivement égal à *nil*, une chaîne, un nombre ou un atome.

La condition *Lien fonction -> composition* est particulière : elle permet d'initier le parcours de la composition d'une fonction à partir d'un élément atomique quel que soit le parcours dans lequel elle est placée. Cette condition sera utilisée pour faire le pont entre le parcours des fonctions classées suivant leur type et l'étude de leur composition dans la représentation analogique des programmes par des cités.

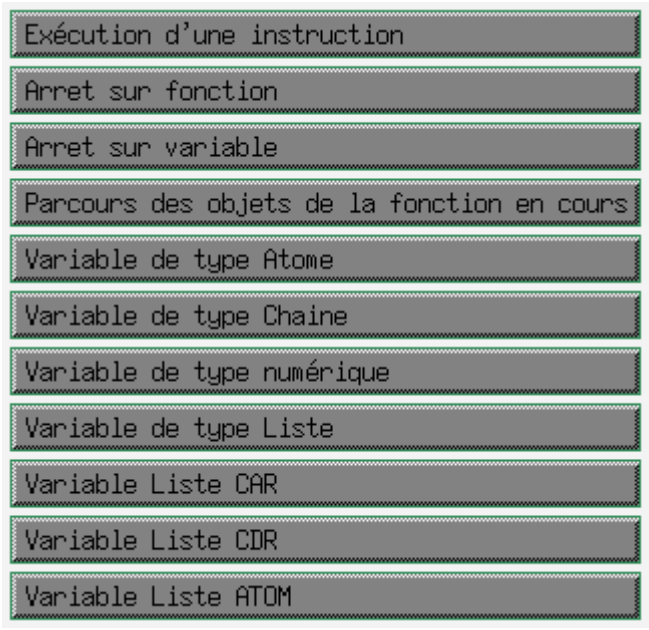


Figure 2.13 Conditions liées à l'étude du comportement des programmes

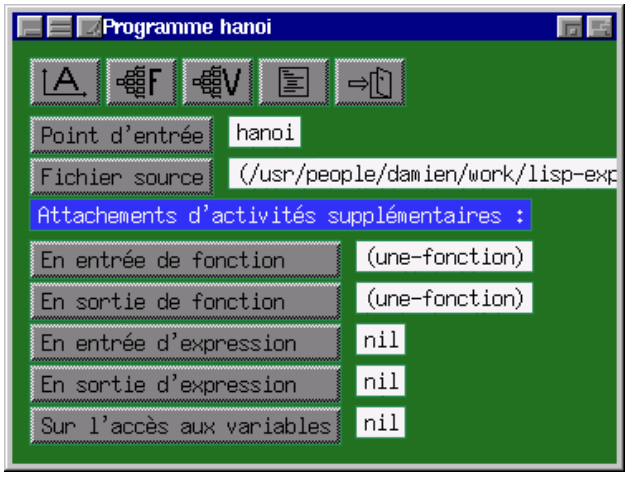


Figure 2.14 Information générale des attachements sur un programme

III.2.1.4.4 Conditions liées à l'étude du comportement des programmes

A la différence des caractéristiques liées à la composition des programmes, les conditions liées au comportement des programmes sont, dans notre système, figées.

Rappelons que les conditions que nous détaillons ici permettent de déclencher l'activation d'ORS dans un schème de génération de représentation analogique. Ainsi, elles désignent un contexte lors de l'exécution d'un programme (l'entrée ou la sortie d'une fonction, l'entrée ou la sortie de l'exécution d'une instruction ou l'accès à une variable), un contexte lié à une valeur manipulée (conditions liées aux types de valeurs), ou un parcours d'une valeur manipulée (dans le cas des listes).

La condition d'activation *Parcours des objets de la fonction en cours* est, elle, d'un autre type. Elle replace Zeugma, au moment de l'exécution du programme, dans le contexte de la génération d'une représentation liée à la composition du programme actif. Cette dernière condition est particulièrement importante pour traiter le cas des fonctions auto-modifiantes dont la représentation de la composition peut ainsi évoluer au cours de son exécution.

Dans le paragraphe suivant, nous allons détailler les procédures d'attachement d'activités supplémentaires au comportement du programme permettant d'activer des ORS au cours de l'exécution d'un programme.

La spécification des attachements entre un programme et une représentation analogique constitue la dernière étape de l'élaboration d'une représentation analogique de programme. En effet, elle permet de spécifier les liens dynamiques entre des points précis de l'exécution du programme et des animations ou des transformations de la représentation analogique.

III.2.1.5 Spécification des attachements liés à l'exécution des programmes

La première étape dans la spécification de ces liens est la consultation des informations sur le programme étudié. Ces informations, présentées dans la fenêtre de la figure 2.14, apparaissent lors de la sélection du point d'entrée du programme de la fenêtre d'interface principale de Zeugma (par le bouton du milieu de la souris). Cette fenêtre a pour fonction :

- de donner accès au résultat des analyses métriques des fonctions du programme (commande ), la figure 2.14 présente les résultats des analyses effectuées sur le programme choisi dans notre exemple. Nous détaillerons dans le paragraphe 2.1.6 la signification de chacune des colonnes de cette fenêtre.
- de présenter un résumé des attachements sur les fonctions et sur les variables du programme

- de donner accès aux outils permettant de spécifier ces attachements : les boutons de commande  et  permettent respectivement d'obtenir l'affichage des représentations arborescentes des flots de contrôle et de données du programme étudié telles qu'elles ont été présentées dans les figure 1.2 et 1.3 du paragraphe 1.2.1.1. Chaque élément de ces deux arborescences représente soit un élément du flot de contrôle du programme, soit de son flot de données. Ces éléments, eux-mêmes sélectionnables, permettent d'accéder aux informations des éléments qu'ils désignent et de les modifier.
- d'afficher le code source du programme () et de pouvoir ainsi placer des liens analogiques sur telle ou telle expression de celui-ci.

III.2.1.5.1 Les informations et attachements sur les fonctions

Les informations que notre système fournit sur chaque fonction du programme sont les suivantes (figure 2.16) :

- La liste des variables locales que la fonction définit et utilise. Ces variables regroupent l'ensemble des variables argument de la fonction ainsi que les variables que la fonction peut définir par l'utilisation de lambdas dans son code.
- La liste des variables globales que la fonction utilise. Ces variables, contrairement aux variables locales, ne sont pas issues d'une définition interne à la fonction.
- Le nom du fichier dans lequel cette fonction est définie.
- Le résultat des analyses sur cette fonction.

Dans la seconde partie de la fenêtre, une série de boutons de commandes permet de spécifier les différents types d'attachements sur la fonction : il est possible, outre la spécification d'un point d'arrêt à l'entrée ou à la sortie de la fonction, d'indiquer un attachement avec un ORS particulier pour les différents types de caractéristiques de programmes liés à l'observation de leur comportement. De plus, le bouton de commande  permet à toutes les fonctions appelées par la fonction d'avoir les mêmes attachements qu'elle.

Par exemple, la figure 2.16 montre les données de la fonction *hanoi*. Un ORS, *une-fonction*, dont l'action graphique a été décrite au paragraphe 2.1.3.2, est attaché à l'entrée et la sortie des fonctions. Cet attachement a été placé automatiquement lorsqu'il a été indiqué que le programme visualisé par notre représentation est le programme *hanoi*. Pour ce type de représentations analogiques (incluant un attachement automatique), ces fonctionnalités peuvent ainsi être utilisées pour indiquer à Zeugma que telle ou telle fonction du programme ne doit pas être prise en compte pour l'animation de la représentation graphique.

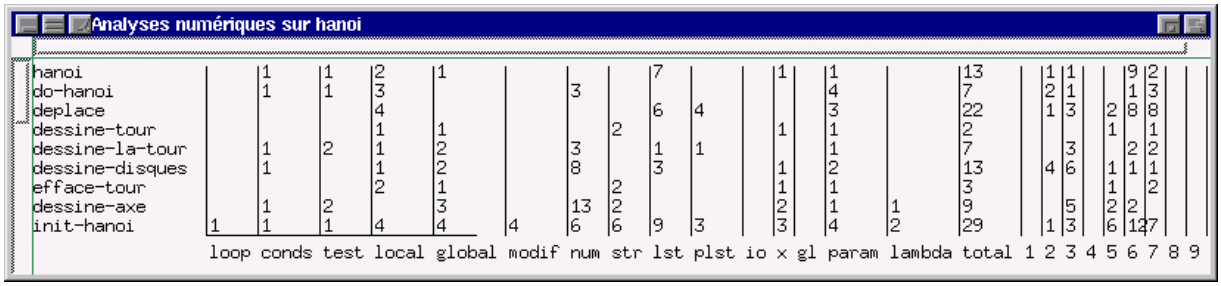


Figure 2.15 : Résultat des analyses de numération sur le programme *Hanoi*

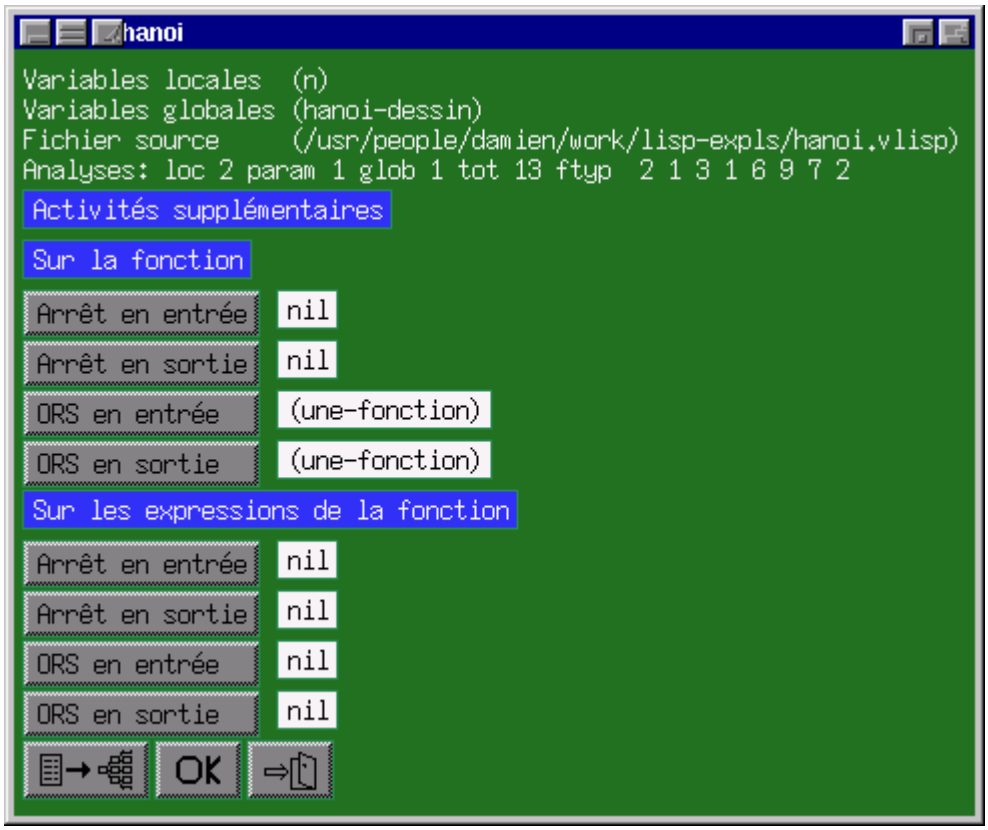


Figure 2.16 Attachements sur une fonction

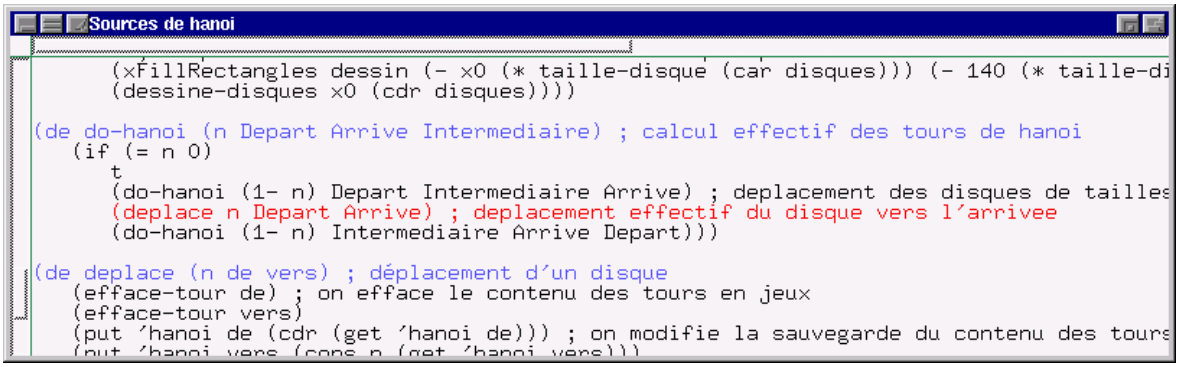


Figure 2.17 Attachement à une expression du programme

De plus, notre système permet l'attachement de l'activation d'un ORS particulier à l'entrée ou la sortie d'expressions du programme. Cet attache-

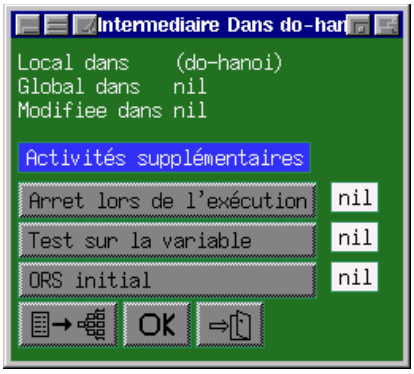


Figure 2.18 Attachements sur une variable

ment, effectué directement par sélection du code, rapproche Zeugma de systèmes de visualisations de programmes basés sur l'ajout d'opérations graphiques à l'intérieur même des programmes tout en ne modifiant pas le déroulement de l'exécution du programme (le code supplémentaire exécuté lors de l'activation de l'ORS n'ayant aucun effet de bord sur le comportement du programme) : la figure 2.17 montre le code source du programme *hanoi* auquel à été ajouté l'activation d'un ORS à chaque déplacement effectif d'un disque.

III.2.1.5.2 Les informations et attachements sur les variables

Pour les couples « variable – fonction » du flot de données, le système donne les informations suivantes (figure 2.18) :

- L'ensemble des fonctions dans lesquelles la variable observée est définie et utilisée de manière locale.
- L'ensemble des fonctions dans lesquelles elle est considérée comme variable globale.
- L'ensemble des fonctions dans lesquelles sa valeur est modifiée de manière destructive.

De la même manière que pour les fonctions, il est possible de spécifier un point d'arrêt sur l'accès à la variable au cours de l'exécution du programme. L'attachement de l'activation d'un ORS à l'accès à une variable peut être spécifié de deux manières différentes :

1) lié au contexte défini par le couple « variable – fonction » observé : l'ORS sera activé uniquement au moment de l'accès à la variable dans une fonction spécifique,
2) non lié à ce contexte : l'activation se fera alors à chaque accès à la variable indépendamment de la fonction dans laquelle cet accès a eu lieu.

De plus, il est possible de spécifier un *test sur la variable* dont l'évaluation conditionnera l'activation de l'ORS.

De la même manière que pour les attachements aux fonctions, il est

possible de répercuter automatiquement les attachements d'un couple « variable – fonction » sur tous les couples liés à ce dernier dans le graphe du flot de contrôle. La figure 2.17 montre les informations du couple « *mem – rechercher* », l'ORS *do-on-var* sera activé lorsqu'un accès à cette variable aura lieu dans cette fonction. L'action graphique liée à cet ORS est un dessin analogique de la valeur contenue dans la variable[113]. L'accès au contexte local peut être nécessaire dans quelques représentations analogiques (comme celle des programmes comme des cités où la représentation graphique de la valeur de la variable se situe dans le jardin de la maison représentant la fonction). Là, il est donc nécessaire de lier l'activation de l'ORS à une description complète du contexte dans laquelle elle a lieu : la variable et la fonction.

III.2.1.6 Données sur les parcours

Après avoir spécifié les données relatives au schème générateur de représentation analogique, il faut procéder à la construction des fonctions Lisp utilisées pour générer les différents objets graphiques de l'analogie et, comme c'est le cas dans notre nouvelle représentation, des fonctions spécifiant les données de transformations graphiques. Notre système propose un ensemble d'indicateurs et de fonctions spécifiques permettant de récupérer et d'utiliser des valeurs liées au programme étudié, à l'évolution de son exécution et au processus de génération de sa représentation analogique.

Ainsi, dans notre exemple de représentation analogique de programmes par des araignées en mouvement sur une toile :

- les objets graphiques utilisés pour représenter les araignées utilisent des données relatives à la fonction qu'elle représente,
- leur placement initial est calculé en fonction de son rang,
- leur déplacement au cours de l'exécution est calculé en fonction de leur position ainsi que de la position de la fonction appelante,
- les couleurs des liens les unissant sont calculées en fonction des valeurs transmises.

III.2.1.6.1 Indicateurs relatifs à la génération de l'analogie

Les indicateurs relatifs à la génération de l'analogie, et donc utilisables pendant celle-ci, sont de deux types :

1) Des indicateurs de progression dans la génération (table 2.1) : ils indiquent, par exemple, la position dans le graphe décrivant les différents flots.

[113] C'est avec ce mécanisme que nous pouvons aisément implémenter des outils d'observations tels que ceux proposés par les systèmes de visualisation de programmes ou d'animations d'algorithmes (cf. partie IV) sans être obligé, comme dans la plus par d'entre eux, de modifier le code source du programme.

2) Des indicateurs relatifs aux éléments pris en compte dans la génération de l'analogie (table 2.2), comme, par exemple, le nom de la fonction ou de la variable ou l'expression traitée, ou encore les données de la numération sur les fonctions.

Nom de la variable	**Description**	**Parcours**
next-nilg	Nombre d'éléments de l'élément suivant dans un enchaînement en profondeur.	*Tous les parcours*
funcs-nelem	Nombre d'éléments de l'ensemble des fonctions du programme représenté (c'est à dire le nombre de fonctions du programme).	*Fonctions du programme*
funcs-ilg	Position dans le parcours de la liste des fonctions.	*Fonctions du programme*
types-nelem	Nombre d'éléments de l'ensemble des fonctions du programme classées par type.	*Fonctions par type*
types-ilg	Position dans le parcours de la liste des fonctions.	*Fonctions par type*
fdc-ilg, fdc-ipr,	Indicateur de largeur et de profondeur dans le graphe	*Flot de contrôle*
fdd-ilg, fdd-ipr,	Idem, pour le flot de données	*Flot de données*
com-ilg, com-ipr, com-iilg, com-nilg	Idem, pour le parcours de la structure « liste – fonction » plus la position et nombre d'éléments de l'expression actuellement parcourue.	*Composition des fonctions Lisp*

Table 2.1 Indicateurs relatifs à la progression dans la génération d'une analogie

Nom de	**Description**	**Parcours**

l'indicateur		
com-instr	Instruction actuellement parcourue	*Composition*
com-func	Fonction dont la composition est parcourue	*Composition*
fdc-func	Fonction actuellement traitée	*Flot de contrôle*
types-func	Idem.	*Fonctions par type*
funcs-func	Idem.	*Fonctions du programme*
fdd-var	Variable du couple « variable – fonction »	*Flot de données*
fdd-func	Fonction du couple « variable – fonction »	*Flot de données*

Analyses numériques sur les fonctions :

elles sont mises à jours pour chaque fonction du programme

Nom de l'indicateur	**Description**
com-loop	Nombre (N.) de boucles
com-lambda	N. de lambda expressions
com-num	N. d'opérations sur les numériques
com-lst	N. d'opérations sur les listes
com-str	N. d'opérations sur les chaînes
com-plst	N. d'opérations sur les p-listes
com-io	N. d'opérations d'entrées ou sorties
com-conds	N. de conditions
com-test	N. d'expressions booléennes
com-x	N. d'opération vers l'interface avec X Windows
com-gl	N. d'opérations graphique utilisant OpenGL
com-total	N. total d'expressions
com-local	N. de variables locales
com-param	N. de variable paramètre
com-global	N. de variables globales utilisées
com-modif	N. de variables modifiées

Table 2.2 Indicateurs relatifs à la génération d'une analogie (fin)

Ainsi, dans notre exemple, la position initiale des araignées est calculée suivant l'équation 2.1 aboutissant ainsi à un positionnement automatique des

fonctions sur un cercle de rayon N quel que soit le programme étudié.

$$\begin{cases} X = N * \cos(\dfrac{2\Pi * types - ilg}{types - nelem}) \\ Y = N * \sin(\dfrac{2\Pi * types - ilg}{types - nelem}) \end{cases}$$

Equation 2.1 : calcul de la position initiale des araignées

III.2.1.6.2 Indicateurs relatifs au comportement des programmes

Les indicateurs relatifs au comportement des programmes permettent, suivant le type d'attachement, d'avoir accès aux valeurs et données manipulées par le programme, aux expressions, et aux noms de fonctions du programme en cours d'exécution. Ainsi, ces indicateurs servent à construire des représentations analogiques prenant en compte le type de valeur manipulé par les programmes sans pour cela avoir à lier cette représentation à un programme particulier (comme, par exemple, dans la représentation de programme par des cités où les valeurs manipulées par les différentes fonctions sont représentées analogiquement par des sphères et des arbres).

Ces indicateurs sont de deux types :

1) Des indicateurs liés directement au programme en cours d'exécution (table 2.3)
2) Des indicateurs et des fonctions liés à la position des objets graphiques représentant des éléments du programme (table 2.4).

Par exemple, la figure 2.19 présente le code source de la fonction Lisp calculant le déplacement des araignées au cours de l'exécution du programme représenté. La nouvelle position est calculée suivant l'algorithme suivant :

- si l'appel de la fonction est récursif, l'araignée se déplace sur l'axe des Z (en hauteur, elle s'envole !),
- dans les autres cas, l'araignée se déplace vers l'araignée représentant la fonction appelante.

Indicateur	**Description**
exe-nfunc	Total du nombre d'accès aux fonctions

exe-func	Fonction actuellement en cours d'évaluation
exe-p-func	Fonction précédemment évaluée
exe-args	Liste des variables arguments de la fonction en cours d'évaluation
exe-n-in	Nombre d'entrée dans la fonction ou nombre d'accès à la variable
exe-n-out	Nombre de sorties de la fonction
exe-is-var	Arrêt sur une variable ?
exe-in	Arrêt en entrée ou en sortie ?
exe-value	Valeur de la variable source de l'activité supplémentaire

Table 2.3 Indicateurs relatifs à l'observation du comportement d'un programme

Indicateurs	**Description**	
GA-X, GA-Y, GA-Z	position absolue de l'objet graphique	
p-x, p-y, p-z	position absolue du précédent objet	
Nom de la fonction	**Paramètre**	**Description**
Z-Undo-Transformations	Aucun	Repositionnement dans un repère absolu
Z-Get-Position	Un élément du programme et un parcours	Retourne la liste (X Y Z) de la position absolue de la représentation graphique d'un élément du programme pendant un parcours
Z-Get-X	Idem.	Idem, retourne uniquement X
Z-Get-Y	Idem.	Idem, retourne uniquement Y
Z-Get-Z	Idem	Idem, retourne uniquement Z
Z-Get-Func-Analyses	Une fonction du programme	Récupération des numérations sur une fonction du programme
Z-Get-Type	Aucun	
Z-Fonction-Racine	Une fonction du programme	

Table 2.4 Indicateurs et fonctions liés à la position des objets graphiques

```
; La fonction deplace-araignee reçoit en argument l'axe (x, y ou z) dont la nouvelle
; valeur est à calculer
(de deplace-araignee (axis)
```

```
(cond
 ; appel reccursif : la fonction appelante est égale à la fonction appelée
 ((eq exe-func exe-p-func)
  (selectq axis  (x GA-X)  (y GA-Y) (z (* exe-elem-nth 5))  (t 0)))
; le rapprochement s'effectue uniquement à l'entrée des fonctions, indiquant ainsi une
 ; attirance particulière entre deux fonctions (ce rapprochement n'est pas annulé à la
 ; sortie des fonctions).
 (exe-in
  (selectq axis
       (x (fix (+ GA-X (/ (- p-x GA-X) 50.0))))
       (y (fix (+ GA-Y (/ (- p-z GA-Z) 50.0))))
       (z GA-Z)
       (t 0)))))))
```

Figure 2.19 Source de la fonction Lisp calculant le déplacement des araignées

Pour pouvoir implémenter cet algorithme, il faut :

- savoir si l'appel de la fonction est récursif, ce qui est le rôle des indicateurs exe-func et exe-p-func indiquant respectivement le nom de la fonction en cours d'exécution et celle qui l'était précédemment,
- pouvoir calculer un déplacement relatif entre deux objets de la représentation analogique. Les indicateurs GA-X, GA-Y, GA-Z, p-x, p-y et p-z indiquant respectivement la position de l'araignée courante et celle représentant la fonction appelante, peuvent dans le cas présent être utilisés directement. En effet, le placement initial des araignées est effectué par une simple translation et, même si cela aboutit à un changement de repère, le système de coordonnées est le même pour toutes les araignées (ce que ne serait pas le cas si des rotations ou des changements d'échelles étaient utilisés).

III.2.2 Inspection d'une analogie

Nous allons maintenant décrire les outils offerts par notre système pour

inspecter une représentation analogique de programme. Par *inspection*, nous entendons les opérations ayant pour finalité d'aider l'utilisateur à voir les différents liens existant entre 1) la représentation graphique et le programme, et 2) l'animation de la visualisation et le comportement pendant une exécution.

Ces outils sont :

- Un outil de navigation graphique, permettant d'effectuer des zooms, des rotations ou des translations sur l'ensemble de la représentation graphique.
- Un éditeur de représentation permettant d'obtenir des informations sur les liens existant entre objet graphique et programme représenté. Cet éditeur est accessible de deux manières :
  1) soit à partir des représentations graphiques elles-mêmes, la sélection d'un objet graphique entraînant l'affichage des informations liées à cet objet particulier (nous avons présenté un exemple de ces informations dans le paragraphe 2.1.2.1),
  2) soit à partir d'un outil spécifique : l'éditeur de représentation que nous allons décrire dans le paragraphe 2.4.2.
- Un outil de contrôle du déroulement de l'exécution du programme pas à pas.
- Un outil proposant des vues multiples sur l'animation d'une représentation.
- Un outil de visualisation du déroulement de l'exécution d'un programme. Ce dernier outil permettant de visualiser les liens entre progression dans l'exécution et animation de la représentation analogique.

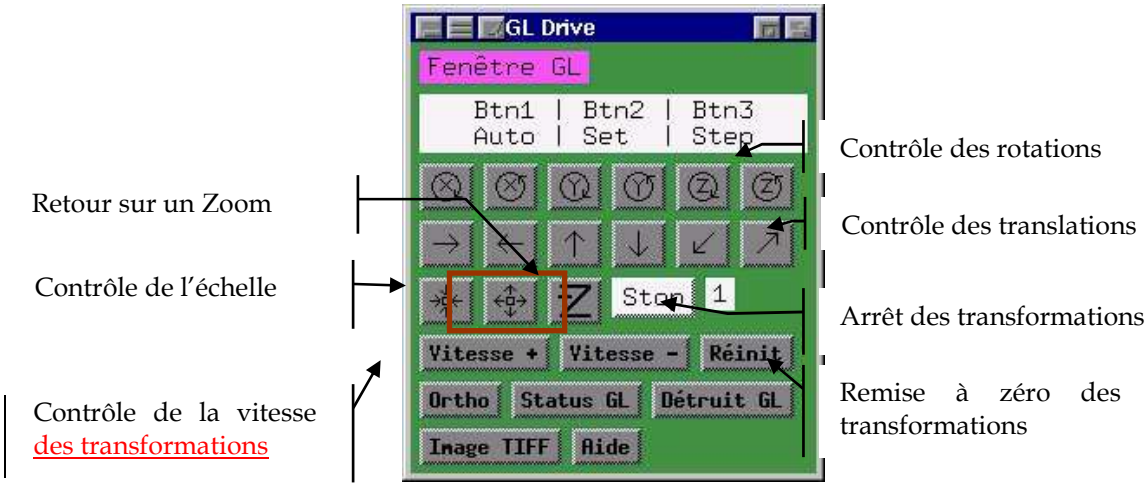


Figure 2.20 Fenêtre de contrôle de la navigation graphique

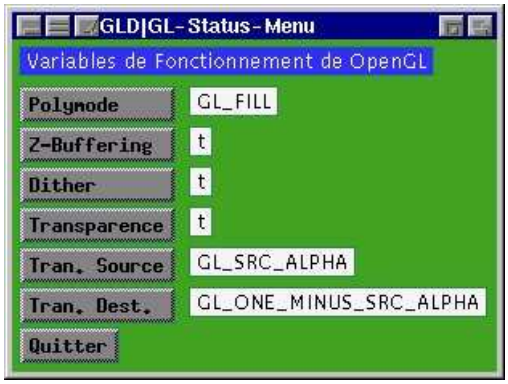


Figure 2.21 Fenêtre de contrôle des modes d'affichage graphique

III.2.2.1 Navigation dans une représentation graphique

L'outil que nous allons décrire à été développé spécialement pour le système Zeugma. Toutefois, il est maintenant inclus comme outil standard dans Xbvl. En effet, il permet de naviguer dans une représentation graphique élaborée en utilisant l'interface avec la librairie graphique OpenGL. Ainsi, les fonctionnalités incluses sont :

- La possibilité d'effectuer des opérations : de translations sur les trois axes, de rotations dans les trois dimensions et des agrandissements ou réductions de l'image dans sa globalité. Ces opérations, effectuées par l'utilisation de boutons de commandes, permettent des modifications continues (bouton de gauche de la souris), pas à pas (bouton de droite de la souris) ou par saisie de valeurs (bouton du milieu). La figure 2.20 présente la fenêtre d'interaction de la navigation et la correspondance entre bouton de commande et fonction graphique. Ces opérations de transformations graphiques peuvent être effectuées par la sélection des boutons de commande correspondants mais aussi par la sélection de la représentation graphique elle-même pour une translation (avec le bouton de gauche) ou pour un zoom sur une partie de l'image (avec le bouton droit de la souris). Le bouton de commande « Retour sur un Zoom » permettra alors de revenir dans les conditions de translations et d'échelle précédent un zoom effectué à l'aide de la souris.
- La possibilité de passer d'une projection orthonormée à une projection en perspective (bouton de commande *ortho*).
- La destruction de la fenêtre d'affichage active (*Kill GL)*.
- La sauvegarde de l'image sélectionnée dans un fichier (*Image TIFF*).

De plus, cet outil donne accès à une fenêtre de contrôle du comportement de la librairie OpenGL (bouton *Status GL*). Cette fenêtre, présentée dans la figure 2.21, offre les contrôles suivants :

- Le mode de dessin des polygones : cette librairie permet en effet de spécifier un dessin en fil de fer (GL_LINE) ou plein (GL_FILL)

pour les polygones. Cette fonctionnalité peut s'avérer nécessaire dans le cas d'images complexes demandant un temps de calcul important en mode plein,

- L'activation du calcul des faces cachées suivant la méthode du Z Buffering[114],
- L'activation du calcul des transparences et de son mode de calcul.

Cette série de contrôles, modifiant de manière importante le rendu de l'image, est principalement utile dans le cas d'images complexes demandant des temps de calculs importants. Ils peuvent de plus être modifiés à tout moment.

Cette première série d'outils n'est pas directement liée à l'étude des liens entre programmes et représentations graphiques. Toutefois, comme par exemple dans le cas d'une représentation analogique de programmes par une cité, il s'avère nécessaire afin de pouvoir en parcourir les différents niveaux par des ajustements des parties visualisées et par des agrandissements successifs sur, par exemple, les différents quartiers ou maisons de la représentation (cf. figure 2.22).

III.2.2.2 Informations sur les analogies et réduction dynamique de la granularité

III.2.2.2.1 Informations sur les liens entre l'image et le programme

La série d'outils (présentés dans la figure 2.23) que nous allons décrire maintenant offre une aide à la lecture des représentations analogiques. Ces outils se basent sur le lien existant entre la condition d'activation d'un ORS et

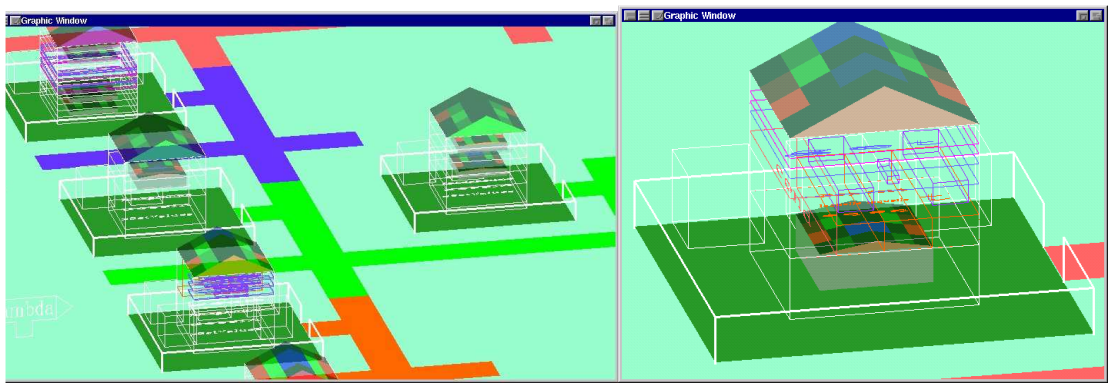


Figure 2.22 Zooms vers une maison

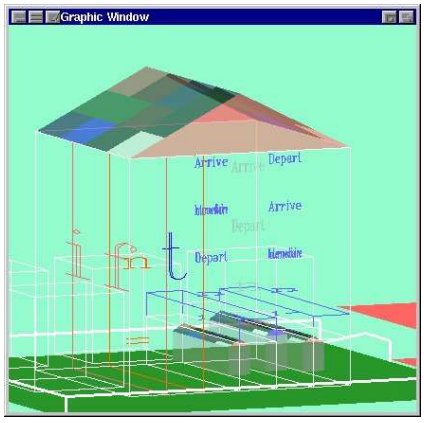
l'objet graphique généré au moment de son activation. De plus, l'activation

[114] Le calcul des faces cachées selon la méthode du Z Buffering consiste en l'affichage des objets graphiques dans un ordre déterminé par leur position sur l'axe des Z (profondeur) : le plus éloigné sera dessiné en premier, l'affichage successif des objets permettant ainsi l'occlusion des faces cachées.

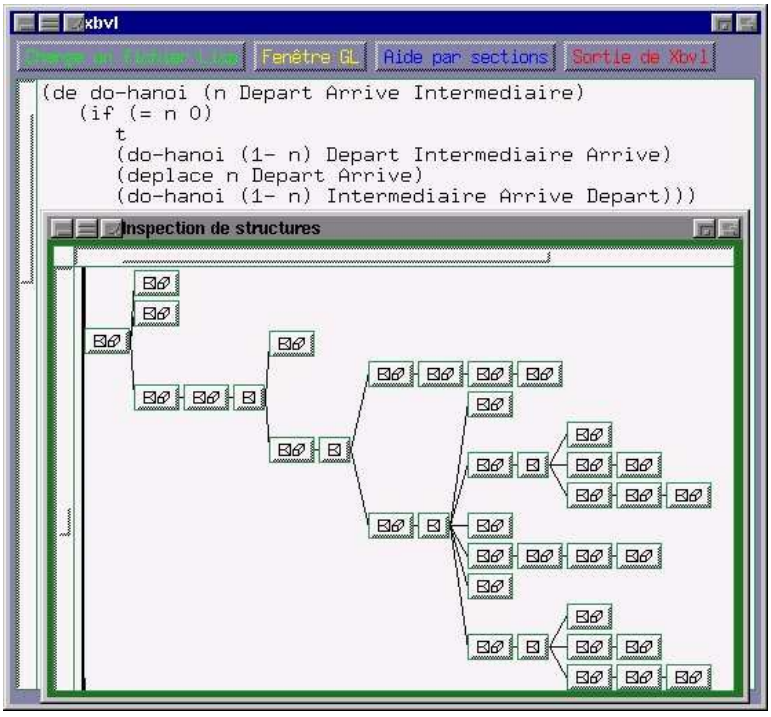
d'un ORS, si elle est conditionnée par la vérification d'une condition particulière, est aussi le résultat d'un parcours d'un ensemble de points de vue sur les programmes par le schème générateur de représentation analogiques. Ainsi, la position, l'orientation, la taille ou la couleur d'un objet est déterminée par l'ensemble des étapes de ce parcours. C'est l'ensemble de ces informations (lien entre un objet graphique et un ORS et lien entre les caractéristiques de cet objet (position, taille, etc.) et le parcours effectué par le schème générateur de représentation analogique) qui seront ainsi fournis à l'utilisateur.

Le premier outil (ou la première fonctionnalité liée à la fenêtre d'affichage des représentations graphiques), évoqué dans le paragraphe 2.2.2.1, consiste à pouvoir obtenir les informations sur les liens entre un objet graphique particulier et un composant du programme représenté. Cet outil permet donc une lecture des liens entre objets graphiques et éléments du programme représenté en partant de la représentation graphique elle-même.

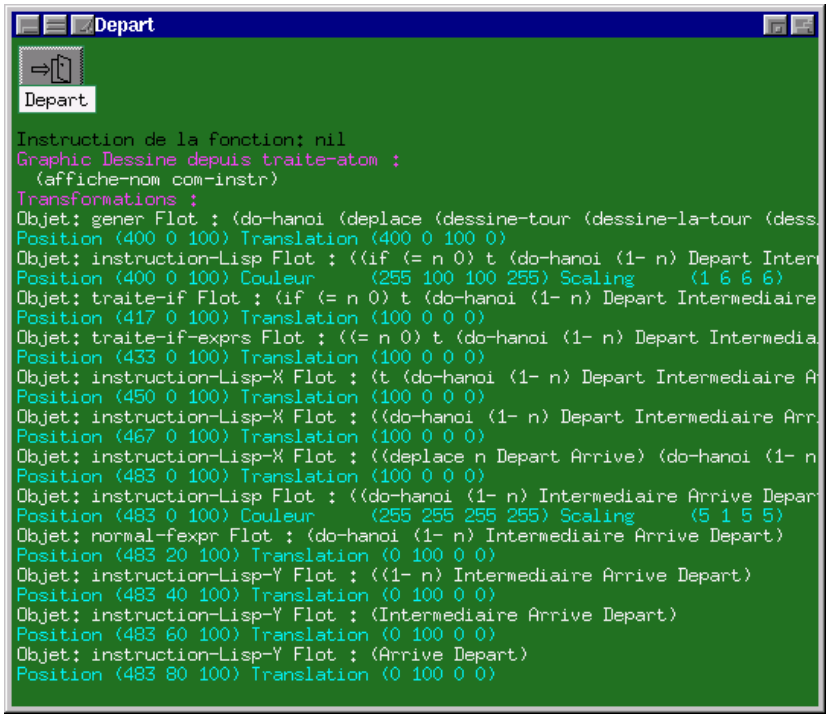
Le second outil, nommé éditeur de représentation, offre la visualisation, sous une forme arborescente, des différentes étapes de la génération de la représentation graphique. Ces étapes – suite de transformations et d'affichage d'objet graphique – formalisent ainsi les opérations graphiques effectuées pendant le parcours du programme par le schème générateur de représentations. La figure 2.23.b montre le code source d'une fonction et l'extrait de l'éditeur de représentation correspondant à sa représentation dans l'analogie *des programmes comme des cités*.



(a) Représentation analogique



(b) Editeur de représentation



(c) Information sur un objet graphique

Figure 2.23 Informations sur une représentation

Cette arborescence comporte trois sortes d'éléments symbolisant :

- la génération d'un objet graphique par un ORS sans qu'une transformation graphique ait eu lieu au moment de son activation,
- l'avènement d'une transformation graphique au moment de l'activation d'un ORS sans qu'un objet graphique ne soit généré,
- la composition d'une transformation graphique et de la génération d'un objet graphique au moment de l'activation d'un ORS.

La figure 2.23 présente en premier lieu l'objet graphique étudié, l'objet du programme auquel il correspond ainsi que la partie de l'éditeur de représentation dans laquelle apparaît l'élément étudié et la fenêtre d'informations sur cet objet.

Figure 2.19 Structure graphique d'une analogie

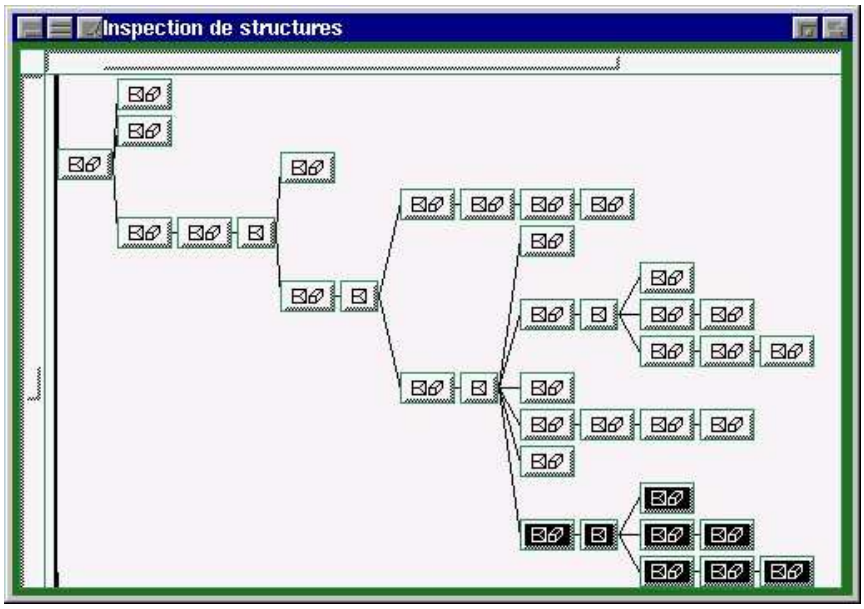
III.2.2.2.2 Réduction de la complexité des représentations

L'éditeur de représentations, outre la visualisation des différentes étapes graphiques de la génération de l'analogie, permet d'effectuer une réduction de la complexité de la représentation graphique. En effet, des représentations analogiques de programmes comme celles que nous avons présen-

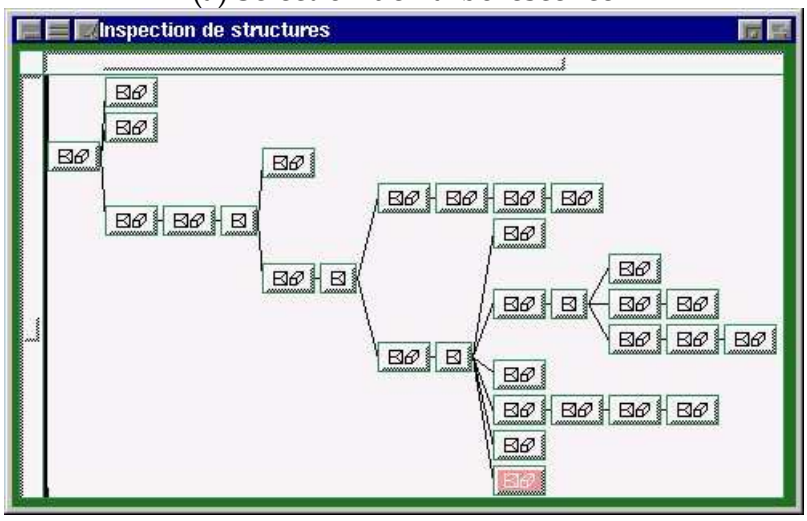
tées peuvent, par le nombre d'objets graphiques en présence ou par leur organisation, aboutir à des images difficiles à lire du fait de leur surcharge.

Pour palier ce problème, outre des fonctionnalités de navigation graphiques qui permettent de se déplacer dans l'image ou d'effectuer des agrandissements sur telle ou telle partie que nous avons présentée précédemment, notre système permet, par l'utilisation de l'éditeur de représentation, d'occulter la représentation graphique d'une partie des objets représentés.

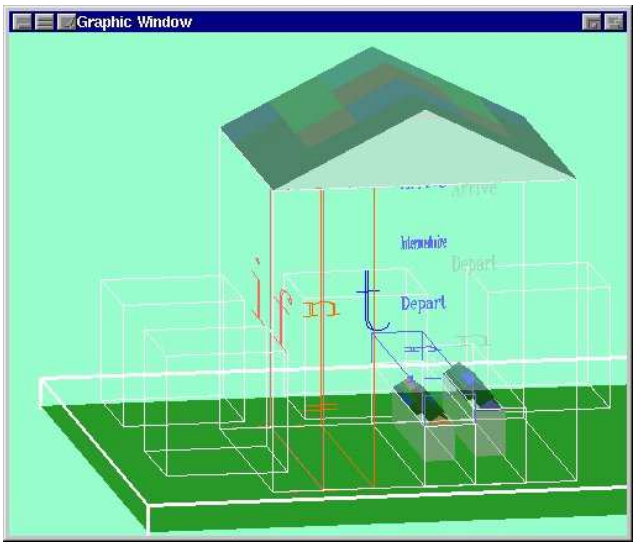
En effet, les objets présents dans l'arborescence de l'éditeur de représentations symbolisent des éléments graphiques de la représentation. Ces éléments peuvent être soit des objets graphiques, soit des transformations graphiques. L'arborescence elle-même symbolise une forme d'encapsulation de ces opérations, ou, pour les transformations graphiques, les éléments graphiques qu'elles influencent.



(a) Sélection de l'arborescence



(b) Arborescence réduite



(c) Représentation graphique réduite

Figure 2.24 : Réduction de la complexité d'une représentation graphique

La réduction d'une représentation se déroule alors en deux étapes :

1) La sélection, par le bouton du milieu de la souris, des éléments graphiques à réduire, par la sélection d'un élément de l'arbre. Cet élément et les éléments présents en dessous de lui dans l'arborescence apparaissent alors d'une couleur différente. Ce sous-arbre montre l'ensemble des éléments graphiques qui seront réduits.
2) Une nouvelle sélection, par le bouton de droite, de ce même élément aboutira à la réduction effective (l'occlusion) des éléments graphiques que l'ensemble des éléments sélectionnés désigne dans l'image de la représentation analogique. L'arborescence disparaît alors et est remplacée par  indiquant la présence d'un sous-arbre réduit.

Ainsi, la figure 2.24.a montre l'arborescence de la figure 2.23 dont une partie est sélectionnée, la figure 2.24.b la même arborescence après réduction et la figure 2.24.c l'image apparaissant après sa réduction.

III.2.2.3 Suivi et contrôle de la visualisation du comportement des programmes

Les outils que nous allons présenter dans cette section visent à aider la compréhension des liens établis entre le comportement d'un programme et l'animation ou la transformation d'une représentation analogique. Ces outils agissent à différents niveaux :

- le contrôle et le suivi de l'exécution du programme par le mode *pas à pas*,
- la visualisation de l'objet graphique animé avec les *points de vue sur*

les animations,

- la visualisation de l'exécution et de ses effets sur une représentation analogique a posteriori avec la *visualisation de l'historique d'une exécution*.

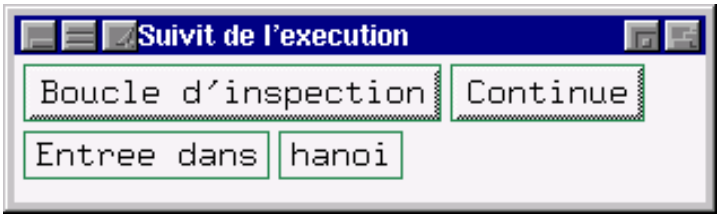


Figure 2.25 Contrôle de l'exécution pas à pas

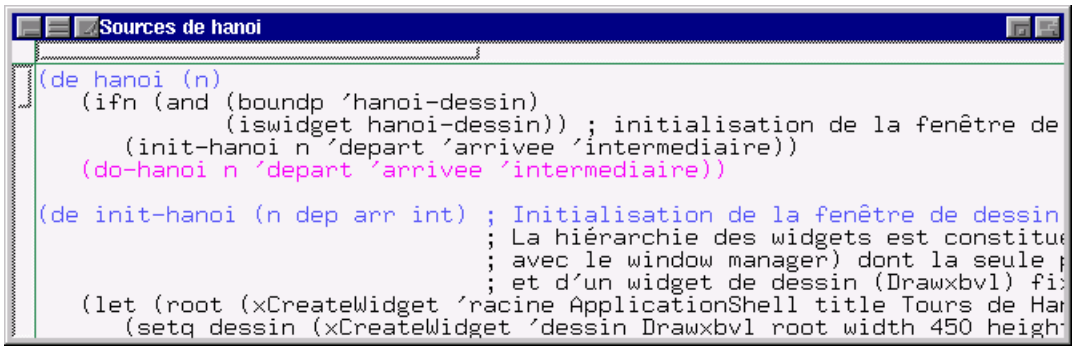


Figure 2.26 Indication de l'exécution d'une expression

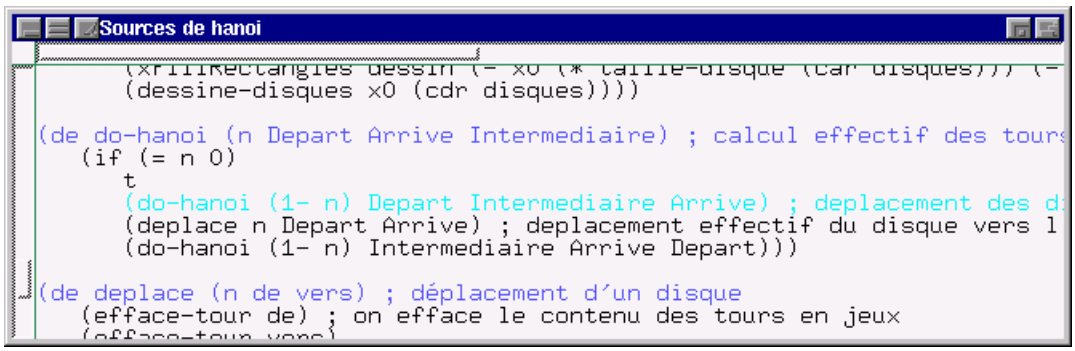


Figure 2.27 Indication de l'exécution d'une expression possédant un attachement

III.2.2.3.1 Suivi pas à pas de l'exécution

Zeugma permet, outre l'attachement de liens entre l'évaluation d'un élément (fonction, variable, expression) d'un programme et une transformation de sa représentation analogique, le positionnement de points d'arrêts. Au lancement de l'évaluation du programme, la fenêtre présentée dans la figure 2.25 s'affiche et montre de manière continue l'élément en cours d'évaluation par Xbvl :

- pour une expression ou une fonction, il est indiqué si l'évaluation de l'objet commence « entrée dans » ou finit « sortie de ». Les expressions en cours d'évaluation apparaissent alors colorées suivant qu'un attachement vers un ORS à été placé (figure 2.27) ou non (figure 2.26).

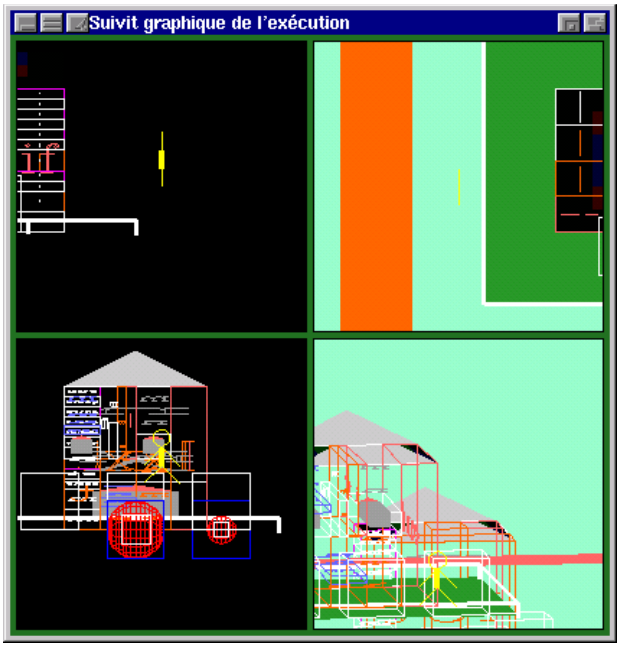


Figure 2.28 : vues multiples de l'évolution de l'exécution

- pour une variable, outre son nom, la valeur contenue au moment de l'accès sera aussi affichée.

Si un élément auquel un point d'arrêt à été spécifié est évalué, le cours de l'exécution est stoppé et l'utilisateur peut alors :

- reprendre l'évaluation jusqu'au point d'arrêt suivant (bouton *Continue*),
- passer dans un mode d'interaction avec Xbvl (« Boucle d'inspection ») dans lequel il pourra, par exemple, modifier la configuration des points d'arrêt liés au programme, ou consulter les états des variables manipulées par celui-ci.

III.2.2.3.2 Points de vue sur les animations

Le mode pas à pas que nous venons de décrire permet un suivi sous une forme textuelle de l'évolution de l'évaluation du programme. Dans le cas

de représentations analogiques complexes mettant en jeu l'animation d'objets graphiques, notre système propose l'utilisation d'un outil de suivi graphique des animations. Cet outil se compose de quatre vues sur l'objet animé. Ces vues montrent l'objet selon les orientations[115] suivantes :

- les trois premières suivent les trois axes de largeur (vue de face), profondeur (vue de coté) et hauteur (vue de dessus) présentées dans les trois premières fenêtres de la figure 2.28.
- la dernière est une vue subjective prenant l'objet animé comme origine et sa destination comme point de fuite.

III.2.2.3.3 Visualisation de l'historique d'une exécution

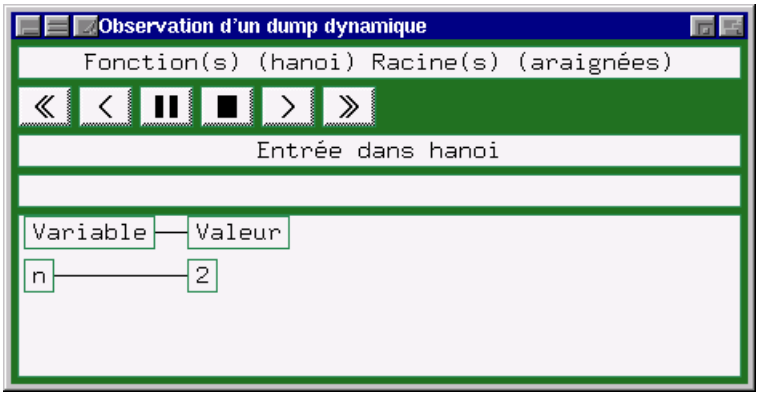


Figure 2.29 Interface de contrôle de la visualisation de l'historique d'une exécution

Le mode pas à pas que nous avons décrit précédemment, ainsi que les points de vue multiples sur les animations, permettent de contrôler et d'observer une exécution pendant son déroulement. Notre système inclut un outil de visualisation de l'historique d'une exécution. Cet outil permet d'examiner, étape par étape, les différentes opérations graphiques générées par l'animation d'une exécution. Se basant sur l'enregistrement de toutes les opérations ayant alors eu lieu, il permet ainsi de naviguer dans celle-ci, de s'arrêter sur tel ou tel point précis ou de revenir en arrière. L'interface de cet outil, présenté dans la figure 2.29, contient :

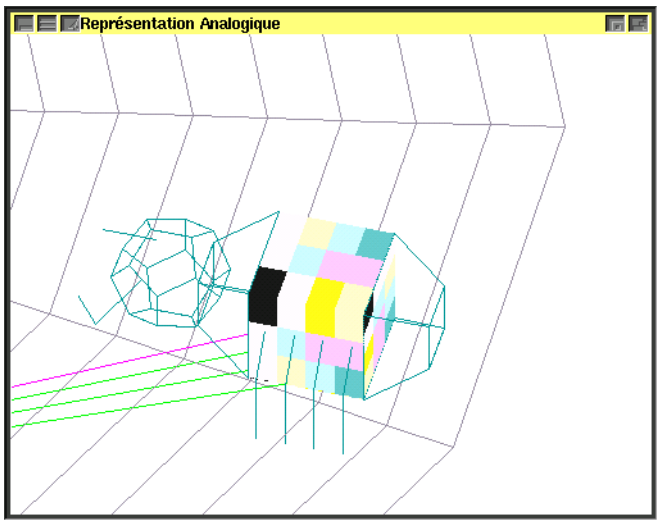
- l'indication du programme observé (*hanoi*) et des ORS racine de la représentation analogique (*araignées*),
- des commandes permettant la navigation dans l'historique,
- l'indication de l'élément du programme actif (*Entrée dans hanoi*),

[115] Les points de vue que nous proposons ici se basent uniquement sur un positionnement de l'œil par des suites de translations et de rotations. Il serait toutefois possible d'utiliser d'autres types de transformations de l'image selon une projection différente (en perspective par exemple) mettant ainsi en exergue d'autres particularités de l'animation.

- la table des couples « variable – valeur » liés au contexte visualisé (*n* est l'unique variable de la fonction *hanoi* et dans l'étape visualisée sa valeur est 2).

A chaque avancée (commande pour une étape ou pour une avancée continue) ou retour (commande pour une étape ou pour un retour continu), la représentation graphique de l'analogie est mise à jour, permettant ainsi de visualiser le lien entre l'évolution de l'exécution et l'animation de la représentation analogique. Dans le cas d'une progression continue, il sera alors possible de l'interrompre temporairement ( ) ou complètement ( ).

La figure 2.31 présente ainsi quatre étapes de la visualisation de l'historique de l'exécution du programme *Hanoi* représenté analogiquement par des araignées sur une toile. Ces étapes montrent quatre entrées successives dans la fonction la fonction *do-hanoi* dont le code source est présenté dans la figure 2.27. Il est ainsi possible de distinguer, comme le montre les zooms sur l'araignée symbolisant la fonction *do-hanoi* présentés dans la figure 2.30, que lors du premier appel la fonction reçoit des données à partir d'une autre fonction du programme. Les images suivantes montrent qu'aucun lien ne part d'une autre araignée vers celle-ci, faisant visuellement apparaître des appels récursifs, ce qui est renforcé par un déplacement vertical de l'araignée. Ce dernier indique, par sa hauteur, le nombre d'appels récursifs successifs ayant eu lieu.



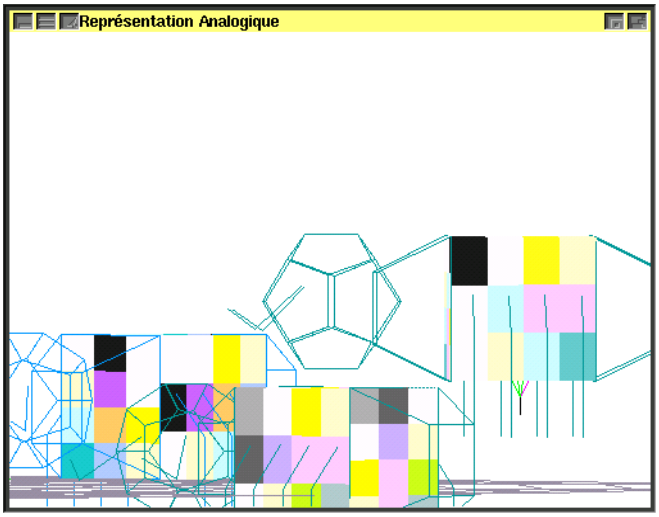
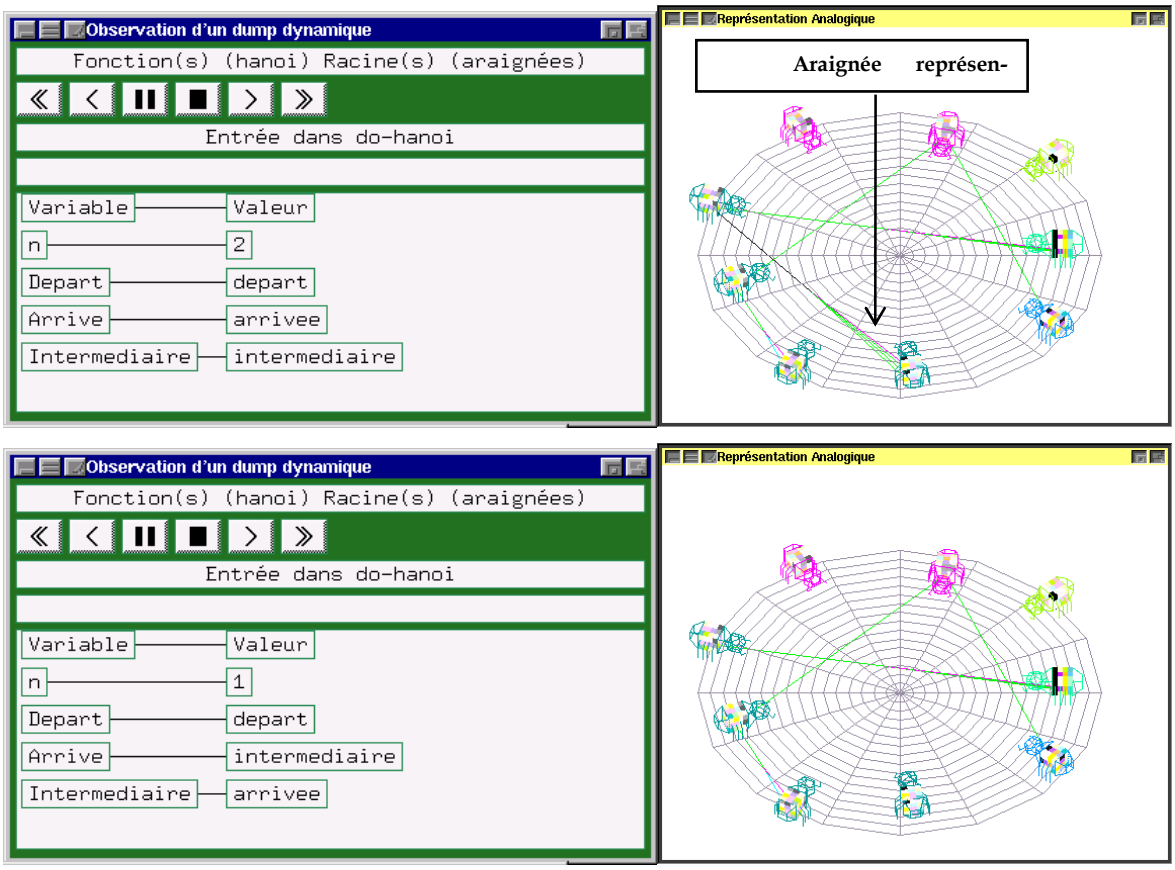


Figure 2.30 Zooms sur l'araignée représentant la fonction *do-hanoi* à la première et la troisième étape de l'exécution



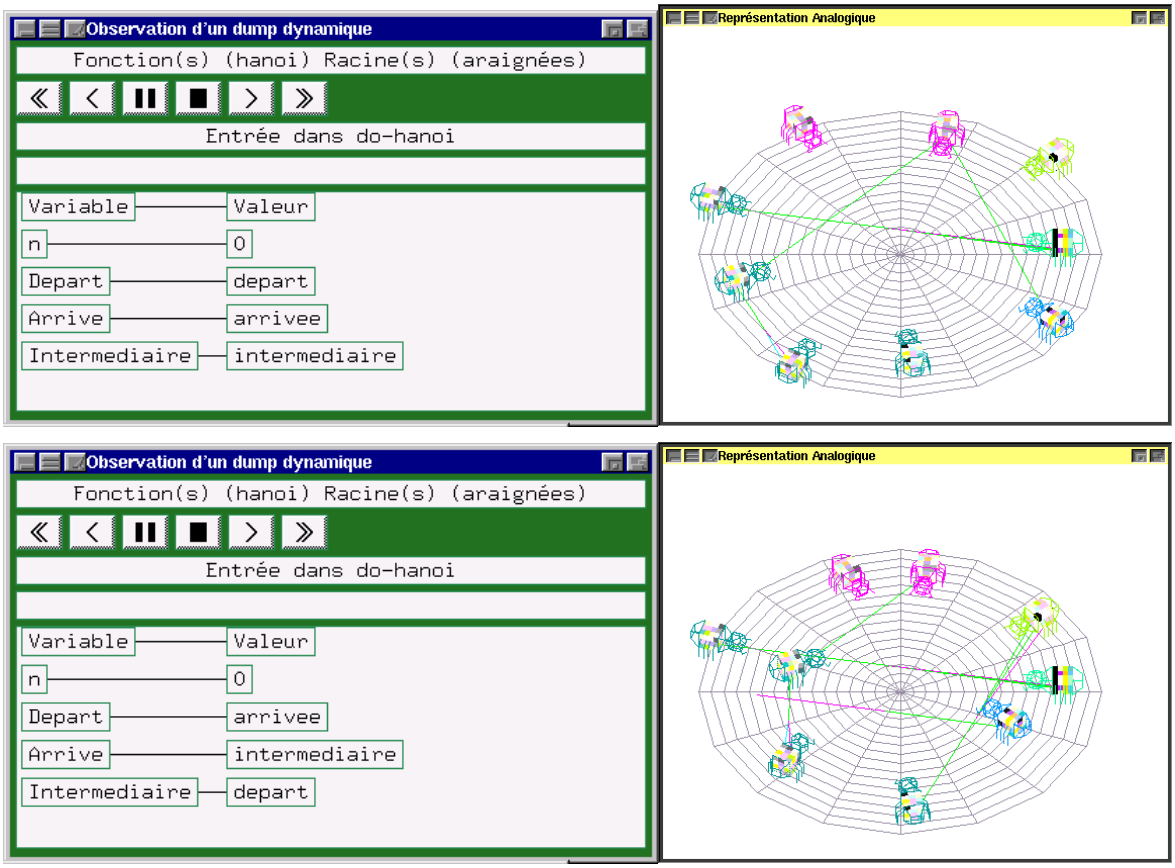
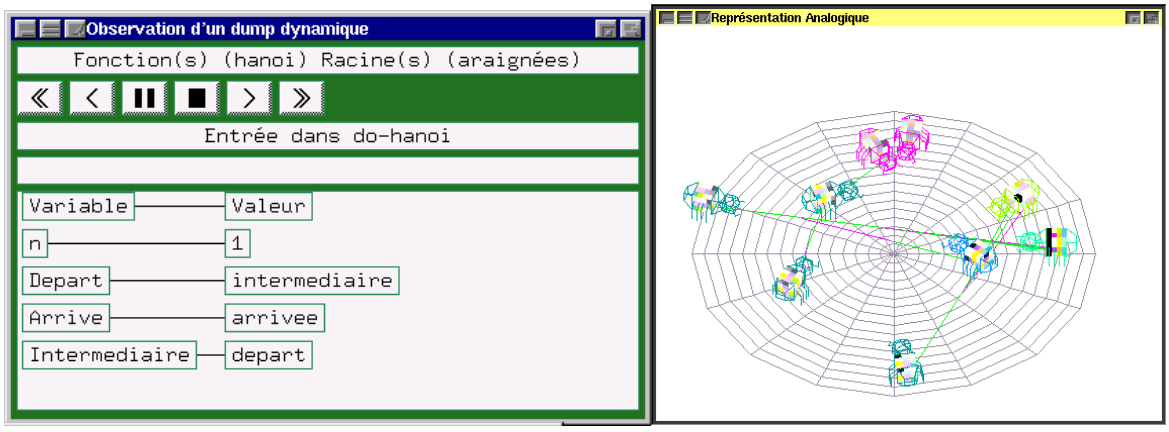


Figure 2.31 Visualisation de l'historique de l'exécution

La figure 2.32 présente les trois appels suivants. Dans la configuration de la position des araignées, un certain nombre d'entre elles ont changé de place par rapport aux étapes présentées dans la figure 2.31. Ces changements de position son le fait du groupe de fonctions utilisées pour construire et mettre à jours la visualisation de l'évolution de la résolution du problème des tours de Hanoi (comme nous l'avons décrit dans la description de la représentation analogique de programmes par des araignées sur une toile dans la partie précédente de ce mémoire).



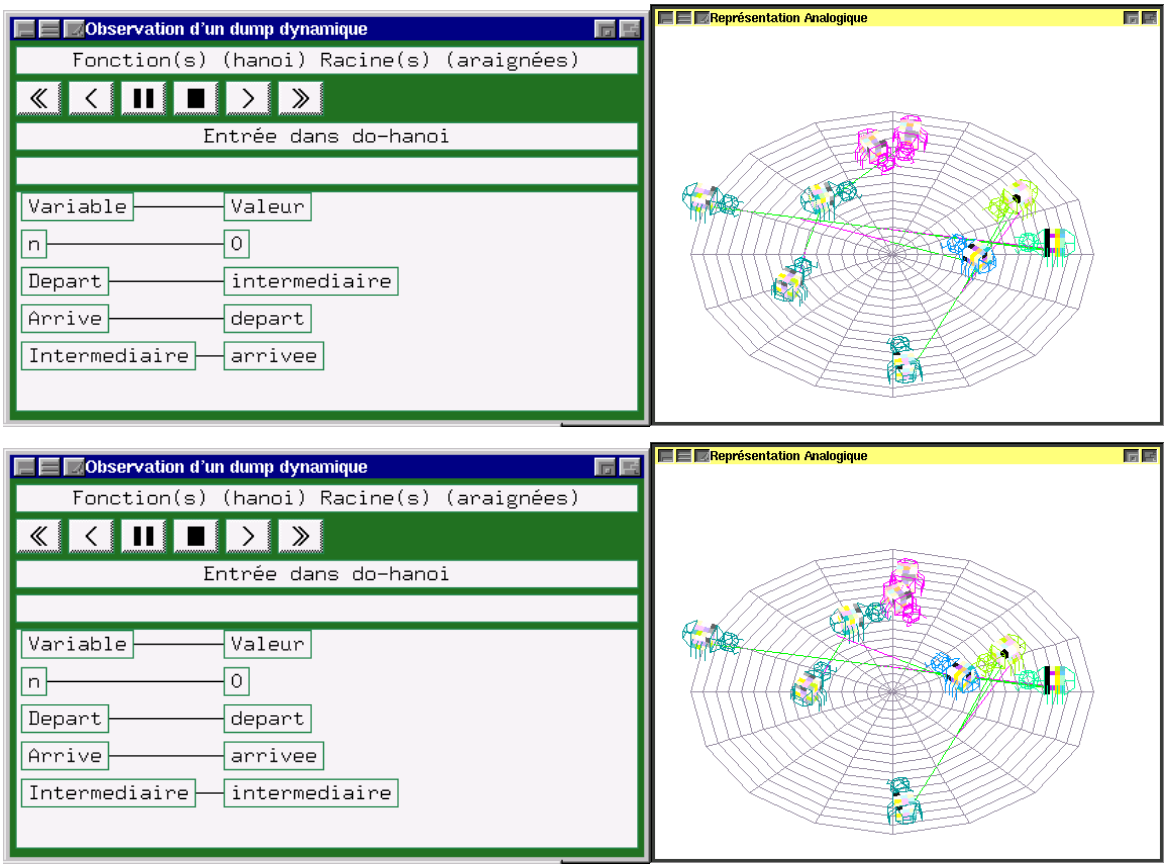


Figure 2.32 Suite de la visualisation de l'historique de l'exécution du programme

Comme nous le montrons dans le quatrième chapitre de ce mémoire, cette visualisation de l'historique d'une exécution est proche du système Zstep95 de Lieberman. Toutefois, notre navigation dans une exécution est guidée par les événements liés à l'animation d'une représentation analogique de celui-ci. Ainsi, elle permet de suivre le déroulement du programme dans un même temps sous sa forme textuelle originale et sous sa forme analogique graphique.

Nous allons maintenant présenter certain points de l'implémentation du système Zeugma. Parmi ceux-ci nous discuterons des informations nécessaires à la mise en œuvre de la visualisation de l'historique d'une exécution de programmes.

Chapitre IV

If indeed, as I believe, cognition and evaluation are the only ways we can deal with the external world, then we would be foolish to discard either. We must use both to obtain the fullest, most penetrating picture we can.

Tsunesaburo Makiguchi

[Makiguchi30, page 65]

IV Comparaison avec d'autres systèmes

Dans ce chapitre, nous allons présenter six systèmes incluant des activités proposées par Zeugma puis nous conclurons par la situation de Zeugma dans le domaine de la visualisation de programmes.

Cette comparaison s'organise suivant les trois domaines abordés par notre système :

1) **La visualisation de programmes :**
   - Le système TPM, permettant la représentation graphique de programmes Prolog (développé à l'Open University),
   - Les systèmes Trip2a - IMAGE, permettant une traduction bi-directionnelle entre données abstraites et données graphiques (développé à l'université de Tokyo).
2) **La visualisation d'algorithmes :**
   - Le système BALSA-II de visualisation d'algorithmes, basé sur la notion « d'événement intéressant » (développé à Brown University),
   - Le système PAVANE, permettant la visualisation d'algorithmes en déclarant des relations entre les programmes et les représentations graphiques (de la Washington University, St Louis, MO).
3) **Les environnements de programmation :**
   - L'environnement de programmation FIELD, intégrant de multiples vues des programmes écrits en C, C++ ou Pascal (développé à la Brown University),
   - Le système Zstep95 de suivi graphique de l'exécution de programmes Lisp (développé au MIT).

IV.1 Visualisation de programmes

IV.1.1 Problématique des systèmes de visualisation de programmes

Les systèmes de visualisation de programmes ont pour finalité de permettre de visualiser sous forme graphique la composition et l'exécution de programmes écrits dans un langage particulier.

Le premier système que nous allons présenter, TPM, est un environnement de programmation qui propose des visualisations de la composition et du comportement de programmes écrits en langage Prolog.

Le second système, Trip, ne se définit pas comme un environnement de programmation appliqué à un langage particulier. Il prend comme base de travail la définition de données abstraites sous forme prédicative et permet d'une part une traduction de ces données vers leur représentation graphique et d'autre part le travail sur ces données à partir de leur représentation graphique. Nous avons choisi de présenter ce système qui travaille sur une abstraction de données et, de ce fait, se rapproche de notre méthode de spécification de la visualisation des programmes. De plus, les auteurs de ce système insistent sur le fait que les données représentées peuvent aisément être issues d'une étude sur les programmes et leur composition.

IV.1.2 Le système TPM

Le système TPM (Transparent Prolog Machine), développé à l'Open University autour de Marc Eisenstadt [Brayshaw88, 91][Eisenstadt87, 88, 89a, 89b, 91, 97b], est en premier lieu un système construit à des fins d'enseignement et de recherche sur le langage Prolog ([Domingue92], [Eisentadt 92a, 92b]). Il est encore utilisé aujourd'hui comme outil d'aide à l'enseignement de ce langage. Cet environnement de programmation définit et donne accès à une représentation graphique des divers aspects de ce langage (offrant une représentation commune à la visualisation statique des programmes et à leur exécution). Dans la section 1.2.1, nous allons détailler les principes sur lesquels ce système se base et dans la section 1.2.2, un exemple de l'utilisation de ce système.

IV.1.2.1 Problématique de TPM

La problématique sous-jacente au système TPM est la mise au point d'une visualisation de programmes Prolog *efficace* dans sa capacité à :

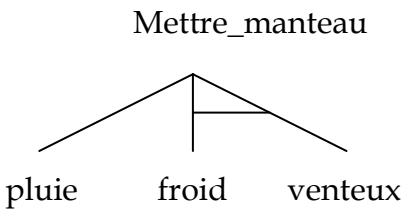
(i) visualiser entièrement l'espace de résolution de Prolog,

(ii) discriminer de manière évidente le processus d'unification de la tête des règles avec la résolution des conditions qu'elle contient.

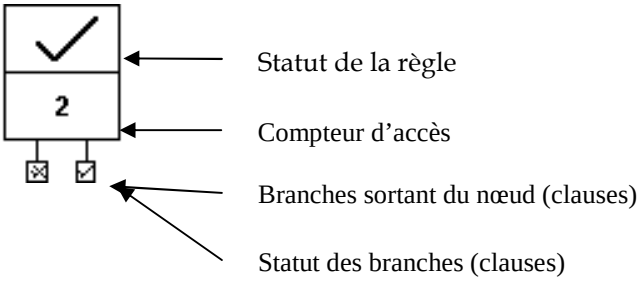
Le problème est que ces deux buts sont contradictoires : le premier s'attache à une possibilité de réduction de la complexité d'une représentation graphique alors que le second est centré sur un accroissement du niveau de détail présent.

Afin de résoudre cette contradiction, TPM propose une notation basée sur deux idées principales :

1) la structure globale de la trace ainsi que les dynamiques en jeu peuvent être visualisées en utilisant des arbres ET/OU animés (cf. Figure 1.1.a présentant un exemple d'un arbre ET/OU). Cette visualisation incluera ainsi la visualisation des processus de *backtrack* et de recherche de solutions alternatives.



(a) Représentation par un arbre ET/OU de
« mettre un manteau s'il pleut OU s'il fait froid ET venteux »



(b) Boîte de représentation du statut de l'exécution d'une règle

Figure 1.1 Eléments graphiques de TPM

2) L'affichage des informations doit pouvoir être modulé dans sa complexité, intégrant ainsi sur une même vue des réductions importantes et des agrandissements du niveau de détail sur un point particulier.

Afin de réaliser ces deux idées, TPM propose une nouvelle représentation des nœuds des arbres ET/OU par la définition de *Boîtes de statut des procédures*[116]. Cette nouvelle représentation est, de notre point de vue, l'innovation principale de ce système. Elle est, comme le montre la figure 1.1.b, composée des éléments suivants :

- l'indication du statut de la règle (en cours d'exécution, inactivée) et le statut de sortie de son exécution (succès, échec). Cette indication est présentée par un symbolisme simple (point d'interrogation pour une résolution en cours ou encore une croix pour exprimer un échec, le symbole présenté dans la figure 1.1 est celui de la réussite de l'exécution).
- Un compteur du nombre d'accès à la règle.
- Des branches correspondant aux différentes clauses de la règle (chaque branche exprime alors une partie OU de l'arbre ET/OU).

Cette visualisation, couplée à des mécanismes de réduction manuelle ou automatique (au cours de la visualisation de l'exécution) de la complexité des arbres ET/OU permet ainsi de répondre aux deux idées fondatrices de TPM. Ces arborescences, des arbres ET/OU avec, comme nœud, les boîtes

[116] Procedure Status Box.

de statut, sont alors nommées diagrammes AORTA (pour AND/OR Tree Augmented) [Eisenstadt89b].

Un autre problème important dans la visualisation de programmes Prolog est celui de composants extra-logiques du langage que sont les coupures[117]. TPM les visualise de la manière suivante[118] :

1) Les clauses *coupées*, et leurs descendants, se voient *gelées* (apparaissent comme prises dans un nuage, ce qui les rend inaccessibles au *backtrack*).
2) Les branches du parent suivant la clause sont elles-mêmes coupées (rendues inaccessibles)
3) La coupure se termine par un succès, comme n'importe quelle clause Prolog ordinaire.

Ce dernier point montre [Eisenstadt97] que la représentation graphique de programmes Prolog par les diagrammes AORTA permet de traiter ce problème délicat de manière simple.

```
% On fait la fête  si l'on est heureux et que c'est notre anniversaire ou
% pour remonter le moral à un ami.

party(X) :- happy(X), birthday(X).
party(X) :- friends(X, Y), sad(Y).
happy(X) :- hot, humid,                 % X est heureux s'il fait chaud et
humide
            !, swimming(X).             % mais seulement si X nage
happy(X) :- cloudy, watching_tv(X).
happy(X) :- cloudy, having_fun(X).
cloudy.             humid.                  hot.
having_fun(tom).    having_fun(sam).
swimming(john).     swimming(sam).
watching_tv(john).
sad(bill).          sad(sam).
birthday(tom).      birthday(sam).
friends(tom, john). friends(tom, sam).

?- party(Name).
Name = sam
```

Figure 1.2 Programme Prolog visualisé dans la figure 1.3

[117] Les coupures permettent, en Prolog, de réduire l'espace de recherche et ainsi de limiter les opérations de *backtrack* à un point déterminé du programme.

[118] TPM considère alors les coupures comme des clauses ayant des ancêtres et des *frères* (ou clauses de la même règle).

IV.1.2.2 Exemple d'interaction avec TPM

La figure 1.2 présente le code source d'un programme exemple construit spécifiquement pour illustrer les possibilités de la représentation graphique des diagrammes AORTA dans TPM par rapport aux coupures.

Ce programme exprime les conditions dans lesquelles *on peut faire la fête*. La visualisation graphique de ce programme par TPM dans la figure 1.3, montre que la clause **happy** s'est en premier lieu conclue par un succès mais qu'elle a causé un échec de la clause **birthday**. Cet échec met alors en route le mécanisme de *backtrack* qui est stoppé par la coupure (comme indiqué par la croix déformée présente dans le statut du nœud). Ce nouvel échec entraîne la mise en route de la seconde règle définissant **party** qui réussit à unifier **friends(tom, john)** mais qui échoue alors sur **sad(john)** provoquant une nouvelle recherche sur le prédicat **sad** aboutissant à un succès.

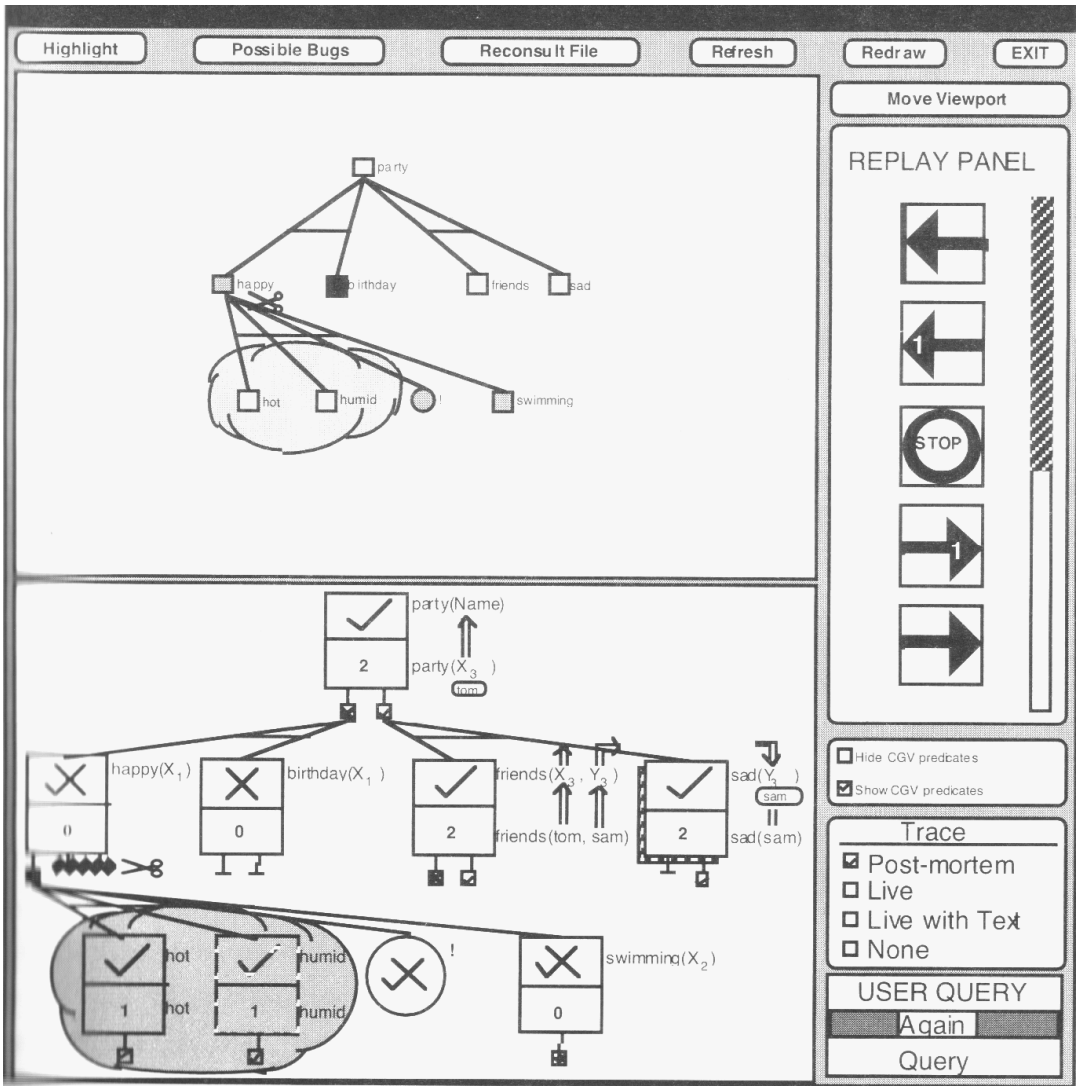


Figure 1.3 Interface du système TPM

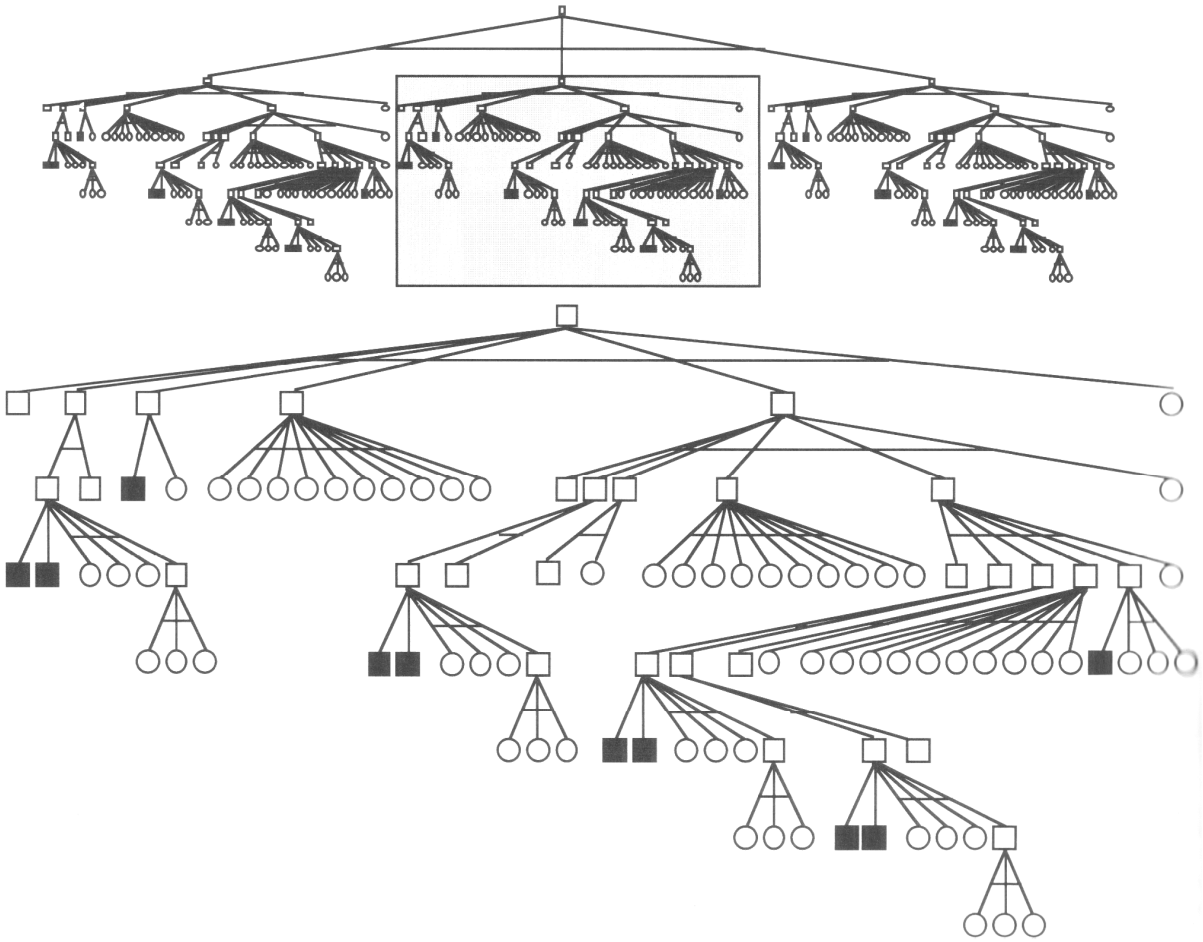


Figure 1.4 Navigation dans un diagramme AORTA avec TPM

IV.1.2.2.1 Outils de parcours et de réduction de la complexité des AORTA

La gestion de la complexité des représentations graphiques, comme nous avons pu le voir dans la section 1 du chapitre III de ce mémoire (page 113), fait partie des enjeux d'importance des systèmes de visualisation de programmes. En effet, le dessin d'outils répondant de manière efficace à cet enjeu permet au système qui les utilise de gérer des programmes de grande taille. Nous allons maintenant présenter des outils, faisant parti de TPM, permettant de résoudre le problème de la réduction de la complexité des diagrammes AORTA complexes ou de faciliter la navigation dans ceux-ci.

Le premier est comparable à une sorte de loupe que l'utilisateur peut déplacer sur le diagramme et qui permet, dans une seconde fenêtre de TPM, de visualiser successivement des parties d'un diagramme de grande taille. La figure 1.4 présente un exemple de l'utilisation de cet outil.

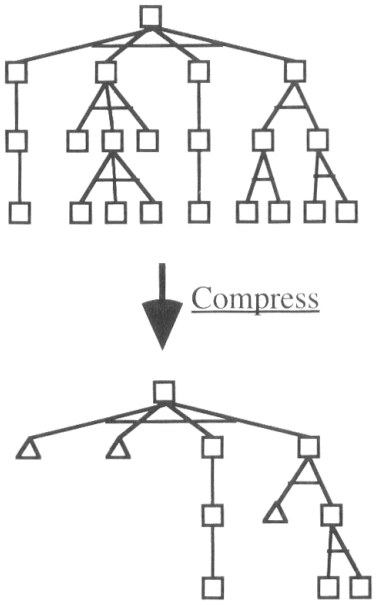


Figure 1.5 Compression d'un arbre

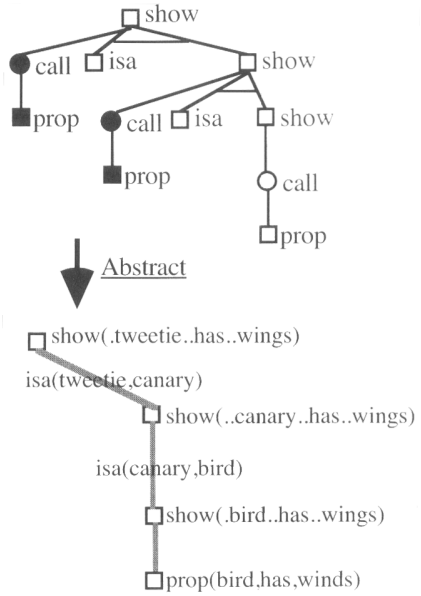


Figure 1.6 Abstraction sur un arbre

La seconde série d'outils que TPM propose permet une gestion interactive de la complexité des diagrammes affichés. Ces outils, intégrés à l'interface présentée dans la figure 1.3, sont :

- la gestion de la granularité en permettant de régler le niveau de détail affiché dans les diagrammes AORTA (la figure 1.3 présente ainsi simultanément les deux vues disponibles).
- L'*échelle* à laquelle va être affiché le diagramme, permettant ainsi une vue globale d'un arbre de grande taille ou une vue rapprochée sur une de ses parties (un exemple de cet outil est présenté dans la figure 1.4).
- La *compression* de parties d'un arbre (figure 1.5), permettant de n'afficher que les parties significatives d'un diagramme et de cacher les autres.
- La définition d'une vue *abstraite* d'un programme. Cette définition (figure 1.6) spécifiée par l'utilisateur, permet d'obtenir une vue d'un programme plus fonctionnelle ou relative au domaine d'application du programme.

IV.1.2.3 Critique de TPM et comparaison avec Zeugma

Le point fort du système TPM, comme nous l'avons dit précédemment, est la définition d'une représentation graphique des statuts de l'exécution des clauses (les *status procedure box*) permettant, tout en utilisant un espace restreint, de visualiser toutes les informations importantes pour comprendre l'exécution d'un programme Prolog. Cette proposition de visualisation contraste, et pas uniquement du fait de l'utilisation de deux ou trois dimensions,

avec celles proposées, par exemple, par le système Cube [Najork91, 92, 93] (la figure 1.7 présente un exemple de visualisation de programme proposé par Cube), dans lequel la visualisation des programmes est plus axée sur l'aspect statique de leur composition et n'inclue pas le même niveau de détail par rapport aux informations relatives à leurs exécutions.

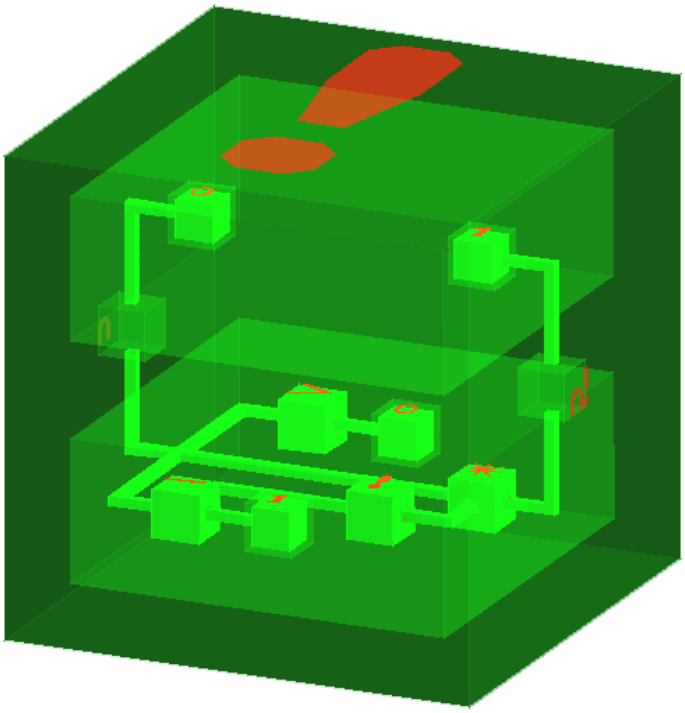


Figure 1.7 Représentation de factoriel par Cube [Najork93]

La différence fondamentale, qui est un choix prédéterminé, entre TPM ou Cube et Zeugma est que notre système permet la visualisation de la composition *et* du comportement des programmes par des graphismes que l'utilisateur du système peut lui-même concevoir. Ainsi, même s'il est certain que l'élaboration de métaphores graphiques efficaces pour représenter le code source et/ou le fonctionnement de programmes appartenant à des langages déterminés est une tâche difficile mais nécessaire, nous pensons qu'il est préférable de permettre à l'utilisateur de construire ses propres représentations. Nous pensons également qu'il est essentiel d'intégrer à un système de visualisation de programmes des outils permettant de contrôler et de visualiser différents points de vues dépendant des caractéristiques particulières que l'utilisateur veut pouvoir *voir* dans les visualisations.

IV.1.3 De Trip à IMAGE

Le système Trip2a [Takahashi94] - IMAGE [Miyashita94], succédant aux systèmes Trip [Kamada89, 91], Trip2 [Takahashi91], et Trip3 [Miyashita92], a pour vocation de permettre une traduction bi-directionnelle entre des données abstraites (*abstract data*)[119] et des données graphiques (*pictorial data*)

[119] Ce système, comme nous le décrivons plus loin, ne prétend pas à s'appliquer à

via la spécification de règles de correspondance[120]. Ce système se situe, à la fois dans le domaine de la visualisation de programmes et d'algorithmes, et dans le cadre plus large de la *visualisation d'informations* et de la traduction des informations graphiques sous forme abstraite (ses auteurs le comparent à des systèmes comme GRAFLOG [Pineda88a, 88b, 88c], une interface de construction d'objets graphiques dans laquelle les images résultantes sont interprétées dans une forme prédicative Prolog).

Ainsi, il ne s'applique pas directement à des programmes mais se base sur une représentation intermédiaire de la structure des données abstraites sous forme de prédicats Prolog et permet, via la spécification des règles de correspondance, la traduction de cette représentation intermédiaire en ensemble d'objets graphiques liés entre eux par des contraintes (les différentes étapes de cette traduction sont le travail des systèmes TRIP et COOL que nous détaillerons dans la section 1.3.1). La traduction dans le sens réciproque (d'une représentation graphique vers une représentation de la structures des données abstraites), est le travail des systèmes Trip2a et IMAGE que nous détaillerons dans la section 1.3.2.

IV.1.3.1 Trip : des données abstraites aux données graphiques

Du fait du grand nombre possible de représentations des données abstraites, comme des données graphiques, le système Trip (définissant la transition entre données abstraites et données graphiques) se base sur des représentations intermédiaires communes aux unes (les données abstraites) et aux autres (les données graphiques). Ces représentations intermédiaires (des prédicats Prolog) indiquent d'une part les données et objets présents et d'autre part les liens les unissant. La figure 1.8 présente le schéma général de la méthode utilisée dans Trip pour traduire les représentations abstraites de données en représentations graphiques.

représenter des données appartenant à un domaine particulier mais prétend permettre la « visualisation de données ». Ainsi, les données « à représenter » sont considérées comme *abstraites*.

[120] Declarative Mapping Rule.

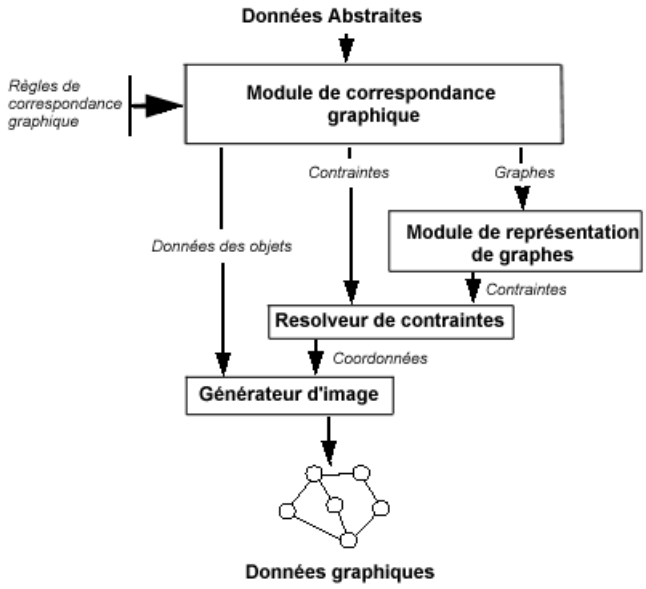


Figure 1.8 Schéma général de la méthode Trip [Takahashi95]

Détaillons les étapes de cette méthode par l'exemple donné pour décrire ce système dans [Takahashi95] :

a) les données abstraites (AD)

Les données abstraites sont, dans le système TRIP, considérées comme étant de la *responsabilité* de l'utilisateur. Toutefois, ce système inclut un analyseur sémantique permettant de traduire directement des *phrases* décrivant des données abstraites en une représentation interne nommée *Abstract Structure Representation* (Représentation Structurelle des Données Abstraites). La phrase donnée en exemple est :

X est formé de p, q et r

b) la représentation structurelle des données abstraites (ADS)

Le passage de la forme initiale des données abstraites dans la représentation intermédiaire utilisable par le système est automatique pour les phrases analysables par le système mais, pour d'autres types de données (comme des programmes par exemple), elle doit être effectuée par l'utilisateur du système[121]. Cette nouvelle forme, prédicative formulée en langage Prolog, sera, pour la phrase donnée en exemple :

```
consiste_en(x, [p, q, r]).
objet(x).
```

[121] En utilisant par exemple des outils comme YACC [Johnson77].

```
objet(p).
objet(q).
objet(r).
```

c) la représentation structurelle des données graphiques (AVS)

Le travail effectué par le système TRIP est la traduction de cette représentation intermédiaire des données dans une nouvelle représentation décrivant les relations structurelles entre les objets graphiques selon des règles de correspondances. Dans l'exemple donné, les règles de correspondances sont :

- Pour le prédicat objet(X) : boxwithlabel(X, Width, Height, X, [visible]) dont les arguments signifient respectivement le nom de l'objet désigné, la taille de la boîte le représentant graphiquement, le label de celle-ci et son mode de dessin (visible signifiant que la boîte est dessinée).
- Pour le prédicat consiste_en(A, [H/L]), l'ensemble des prédicats :
  - horizontallisting([A, H], Xgap, []). (positionnement horizontal)
  - verticallisting([H/L], Ygap, []). (positionnement vertical)
  - multi_connect([A], [H/L], bottom, top, [solid, orthogonal]). (organisation des connexions multiples)

A partir de la représentation intermédiaire des données abstraites, les prédicats générés par le système décrivant la structure de la représentation graphique finale seront :

```
verticallisting([x, [p, q, r]], Ygap, []).
horizontallisting([p, q, r], Xgap, []).
multi_connect([x], [p, q, r], bottom, top, [solid, orthogonal]).
boxwithlabel(x, Width, Height, x, [visible]).
```

d) les données graphiques (AV)

A partir de cette représentation intermédiaire des données graphiques, Trip inclus le système COOL [Kamada89] de résolution de contraintes graphiques permettant la génération de la représentation graphique finale. A ce jour [Takahashi95], ces relations graphiques solvables par COOL définissent plusieurs organisations spatiales en 2 et en 3 dimensions principalement basées sur le dessin de graphe. De plus, les systèmes IMAGE et Trip3 intègrent des outils de définition de nouvelles relations structurelles graphiques selon une méthode inspirée de la *Programmation par l'exemple* : un éditeur graphique permet de dessiner des exemples de ces nouvelles relations que le système intégrera alors. La figure 1.9 présente ainsi la représentation graphique finale pour l'exemple donné.

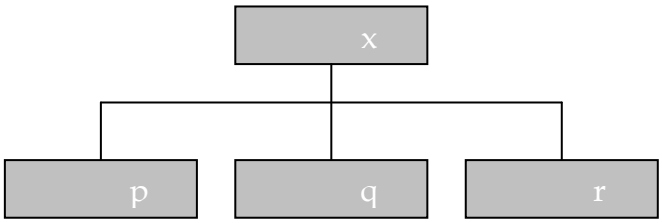


Figure 1.9 Représentation graphique de « x consiste en p, q et r » par Trip

Ainsi, le système Trip permet d'une part d'appliquer des règles de correspondance graphique entre des formes prédicatives décrivant des données abstraites et des formes prédicatives décrivant l'organisation d'objets graphiques. D'autre part, il intègre un resolveur de contrainte permettant de générer une représentation graphique à partir de définitions d'objets graphiques et de formes prédicatives décrivant leur organisation spatiale.

IV.1.3.2 Traduction inverse : des données graphiques aux données abstraites

Le développement de Trip a conduit ses auteurs au système Trip2 et à ses successeurs intégrant la traduction en sens inverse : des données graphiques aux données abstraites. Cette traduction suit une méthode comparable à celle utilisée pour passer des données abstraites aux données graphiques même si des règles de conversions doivent de nouveau être spécifiées pour passer de la représentation structurelle des données graphiques à la représentation structurelle des données abstraites.

Toutefois, cette nouvelle démarche permet à ces systèmes d'intégrer la double compétence de représentation graphique de données et de modification des données par la manipulation d'objets graphiques. Ainsi, la figure 1.10 présente le schéma général de la méthode intégrant cette traduction bidirectionnelle entre données abstraites et données graphiques.

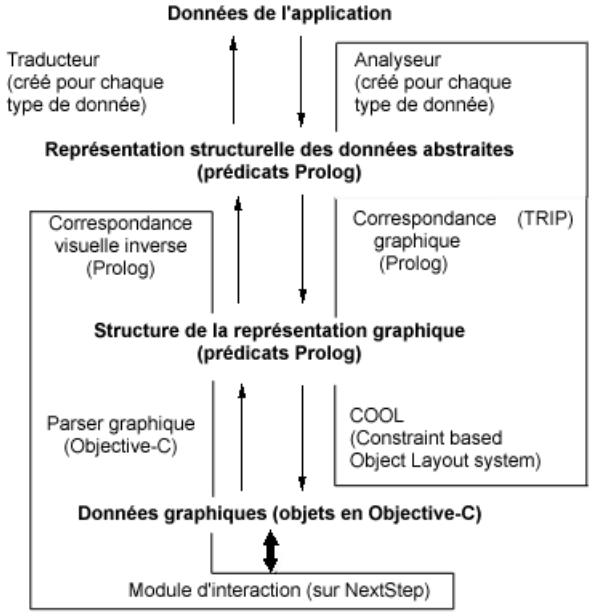


Figure 1.10 Schéma général de Trip2

IV.1.3.3 Critique de TRIP2 et comparaison avec Zeugma

La force principale des systèmes issus de la méthode développée avec le système Trip est la conservation d'une représentation interne cohérente (des prédicats Prolog) pour les données représentées (dénommées données abstraites) et pour les représentations. C'est cette cohérence qui permet en fait cette bi-directionnalité entre les représentations. D'où, Trip2 serait un outil tout aussi performant pour l'élaboration de représentation de données que pour la manipulation de données via un média basé sur le graphisme.

Mais cette force peut aussi être la principale faiblesse de ce système comme le sous-entendent ses auteurs en ne prévoyant pas des traductions automatiques entre données *concrètes*[122] et la représentation interne de la structure des données abstraites. De plus, le système ne propose pas la traduction automatique des règles de correspondances entre la représentation interne de la structure des données abstraites et graphiques, et les règles de correspondance visuelle inverse permettant de passer des données graphiques aux données abstraites.

Ce système pourrait aussi être comparé à des systèmes de visualisations de données comme AutoVisual [Beshers93] qui permettent une abstraction dans la complexité des données visualisées. Toutefois, le point de comparaison avec Zeugma, qui justifie la présence de Trip dans cette section, est la procédure par étapes dans l'élaboration des représentations. Ainsi, la méthode de Trip :

[122] Nous utilisons ici le terme *concrète* pour désigner des données liées à un contexte particulier comme un programme appartenant à un langage donné ou des données structurées comme, par exemple, celles décrivant un système de fichier.

$$AD \Leftrightarrow ADS \Leftrightarrow AVS \Leftrightarrow AV$$

peut être mise en correspondance avec la méthode utilisée par Zeugma :

Programmes ⇒ Abstraction sur les programmes ⇒ Structure de la représentation graphique ⇒ Représentation graphique

La force de la représentation interne prédicative réside dans la possibilité de bidirectionnalité du modèle : d'une part, pouvoir générer une nouvelle représentation graphique après modification des données abstraites, et d'autre part, générer de nouvelles données abstraites après modification des représentations graphiques. Dans le contexte de Zeugma, cette partie d'édition graphique – et donc de modification de graphique des programmes - n'est actuellement qu'en phase de construction. Soit dit *en passant*, cette extension imposera à l'utilisateur des tâches supplémentaires dans la définition des *points de vues* sur les programmes[123]. En effet, pour pouvoir construire une bidirectionnalité, il sera nécessaire de fournir une méthode inverse permettant de reconstruire les programmes à partir de données abstraites caractéristiques d'un point de vue.

IV.2 Visualisation d'algorithme

IV.2.1 Problématique des systèmes de visualisation d'algorithmes

Les systèmes de visualisation d'algorithmes ont pour finalité de permettre la spécification, la visualisation et l'animation de représentations graphiques d'algorithmes. De ce fait, ces systèmes ne prennent pas en considération le langage de programmation utilisé, même si ce choix affecte le système résultant. La distinction principale entre les différents systèmes réside dans la méthode utilisée pour lier le programme implémentant un algorithme choisi et la représentation graphique de celui-ci. Nous avons choisi de présenter deux systèmes représentatifs des méthodes utilisées aujourd'hui à cette fin :

- le système BALSA-II, successeur de BALSA, par lequel fut introduit la notion *d'événement intéressant*,
- le système PAVANE utilisant la méthode de *Visualisation déclarée*.

Dans notre présentation de ces systèmes, nous détaillerons d'abord la méthode qu'ils utilisent puis, par la présentation d'exemples, les différentes

[123] Comme nous l'avons décrit, la définition d'un point de vue sur les programmes demande uniquement, dans Zeugma, une méthode permettant d'abstraire les données spécifiant un point de vue à partir des programmes (cf. page 115).

fonctionnalités qu'ils proposent.

IV.2.2 Le système BALSA-II

Le système BALSA-II [Brown88a, 88b] est développé au System Research Center de Digital Equipment par Marc Brown et Robert Sedgewick à partir du système BALSA [Brown84, 85], et suivi par la même équipe de recherche que celle du système Zeus [Brown91]. Nous avons choisi de présenter Balsa-II car les outils qu'il propose (et que nous décrirons dans la section 2.2.2) illustrent parfaitement la notion d'*événements intéressants* (décrite dans la section 2.2.1), proposée par Marc Brown comme base de construction des visualisations d'algorithmes.

IV.2.2.1 La notion d' « Evénements Intéressants »[124]

Dans sa présentation de la notion « d'événement intéressant » [Brown97c], Marc Brown pose la problématique suivante : « *Most* program visualizations *can be created automatically, whereas most* algorithm visualizations *cannot.* »[125] pointant ainsi la distinction nécessaire entre les étapes de la résolution d'un algorithme et les différentes instructions spécifiques à un langage utilisées pour l'implémenter. En effet, l'étude des étapes d'un algorithme se focalisera sur la signification de chacune par rapport à la résolution d'un problème alors que l'étude de la composition des programmes étudie généralement la manière dont sont codifiées les étapes. Pour illustrer ce point de vue, on peut utiliser la métaphore de la distinction, pour un même texte, entre étude sémantique d'une part, syntaxique et grammaticale d'autre part.

Marc Brown soutient que seule une personne ayant une connaissance approfondie d'un algorithme, et de son implémentation, est à même d'indiquer ce qu'il effectue. En effet, dans la plupart des cas, la génération automatique de la visualisation d'un algorithme demanderait soit que :

- les structures de données utilisées et leur évolution pendant l'exécution du programme implémentant l'algorithme décrivent parfaitement les opérations effectuées (ce qui permettrait ainsi l'approche de *Visualisation Déclarée* présentée dans la section suivante de ce chapitre),
- l'algorithme soit implémenté de manière procédurale, chaque procédure indiquant une étape de l'algorithme (qui est principalement le choix effectué dans Zeugma).

La première condition pose problème : il faudrait pouvoir détecter automatiquement le type d'opération effectué à partir des variations des don-

[124] Interisting Events.

[125] La plupart des *visualisations de programmes* peuvent être créées automatiquement, alors que la plupart des *visualisations d'algorithmes* ne le peuvent pas.

nées manipulées. La seconde condition pose également problème : les algorithmes sont rarement proposés sous une telle forme procédurale. En fait, la visualisation automatique d'un algorithme demanderait une réécriture complète de celui-ci.

La solution à ces problèmes est l'annotation des algorithmes avec des *Evénements Intéressants*[126] : ils seraient transmis au système construisant l'affichage des visualisations. L'avantage de cette méthode est qu'elle ne demande pas la réécriture de l'algorithme mais juste son augmentation. De plus, elle permet :

- une mise au point plus aisée des représentations (qui peuvent être testées par une série d'événements arbitraires),
- une utilisation multiple des représentations qui, indépendantes de l'implémentation d'un algorithme, peuvent alors être utilisées pour d'autres algorithmes,
- l'indépendance du langage de programmation puisque le mécanisme d'augmentation avec la génération d'événements intéressants peut être aisément implémenté quelque soit le langage,
- la constitution d'un historique de l'exécution d'un algorithme par la sauvegarde de tous les événements intéressants générés,
- la possibilité d'utiliser les événements intéressants pour contrôler le déroulement de l'animation des algorithmes, permettant ainsi une décomposition visuelle des opérations ; pour synchroniser les visualisations multiples du déroulement d'un algorithme ; ou encore comme élément de mesure du temps pris par un algorithme pour effectuer un calcul.

Pour illustrer cette notion d'événement intéressant, nous allons décrire l'exemple de la spécification de l'animation d'un QuickSort implémenté suivant le programme présenté dans la figure 2.1

[126] Qui ressemblent étrangement aux méthodes *changed* et *update* de la trilogie Modèle-Vue-Contrôleur chère aux Smalltalkiens [Kay76][Golberg83].

```
1   Procedure QuickSort(l, r : integer) ;
2   var … ;
3   begin
4   if r-l > 0 then
5     begin
6     v := a[r] ; i := l-1 ; j := r ;
7     repeat
8       repeat i := i+1 until a[i] >= v ;
9       repeat j := j-1 until a[j] <= v ;
10      t := a[i] ; a[i] := a[j] ; a[j] := t ;
11    until j <= i ;
12    a[j] := a[i] ; a[i] := a[r] ; a[r] := t ;
13    QuickSort(l, i-1) ;
14    QuickSort(i+1, r) ;
15    end
16  end
```

Figure 2.1 Une implémentation de QuickSort.

Pour animer cet algorithme, Brown propose d'ajouter les quatre instructions supplémentaires (indiquant des événements intéressants) :

```
InterestingEvent(Exchange, i, j)
```

après la ligne 10. Afin d'indiquer l'échange de deux éléments et

```
InterestingEvent(Exchange, i, j)
InterestingEvent(Exchange, i, r)
InterestingEvent(ElementInPlace, i)
```

venant se placer après la ligne 12, permettant de spécifier l'échange des éléments et le fait qu'un élément n'ait pas changé de place. Ainsi, l'ajout de seulement quatre lignes supplémentaires permet de capturer les événements décrivant l'algorithme.

Dans la section suivante, nous allons décrire les différentes utilisations des événements intéressants proposées par le système Balsa-II.

IV.2.2.2 Présentation de Balsa-II

Balsa-II permet l'utilisation des événements ajoutés aux algorithmes afin d'effectuer une série de contrôles et de mesures sur son déroulement :

- La spécification des données nécessaires au déroulement de l'algorithme à partir de sa représentation graphique. Un exemple de cette fonctionnalité est présentée dans la figure 2.2, visualisation du parcours en profondeur d'un graphe, où le nœud initialisant l'algorithme est sélectionné à partir de la représentation.

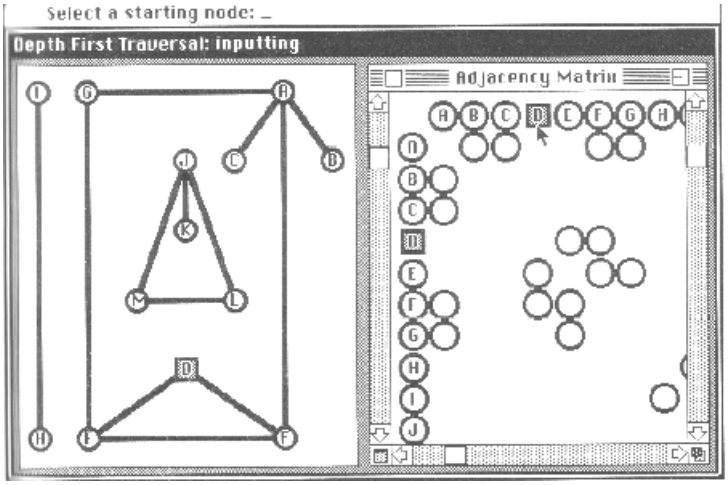


Figure 2.2 Initialisation graphique d'un algorithme.

- La visualisation du compte et de l'utilisation comme points d'arrêt des événements intéressants. La figure 2.3 présente la fenêtre d'interaction de Balsa-II qui permet de visualiser le nombre de fois qu'un événement à été atteint (colonne de gauche), le nombre de fois que l'algorithme s'est arrêté (colonne du milieu) et si un point d'arrêt à été spécifié sur son avènement (colonne de droite).

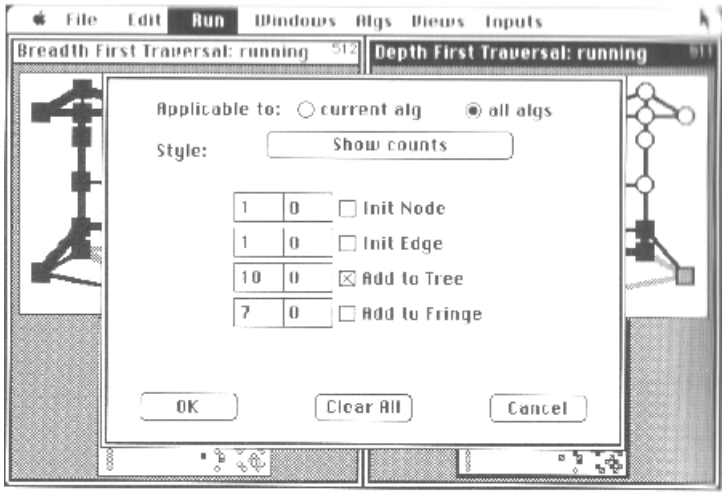


Figure 2.3 Contrôle de l'exécution d'un algorithme.

- La visualisation simultanée de deux algorithmes traitant des même données. Ainsi, la figure 2.4 présente simultanément le parcours en profondeur et en largeur sur les mêmes données.

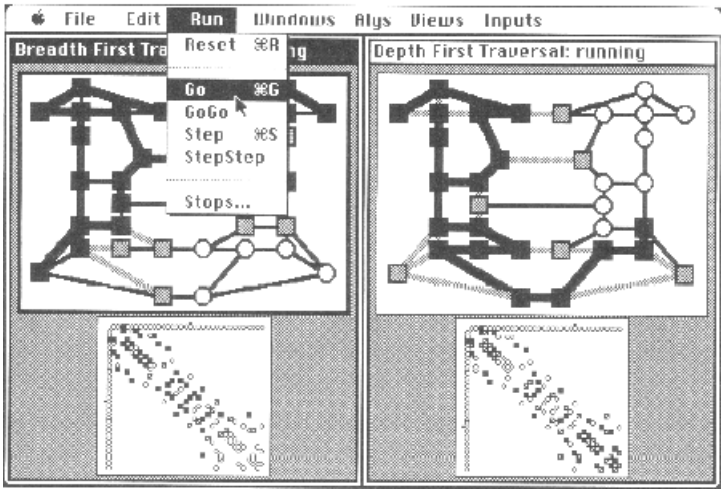


Figure 2.4 Comparaison graphique de deux algorithmes.

- L'indication du coût des opérations de l'algorithme par la visualisation du compte des événements. Ainsi, la figure 2.5 présente le nombre d'occurrences de chaque événement (colonne de gauche), ainsi que le temps attribué à ceux-ci (colonne de droite). Cette fenêtre est affichée lorsque le bouton de commande *Show Count* de la figure 2.3 est sélectionné.

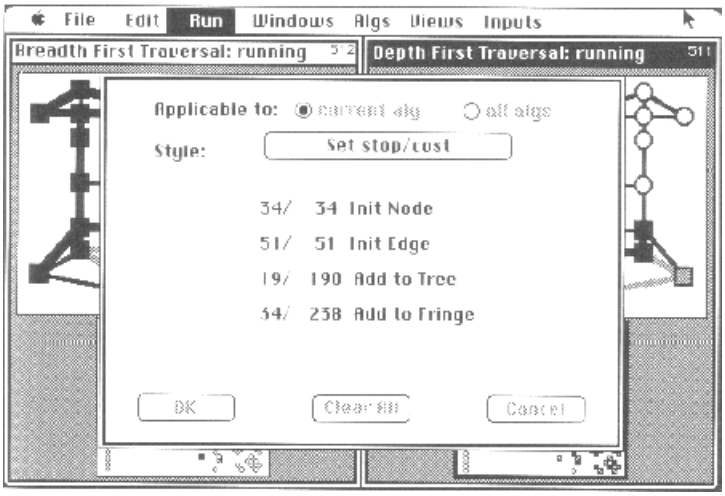


Figure 2.5 Calcul du coût d'un algorithme.

- la synchronisation de multiples vues du déroulement d'un algorithme. Dans la figure 2.6, le nombre présent en haut à droite de chaque visualisation (*1102*) indique le *temps* de déroulement de chaque exécution de l'algorithme.

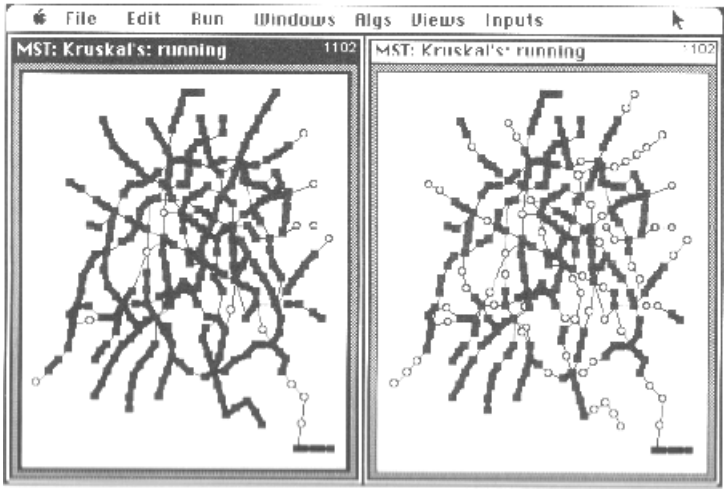


Figure 2.6 Synchronisation d'algorithmes.

Le système Balsa-II utilise donc les événements intéressants dans des activités qui sont habituellement présentes dans les environnements de programmation liés à des langages particuliers, ce qui permet, en fait, de superposer à une considération des programmes comme série d'étapes issues d'un algorithme, une vision des programmes comme une série d'instructions ou d'expressions appartenant à un langage de programmation particulier.

IV.2.3 Le système Pavane

Le système PAVANE, développé au département d'informatique de la Washington University, St Louis MO, sous la direction de Gruia-Catalin Roman, a pour vocation la construction d'animations d'algorithmes. La principale différence avec BALSA-II, que nous allons détailler dans la section 2.3.1, est la notion de *Visualisation Déclarée*[127] mettant en relation non plus des événements spécifiés par l'utilisateur mais la déclaration de liens entre le programme et la visualisation.

IV.2.3.1 La notion de « Visualisation Déclarée »

Dans [Roman89] et [Cox92a, b], les auteurs du système PAVANE définissent la visualisation d'algorithmes comme la mise en correspondance[128] des programmes avec des visualisations graphiques. Cette mise en correspondance consiste pour eux en la *déclaration* de transformations entre des états de l'exécution du programme et la représentation graphique finale. Dans PAVANE, les transformations sont exprimées sous la forme d'une série de correspondances entre espaces (ensemble de *tuples*), correspondances composées d'une ou plusieurs règles spécifiants la relation entre les deux espaces.

[127] Decalarive Visualization.

[128] Mapping.

Par exemple, la déclaration suivante (répondant à la syntaxe PAVANE) :

```
id, ack, p, pp, dist :
  process(id, active, ack, p), vertex(p, pp, dist)
⇒        vertex(id, p, dist + 1)
```

se lira :

> « Pour chaque instanciation des valeurs (*id, ack, p, pp, dist*) dans l'espace d'entrée (correspondant au programme représenté), si le prédicat process(id, active, ack, p) peut être trouvé dans l'espace d'état et vertex(p, pp, dist) peut être trouvé dans l'espace de preuve précédent, un *tuple* vertex(d, p, dist + 1) sera ajouté dans l'espace de des objets. »

Les déclarations mettent ainsi en jeu les niveaux (espaces) suivants :

1) *l'espace d'entrée* : cet espace correspond à l'ensemble des données manipulées par le programme décrivant l'algorithme.
2) *l'espace d'état* : un espace intermédiaire dans lequel des relations peuvent être instanciées entre des éléments de l'espace d'entrée.
3) *l'espace de preuve*[129] : l'espace contenant les relations définissant les objets (graphiques) construits au cours de l'animation de l'algorithme.
4) *l'espace d'objet* : l'espace des nouveaux objets graphiques dont les instances passeront ultérieurement dans l'espace de preuve.

Cette déclaration met en jeu la représentation de l'activation d'un processus (désignés par un identificateur *id*) par l'affichage d'un vertex (ligne) le reliant à son processus père (désigné par un identificateur *p*).

IV.2.3.2 Présentation de PAVANE

La déclaration présentée dans le paragraphe précédent se réfère à la première version de PAVANE [Cox92a] implémentée afin de permettre la visualisation d'algorithmes distribués[130] programmés en utilisant le langage Swarm[Roman90] (un modèle computationel basé sur UNITY[Chandy88] et sa méthode de preuve logique) en ajoutant des méthodes d'accès au contenu des données, aux expressions dynamiques et à une synchronisation dynamique partielle. Cette première version de PAVANE était composée de trois modules :

[129] proof space.

[130] Concurrent.

1) SwarmParse : le compilateur de PAVANE dont la fonction est de traduire les programmes et la définition des visualisations dans une forme exécutable.
2) SwarmExec : exécutant les programmes Swarm compilés et les règles de visualisation en produisant des traces décrivant les animations.
3) SwarmView : lisant les traces décrivant les animations et construisant les représentations graphiques finales.

Une seconde version du système [Cox92b] définit alors PAVANE selon le schéma présenté dans la figure 2.7. Cette nouvelle version permet dès lors la spécification de visualisation pour des programmes écrits en langage C. Le passage de Swarm à C implique cette fois un certain nombre de modifications inhérentes au passage d'un mode interprété à un mode compilé :

- impossibilité de modifier les règles de visualisation « à la volée » puisqu'elles sont maintenant traduites par deux compilateurs (*sc* et *cc*),
- augmentation significative de la rapidité (d'un facteur 100),
- accroissement de la portabilité.

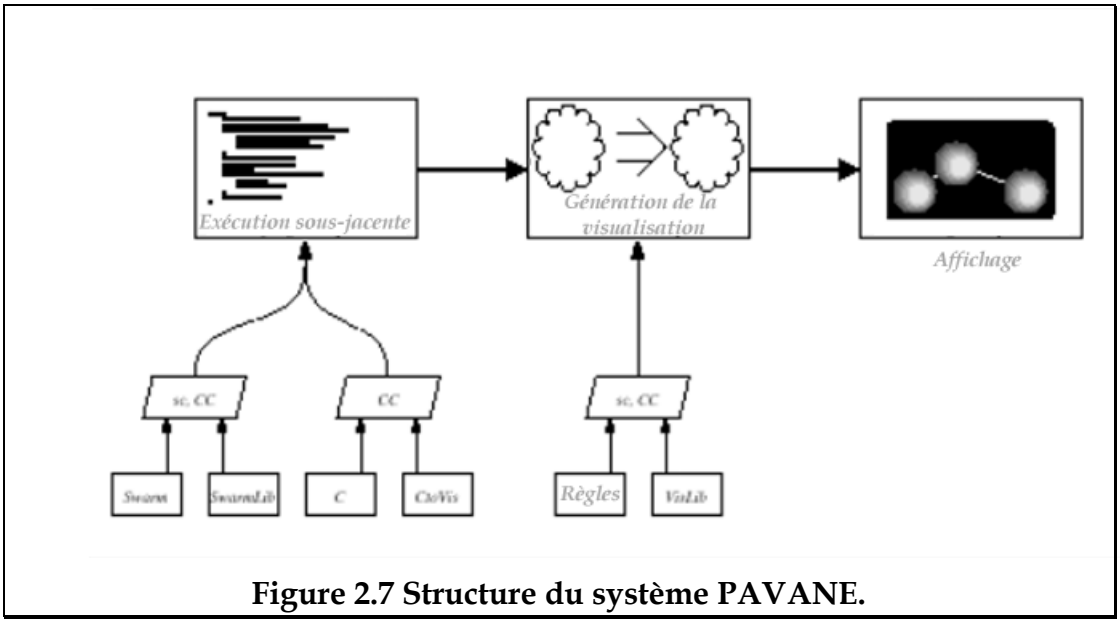


Figure 2.7 Structure du système PAVANE.

IV.2.3.2.1 Présentation d'un exemple de visualisation d'algorithme avec PAVANE

Nous allons maintenant décrire le système PAVANE par la présentation d'un exemple de visualisation de l'animation d'un programme des Tours de Hanoi (figure 2.8).

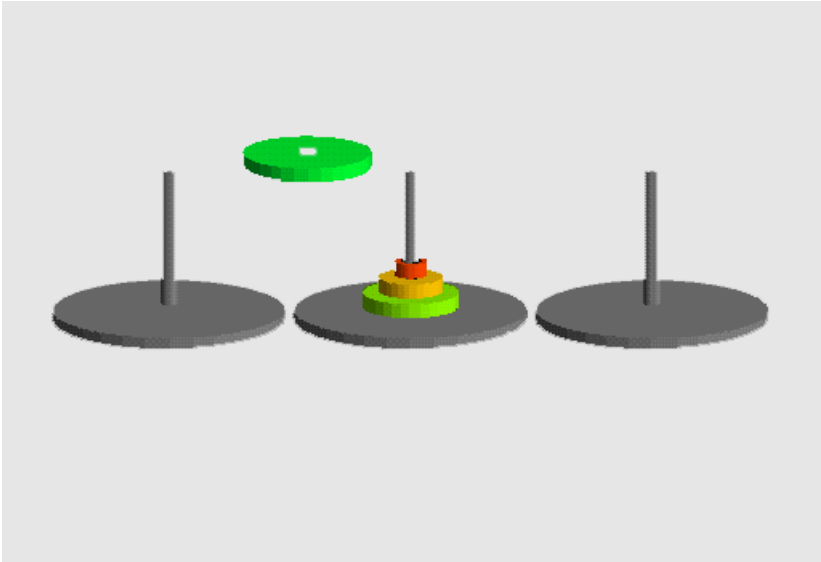


Figure 2.8 Visualisation des Tours de Hanoi par PAVANE.

Le premier point d'importance (comme nous l'avons vu dans le deuxième chapitre de ce mémoire) est l'implémentation utilisée pour l'algorithme. Le programme présenté ici est codé en langage C, utilise des variables globales indiquant pour chacune des trois tours : son contenu (la taille des disques présents est aussi conservée), la position du disque le plus haut et son état (quel est la tour de départ et celle d'arrivée). Cette implémentation définie quatre fonctions :

1) l'ajout d'un disque sur une tour,
2) l'effacement d'un disque d'une tour,
3) le déplacement d'un disque d'une tour vers une autre,
4) la résolution effective du problème des tours de Hanoï.

En plus de ces fonctions, l'animation de l'algorithme nécessite deux fonctions supplémentaires :

1) une fonction d'initialisation de la tour de départ, initialisant la variable globale désignant cette tour ainsi que sa représentation graphique,
2) une fonction principale initialisant les variables globales (le nombre de disques, la première tour de départ, …) ainsi que des appels à des fonctionnalités de PAVANE établissant les liens entre les données représentées et les données manipulées par le programme.

IV.2.3.2.2 Déclarations des règles de la visualisation

Présentons maintenant deux des règles permettant l'affichage et l'animation des disques aux différentes étapes de la visualisation du dérou-

lement de l'algorithme.

a) show_always : dessin des disques n'ayant pas changé de place

Cette première règle a pour finalité de dessiner, à chaque étape, les disques n'ayant pas changé de place depuis le dernier dessin des tours. Le code source de cette règle commenté est présenté dans la figure 2.8.

Cette règle exprime la puissance de l'expression des spécifications sous forme prédicative de la méthode proposée par Cox et Roman : elle définit par les expressions 0 <= t < 3 et 0 <= i <= n[t] une double itération contrainte par le test T[t,i] = oT[t,i].

```
show_always                                    Déclaration de la règle
    == array of 3 array of 9 integer T, oT;    Déclaration des données utilisées par
       array of 3 integer n;                   la règle : T et oT désigne les tours, n le nombre
       integer i, t                            de disques présent sur les tours, i et t
                                               sont ici utilisés comme compteurs.

    : towers(T), old.towers(oT), nums(n),      Liens entre les données et le programme :
      0 <= t < 3, 0 <= i <= n[t], T[t,i] = oT[t,i] towers et old.towers fait référence aux tours à
                                               deux étapes successives de l'algorithme
    =>
        _cylinder(_center   := [15*t-15,0,i], Spécification du disque à afficher
                  _radius   := T[t,i], _zsize := 0.8,
                  _fill     := true,  _facefill := true,
                  _color    := RGB(T[t,i])),

        _cylinder(_center   := [15*t-15,0,i], et de son centre (vide).
                  _radius   := .5, _zsize    := 0.9,
                  _fill     := true, _facefill := false,
                  _color  := [0,0,0]);
```

Figure 2.8 Spécification de la règle *show_always*.

b) show_init : mise en place de la représentation initiale

Cette règle affiche les disques présents sur la tour initiale au début de l'animation de l'algorithme. Le début est détecté par le prédicat *step* indiquant le numéro d'ordre de l'étape en cours. Le code source de cette règle est présenté dans la figure 2.9.

```
show_init
      == array of 3 array of 9 integer T;      les données s et x1 désignent ici l'étape
           array of 3 integer n;               et le numéro de la tour dont les disques
           integer s, x1                       sont à dessiner.

      :  towers(T), nums(n), step(s), this(x1),
         s = 0, n[x1] >= 0
      =>
            _cylinder(_center   := [15*x1-15,0,n[x1]],
                  _radius   := _ramp(0,0,5*top(T, n, x1),top(T, n, x1)),
                  _zsize    := 0.8, _fill     := true,
                  _facefill := true, _color    := RGB(top(T, n, x1)));
```

Figure 2.9 Spécification de la règle *show_init.*

IV.2.3.2.3 Déclaration de l'animation dans le programme

Du fait de l'utilisation du langage C, compilé, comme langage d'implémentation des algorithmes et de l'absence d'un mécanisme d'observation de l'exécution des programmes dans ce langage, PAVANE nécessite l'ajout d'instructions supplémentaires dans le code source du programme. Ces instructions signifient au système d'une part l'initialisation (déclaration des liens entre les données effectives et initialisation de la représentation graphique, que nous avons omis ici) et d'autre part la nécessité de la mise à jour de celle-ci. Ainsi, la figure 2.10 présente le code source de la fonction de résolution effective des tours de Hanoi dans laquelle ont été placés un appel à la fonctionnalité PAVANE *VisualUpdate* (ordonnant la mise à jours de l'animation) ainsi que la mise à jours des variables globales *FROM* et *TO* (nécessaires à celle-ci).

```
void transfer_hanoi (int from, int to, int level)          /* Recursively move disks from one tower to another */
{
  int other;
  if (level == 0) { move_disk(FROM = from, TO = to); VisualUpdate(); }
  else { other = 3 - from - to;
        transfer_hanoi(from, other, level - 1);
        move_disk(FROM = from, TO = to);
        VisualUpdate();
        transfer_hanoi(other, to, level - 1); }
}
```

Figure 2.10 Code source de la fonction de résolution effective des Tours de Hanoï avec les indications d'animation pour PAVANE.

IV.2.4 Critique de Balsa-II et de PAVANE et comparaison avec Zeugma

Les deux systèmes de visualisation d'algorithmes présentés, quoique fondamentalement différents par leur méthodologie de spécification des visualisations, se rejoignent dans la volonté d'indépendance par rapport à un langage de programmation et l'attachement à construire des représentations graphiques montrant les différentes opérations effectuées par un algorithme.

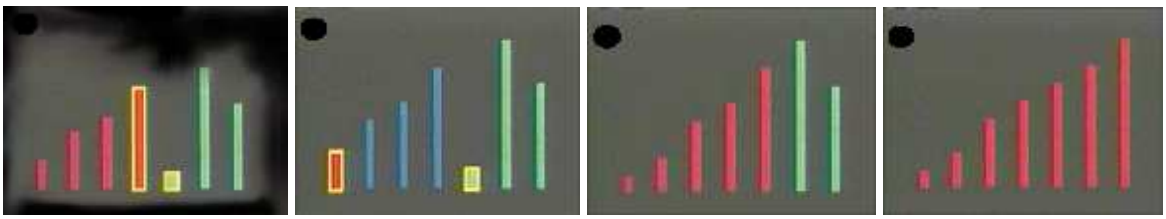
Comme nous l'avons montré dans ce mémoire, notre démarche vise à spécifier des visualisations de programmes pouvant également illustrer ce dernier point. La méthode que nous utilisons se rapproche de celle utilisée dans PAVANE que celle utilisée dans Balsa-II même si, comme par exemple dans des systèmes de visualisation d'algorithme comme Lens [Stasko97] (successeur de Tango[Stasko89, 90] et Xtango[Stasko92]), notre système permet l'attachement d'instruction d'animation à des expressions spécifiques du programme.

La première critique distinguant notre système de BALSA-II et de PAVANE est que, dans ces deux système ainsi que dans les paradigmes sous-jacents, les représentations sont intimement liées à un algorithme particulier, voire à son implémentation. Ainsi ces systèmes permettent difficilement d'utiliser une animation pour un autre algorithme (possédant alors nécessairement des *événements intéressants* différents) ou même pour une implémentation différente d'un même algorithme (les visualisations déclarées étant intimement liées aux structures de données utilisées dans l'implémentation). Comme le montre notre exemple d'animation d'algorithmes présentée dans le chapitre II de ce mémoire, notre système permet de construire des représentations pour une *classe* d'algorithmes et d'en vérifier la pertinence pour différents algorithmes de cette classe ou pour différentes implémentations.

Notre seconde critique par rapport à ces systèmes, même si Roman définit avec précision la démarche en jeu dans l'élaboration des visualisations, porte sur le fait que les visualisations proposées décrivent non pas l'algorithme, mais seulement ses effets : l'exemple de QuickSort en BALSA-II montre bien l'ordre temporel des déplacements mais ne montre nulle part comment l'algorithme trouve les objets à déplacer. Ou encore, l'exemple des Tours de Hanoi en PAVANE nous montre un beau film des déplacements des disques, mais l'observateur n'y trouve pas le moindre indice lui permettant de comprendre l'algorithme sous-jacent.

En fait, les systèmes de visualisation de programme actuels ne visualisent pas – au contraire de ce qu'ils prétendent – des algorithmes mais uniquement les modifications que les algorithmes engendrent sur leurs données : ce sont des systèmes d'animation visuelle de données. Nous invitons le lecteur à se tourner vers notre présentation des algorithmes de tris présentés au chapitre II de ce mémoire (pages 86 – 102) pour y voir de *vrais* visualisations d'algorithmes. D'autres visualisations d'algorithmes, comme celles présentées par R. Baecker [Baecker81, 97a, 97b], présentent, elles, un réel souci de relier l'objet d'une représentation et la représentation elle-même. Par exemple, la figure 2.11 présente la visualisation de l'insertion d'un élément *à la bonne place* dans un tri par insertion. L'utilisation des animations et des couleurs permet de distinguer les différentes étapes de cet algorithme. Toutefois, ces animations d'algorithmes ne sont pas générées par un système mais construites spécifiquement afin d'illustrer différents points de vues sur des algorithmes déterminés. Ce travail, du fait de la prise en considération des différentes étapes présentes dans les algorithmes étudiés se révèle être un outil efficace dans l'enseignement des techniques de programmation.

De fait, notre système Zeugma est plus proche des démarches de Brown et Roman dans la construction de systèmes permettant de spécifier de nouvelles visualisations et animations d'algorithmes. En effet, nous considérons comme problématique centrale les questions : *Comment capter, dans la visualisation d'un algorithme, la totalité de son comportement ?* et *Comment s'assurer qu'effectivement rien n'a été caché ?* plutôt que *Quoi représenter ?* ou *Comment construire les objets présents dans la représentation graphique ?* C'est à cette problématique que nous tentons de répondre avec notre système Zeugma.



(a) (b) (c) (d)

A chacune des étapes, ou passe, l'observateur peut dans cette représentation distinguer clairement les éléments déjà triés, ceux en cours de tris et ceux restant à trier. De même, cette représentation illustre la méthode utilisée par l'algorithme pour placer un élément.
(a) Première comparaison de la quatrième passe, avec le premier des quatre éléments déjà triés ;
(b) Comparaison finale de la quatrième passe ;
(c) Fin de la quatrième passe, après que les cinq éléments aient été déplacés vers l'avant ;
(d) Les données sont triées.
Vert signifie "pas encore trié", rouge "trié" et le tour jaune indique une comparaison en cours.

Figure 2.11 Visualisation d'un tri par insertions dans Sorting Out Sorting [Baecker97a, page 45].

IV.3 Environnements de programmation

IV.3.1 Problématique des environnements de programmation

Cette section présente deux environnements de programmation intégrants des visualisations graphiques : FIELD et Zstep95.

IV.3.2 Le système FIELD

Le système FIELD [Reiss90a, 90b, 94a], est présenté [Reiss98] comme un environnement intégrant de nombreux utilitaires de programmation liés au système UNIX auquel ont été ajoutés des outils offrant une aide à la programmation et la visualisation de programmes C, C++ et Pascal. Nous allons présenter en premier lieu les principes qui ont conduit à sa construction puis nous présenterons un exemple de son utilisation.

IV.3.2.1 Principes fondateurs de FIELD

Le système FIELD à été conçu comme un environnement de programmation *ouvert*[131] intégrant des fonctionnalités d'édition, de compilation, de détection d'erreurs et de visualisation de programmes. De plus, cet environnement à été conçu pour permettre le travail dans plusieurs langages de programmation compilés.

IV.3.2.1.1 Echange de données dans FIELD

Cet environnement est d'abord basé sur un mécanisme de passage de messages entre les différents acteurs[132] le composant. Ce mécanisme, centralisé par FIELD, permet aux différentes applications d'envoyer des messages de requête d'exécution d'une autre commande ou de retour de données après une exécution.

De plus, afin de fournir des données aux différents outils le composant, cet environnement intègre une base de données de références croisées sur les programmes en incluant par exemple des données sur les références des fonctions définies, leurs portées, appels et descriptions, ainsi que des informations sur les données : les différents types définis, la hiérarchie des classes définies (dans le cas de C++) ainsi que l'emplacement de leur utilisation. Outre cette base de donnée, qui construit ses informations à partir du texte des programmes, du résultat de leur compilation ou d'outils du système UNIX comme les utilitaires de configuration (*make*) ou de gestion de version

[131] Le terme *ouvert* désigne ici un environnement dans lequel l'ajout de fonctionnalité par l'utilisateur est simple et rapide.

[132] Editeur, compilateur, débugger et outils de visualisation.

(*rcs*), FIELD propose une librairie permettant de collecter des informations sur l'exécution des programmes (les informations fournies par cette librairie concernent les allocations mémoire et les entrées/sorties effectuées par les programmes au cours de leur exécution) et de les rendre accessibles aux autres outils qui le composent.

IV.3.2.1.2 Outils proposés par FIELD

Les outils proposés par FIELD se basent sur un double souci : utiliser de façon optimale des capacités graphiques des stations de travail utilisées aujourd'hui, et fournir aux utilisateurs des moyens de parcourir la structure des programmes et d'inspecter leur comportement. Ces outils sont :

- un éditeur d'annotation indiquant les annotations relatives aux références croisées, à des marqueurs fixés par l'utilisateur (*focus*), aux points d'arrêt ou à l'indication de la progression dans l'exécution des programmes,
- l'affichage d'un arbre d'appel dynamique traçant visuellement la progression dans l'exécution du programme,
- un outil de visualisation de la hiérarchie des classes définies,
- un outil de visualisation des dépendances entre les différents fichiers définissant le programme,
- un outil de visualisation de la structure des données,
- un outil de visualisation de l'occupation de la mémoire,
- un outil de visualisation des entrées/sorties.

Ces outils[133], comme le dit Reiss dans [Reiss98], sont aujourd'hui considérés comme *communs* dans les environnements de programmation et se retrouvent dans des systèmes à vocation de recherche comme Marvel [Kaiser88], Arcadia [Taylor89], Proteus [Graham92] et TPM que nous avons décrit plus haut, ou même des systèmes commerciaux comme, par exemple, CodeVision de Silicon Graphics (SGI) ou ObjectWorks de ParcPlace.

IV.3.2.2 Présentation de FIELD

Nous venons de voir que le système FIELD est composé d'un certain nombre d'outils. Nous allons présenter ici ceux qui se rapprochent le plus

[133] Il est important de noter, comme le dit Reiss dans son article [Reiss97], que les outils proposés dans FIELD s'inspirent de travaux antérieurs comme ceux qui ont abouti au système PECAN [Reiss84, 85], Magpie [Delisle84] basé sur le langage SmallTalk [Goldberg83] ou encore Gandalf [Notkin85] pour la visualisation du code sources des programmes ou de leur exécution ; et des travaux de Myers et de son système INCENSE basé sur la visualisation de la structure des données [Myers83].

des outils inclus dans notre système Zeugma : l'éditeur d'annotation et la visualisation de l'arbre d'appel des fonctions d'un programme pendant son exécution.

La figure 3.1 présente une capture d'écran rassemblant une partie des outils existant dans FIELD. Cette capture d'écran montre l'interface principale du système (haut et milieu à gauche), une visualisation graphique des données (en haut à droite), la visualisation de la dépendance entre les différents fichiers du programme (en bas à gauche), la visualisation des données et de la progression dans le programme sous une forme textuelle (au centre à droite) et l'éditeur d'annotation (en bas à droite).

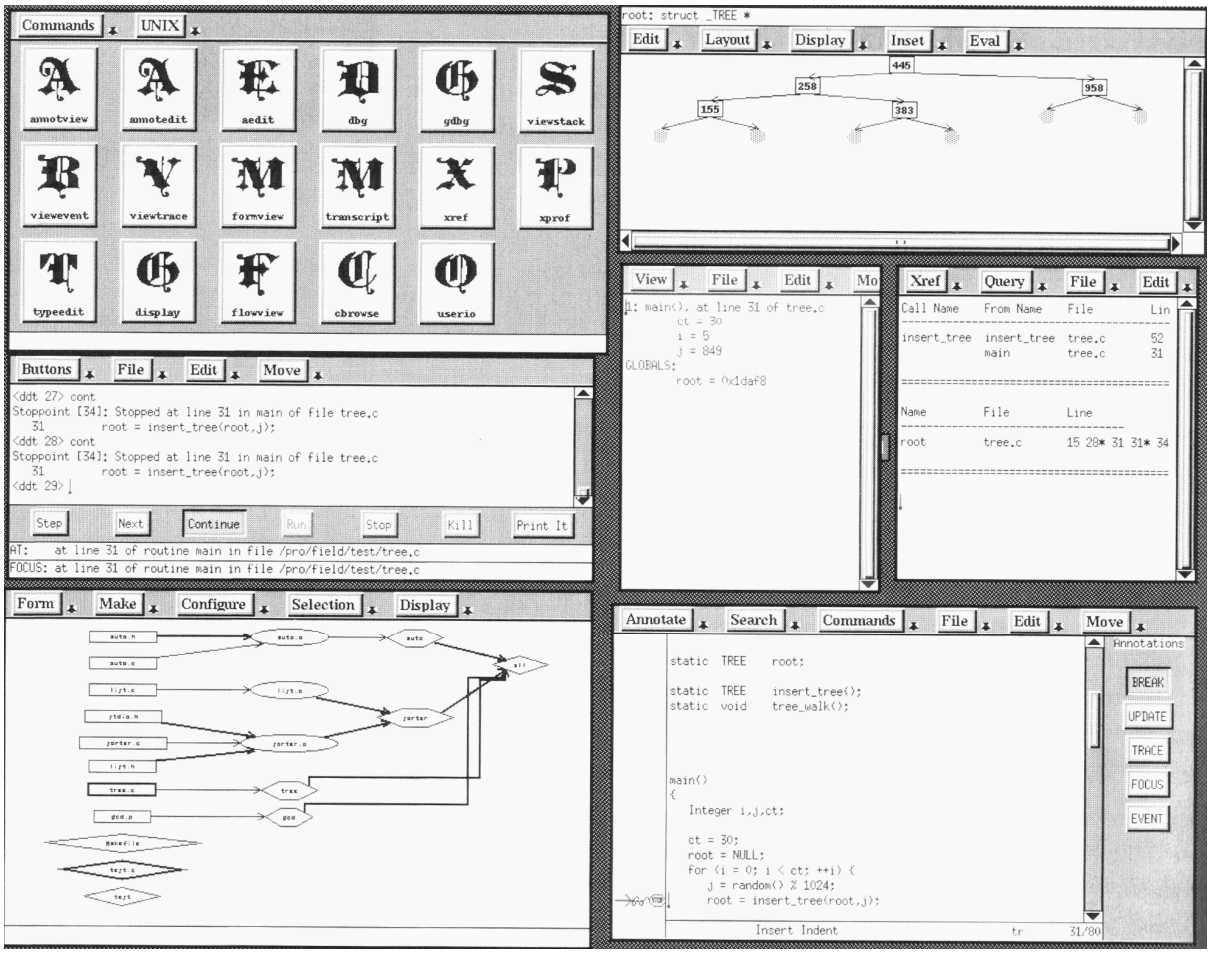


Figure 3.1 Aperçu des fonctionnalités proposées par FIELD [Reiss98].

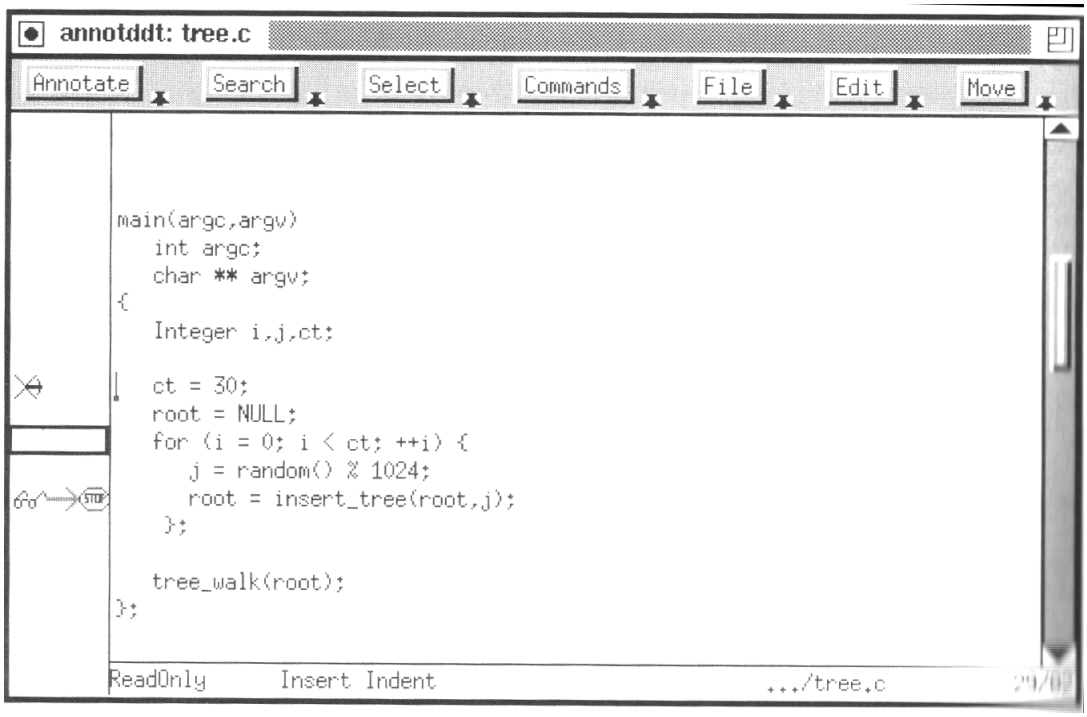


Figure 3.2 L'éditeur d'annotation de FIELD [Reiss97].

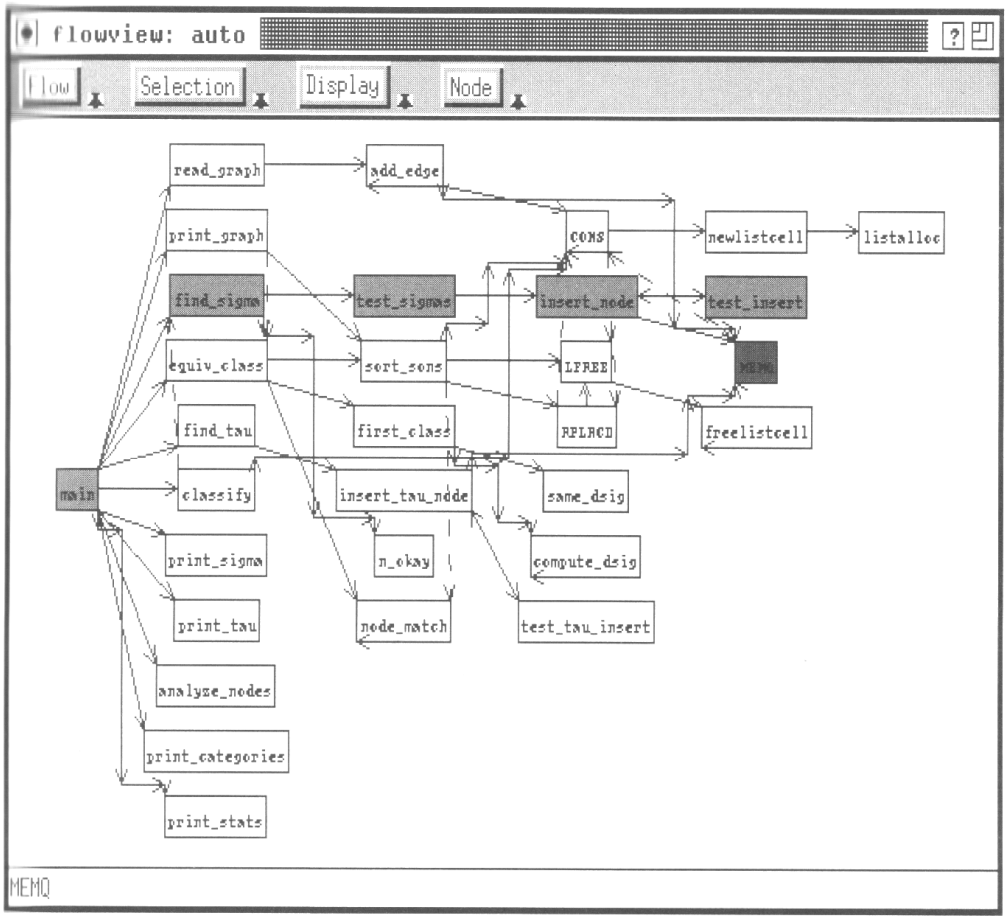


Figure 3.3 Visualisation de l'arbre d'appel dans FIELD [Reiss97].

IV.3.2.2.1 Un éditeur d'annotation

L'éditeur d'annotation du système FIELD (présenté dans la figure 3.2) est le premier outil que ce système a contenu. Il fournit à l'utilisateur un éditeur de texte complet ainsi qu'un outil permettant la spécification visuelle d'annotations sur le texte des programmes (partie gauche de l'éditeur dans la figure 3.2). Les annotations sont alors signalées par des icônes les symbolisant. Ces annotations peuvent être affichées ou non et même servir de moyen de parcours du programme comme autant de marques disposées à des emplacements précis du code. Les annotations, en plus d'être disposées manuellement par l'utilisateur, peuvent également être mises à jours automatiquement par le serveur sous-jacent à FIELD provoquant ainsi l'affichage de la ligne de code exécutée, changeant pour cela au besoin de fichier affiché.

Ainsi, l'éditeur d'annotation fournit un outil de visualisation du code de manière statique (parcours par les annotations présentes) aussi bien que dynamique (mise à jour automatique de l'indication de la ligne de code exécutée).

IV.3.2.2.2 La visualisation de l'arbre d'appel

La visualisation de l'arbre d'appel d'un programme en cour d'exécution (cf. figure 3.3) permet la visualisation graphique sous la forme de graphes de la progression de l'exécution du programme.

Cet outil récupère les informations sur la structure de l'arbre d'appel du programme à partir de la base de données de références croisées de FIELD, structure ces données de manière hiérarchique en regroupant les fonctions par fichier, les fichiers par répertoires et les répertoires par rapport à leur parent[134]. Il fournit une série d'outils variés permettant à l'utilisateur de focaliser la visualisation sur tel ou tel point particulier de l'arbre en donnant de nombreuses informations sur le programme par l'intermédiaire d'une fenêtre textuelle. Il fournit également un outil de visualisation graphique dynamique de la progression de l'exécution. Ainsi, la figure 3.3 présente un exemple de cette visualisation : le nœud désignant la fonction en cours d'exécution apparaît en foncé et les autres fonctions présentes sur la pile d'appel apparaissent en grisé.

IV.3.2.3 Critique de FIELD et comparaison avec Zeugma

Comme Xbvl, le système FIELD est un environnement de programmation complet : il contient en effet les différents niveaux nous paraissant indispensables pour de tels systèmes :

[134] Ce type de fonctionnalité apparaît également dans des systèmes plus récents comme BBS [Holt96] ou Rigi [Muller93] afin de présenter *visuellement* le résultat de calcul d'abstractions sur des programmes de grande taille.

1) une étude approfondie des programmes, par divers outils, permettant la construction et l'utilisation d'informations de bas niveau (des instructions) comme de plus haut niveau (références croisées ou les différents flots),
2) de multiples vues simultanées du programme illustrant différents aspects de celui-ci,
3) la capacité intrinsèque à intégrer de nouveaux outils de visualisation.

Notre système Zeugma s'inspire, dans sa dimension d'outils à la mise au point des programmes, de ces trois niveaux. Parmi eux, celui qui nous paraît le plus important est le troisième : la capacité d'un système à intégrer de nouveaux outils de visualisation. C'est également la démarche que nous avons voulu suivre dans l'implémentation de notre système, étendant ainsi considérablement l'environnement de programmation Xbvl.

La seule limitation opposable à de tels systèmes, limitation due au choix des langages visualisés, est la complexité et le temps pris par le système pour tester même des modifications limitées des programmes. En effet, si un utilisateur travaille sur une petite partie d'un grand programme, le test d'une modification de cette partie le forcera à une recompilation de la totalité. Nous pensons, et c'est là une des motivations principales guidant notre choix du langage Lisp pour Zeugma, que des environnements de programmation aidant à la mise au point de prototypes ou d'algorithmes sont plus efficaces s'ils utilisent des langages interprétés.

IV.3.3 Le système Zstep95

Le système ZStep95, développé au MIT par Henry Lieberman et Christopher Fry [Lieberman95, 97] [Fry95], est un outil destiné à aider le programmeur dans la recherche d'erreurs ainsi que la compréhension du lien existant entre le code source des programmes et leur exécution. Ce système est conçu pour traiter des programmes écrits en langage Lisp.

IV.3.3.1 Principes fondateurs de ZStep95

Le système Zstep95 vise à aider le débogage et la compréhension des programmes Lisp via des outils basés sur :

- l'*instrumentalisation* : l'activité consistant à rechercher le comportement effectif d'une partie du code du programme lors de son exécution. Les outils communs que les environnements de recherche d'erreurs fournissent à cet effet sont en règle générale les mécanismes de placement de points d'arrêt, de trace de variable ou l'insertion d'instruction d'impression qui permettent de suivre la progression de l'exécution. Le problème principal avec le placement des points d'arrêt est que l'utilisateur du sys-

tème doit savoir *où* les placer afin de visualiser la partie du programme en cause dans des erreurs,

- la *localisation* : l'activité consistant à isoler la partie du programme responsable du comportement inadéquat de son exécution sans en avoir idée a priori. Les outils communs utilisés à cette fin sont ceux qui permettent une exécution *pas à pas* du programme limitant ainsi intéractivement le comportement de l'interprète.

La solution apportée par Zstep95 à ces problèmes est de fournir une structure de contrôle réversible. Ce système conserve un historique complet de l'exécution du programme ainsi que de ses effets de bords. Comparé à d'autres travaux sur la réversibilité de l'exécution des programmes (comme ceux de [Balzer69], [Zelkowitz73], [Lieberman84a], [Lieberman87], [Moher88] et [Agrawal91]) dont le souci principal est de déterminer quel est le type d'information minimal nécessaire à la réversibilité, Zstep95 se concentre particulièrement sur le fait de pouvoir fournir une réversibilité de l'exécution du programme ainsi que des visualisations graphiques ou des effets de bords qu'il met en œuvre.

Dans la section suivante, nous allons présenter, à l'aide d'exemples, les outils intégrés à Zstep95 qui permettent le contrôle et le suivi de l'exécution des programmes ainsi que du parcours de son historique.

IV.3.3.2 Présentation de Zstep95

Ce système permet une navigation bidirectionnelle dans l'historique de l'exécution d'un programme et met à jour, à chacune des étapes, les représentations textuelles (du code exécuté ou des valeurs manipulées) et les effets de bords (incluant les représentations graphiques). Il propose un outil de navigation (fenêtre au milieu gauche de la figure 3.4) qui permet :

- une progression étape par étape (avec les boutons [>] et [<]),
- d'aller directement au début ([<<<]) ou à la fin ([>>>]) de l'exécution,
- de progresser dans l'exécution sans arrêt tout en affichant les variations des valeurs manipulées (avec [>>] pour une progression dans le sens normal et [<<] dans le sens inverse de l'exécution) : pendant l'exécution du code du programme une fenêtre affichant la ou les variables manipulées se déplace en parallèle à l'expression active, actualisant leur valeur à toutes les étapes (cf. figure 3.5),
- de progresser dans l'exécution jusqu'à l'expression ayant abouti à une modification de la représentation graphique générée par le programme (avec [G <] et [G >]).

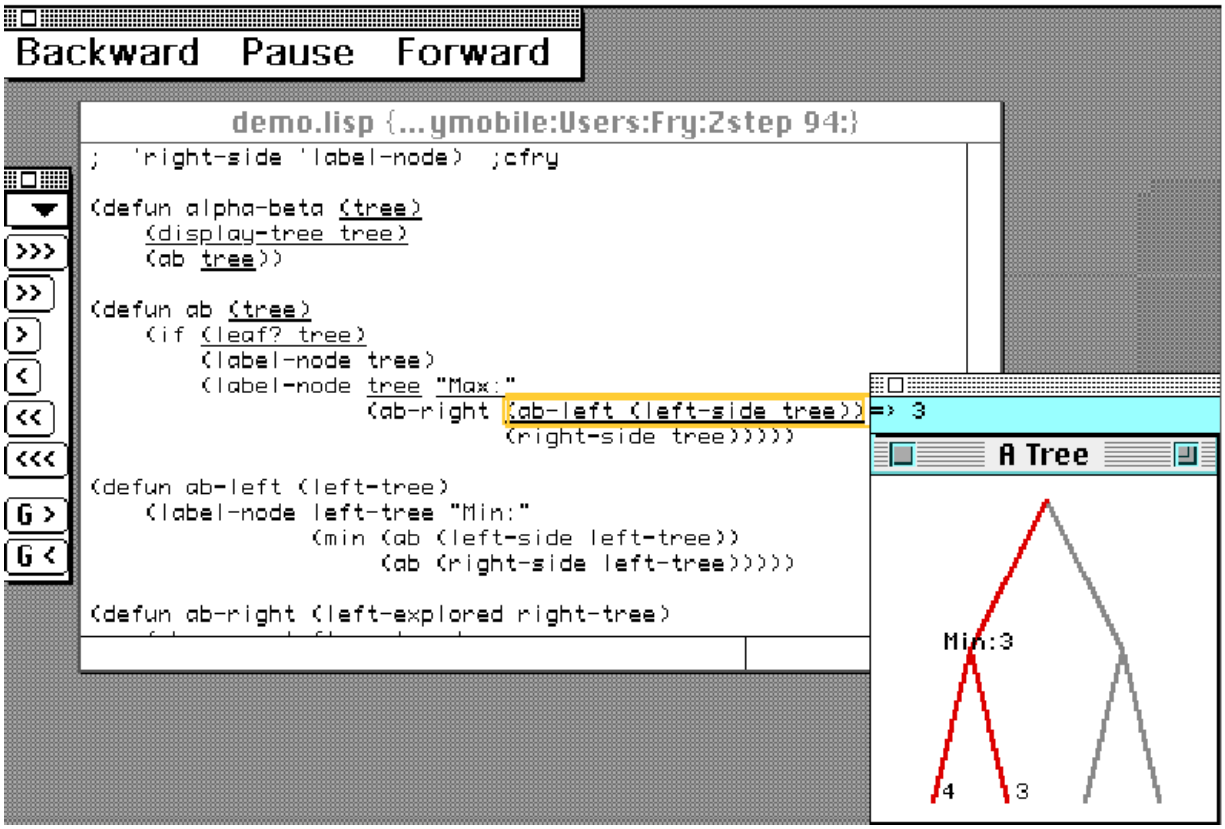


Figure 3.4 Vue de Zstep95

Ainsi, comme le montre la figure 3.5, des fenêtres affichant la valeur des variables manipulés se déplacent en parallèle à l'indication de la progression dans l'exécution. Cette fenêtre affiche non seulement les valeurs des variables mais aussi les valeurs résultantes de l'évaluation des expressions[135].

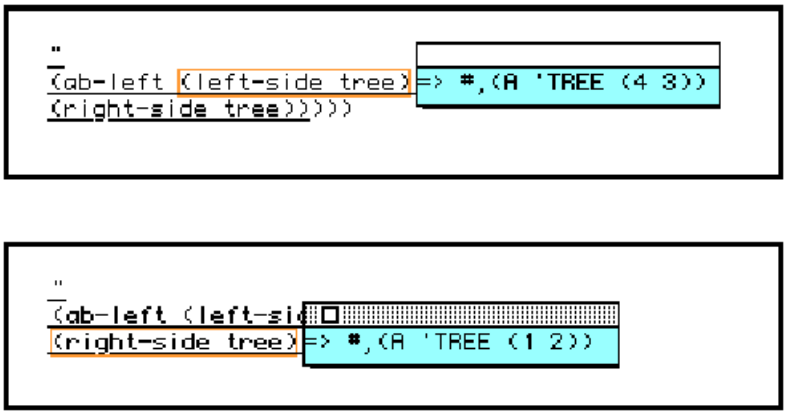


Figure 3.5 Visualisation des variations des valeurs manipulées.

Le programme présenté en exemple implémente le calcul d'un alpha-béta et construit en parallèle à la résolution une représentation graphique par l'affichage d'un arbre. Au cours du calcul, le programme indique le parcours de l'arbre (par la coloration des branches) et les valeurs présentes aux diffé-

[135] Rappelons qu'en Lisp l'évaluation de toute expression retourne une valeur.

rents nœuds (de manière textuelle). Comme le montre la figure 3.6, Zstep95 permet, pour chaque objet graphique de la représentation, soit d'afficher le code source du programme ayant abouti à son affichage, soit de replacer le programme dans le contexte où l'affichage a eu lieu. Si ce dernier choix est sélectionné, le contexte programmatoire (la pile d'exécution et l'état des variables) et la représentation graphique sont restitués.

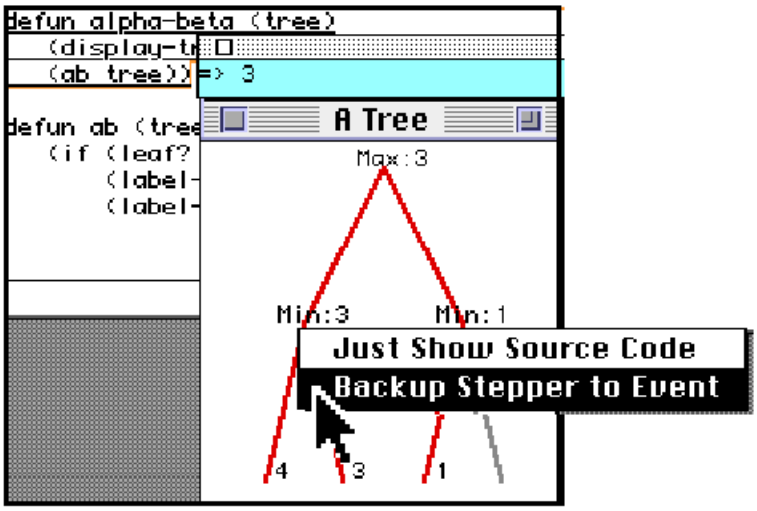


Figure 3.6 Progression interactive guidée par la représentation graphique.

Outre les outils présentés précédemment et correspondant aux activités d'instrumentalisation, le système prévoit un outil d'affichage localisé des messages d'erreurs de l'interprète (cf. figure 3.7) permettant leur localisation directe[136].

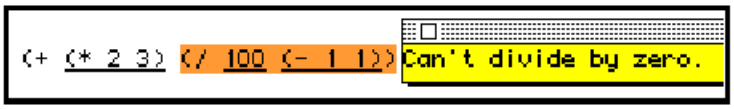


Figure 3.7 Affichage d'un message d'erreur et indication de sa localisation.

Ainsi, Zstep95 se présente comme un outil d'aide à la détection d'erreur. Il offre deux avantages :

1) visualiser le comportement du programme et les variations dans les valeurs manipulées,
2) guider la visualisation du programme par une interaction sur les opérations graphiques qu'il effectue.

IV.3.3.3 Critique de Zstep95 et comparaison avec Zeugma

Zstep95 nous a grandement influencé dans la mise au point de notre outil de visualisation de l'historique des animations de programmes de

[136] Un outil comparable existe dans Xbvl mais, à la différence du traitement des erreurs dans Zstep95, il permet à l'utilisateur (suivant le type d'erreur) d'effectuer des modifications permettant la reprise de l'exécution du programme.

Zeugma. En effet, notre outil intègre, lui aussi, cette dimension de réversibilité dans la visualisation de l'évolution des données aussi bien que dans les différentes étapes des visualisations graphiques.

La critique principale que nous opposerons au système Zstep95 est qu'il n'est pas un système de visualisation de programmes, même s'il se présente comme tel : ses outils ne permettent pas à l'utilisateur de construire des représentations graphiques du programme ou de son comportement autrement que par le programme lui-même. De plus, comme FIELD le propose, Zstep95 ne se présente pas comme un système ouvert permettant à ses utilisateurs d'avoir accès aux différentes informations qu'il manipule.

IV.4 Conclusion : situation de Zeugma dans le domaine de la visualisation de programmes

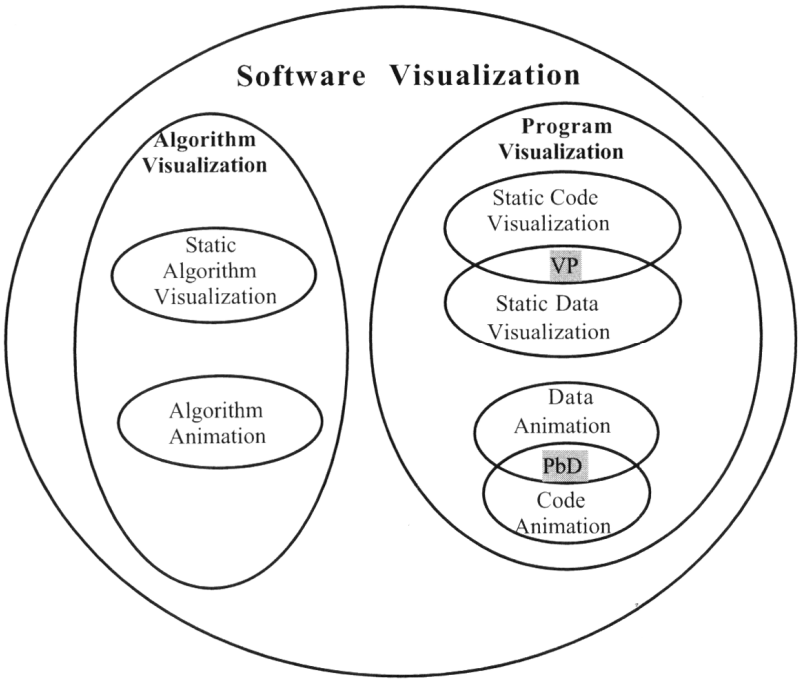


Figure 4.1 Fondement de la classification des systèmes de visualisation de programmes de Price [Price97].

La classification des systèmes de visualisation de programmes, présentée par Blain Price, Ronald Beacker et Ian Small dans [Price97] en introduction à un ouvrage dédié à la visualisation de programmes et qui constitue déjà un ouvrage de référence dans le domaine, établit une série de critères permettant d'évaluer les mérites relatifs des différents systèmes de visualisation de programmes existants.

Leur classification est fondée, comme l'illustre la figure 4.1 sur une distinction, dans les *programmes*[137], entre algorithme et implémentation[138]. Pour

[137] Nous utilisons le terme *programme* comme traduction de *software*.

[138] L'implémentation correspondant au terme anglais *program*.

eux, il existe une différence fondamentale entre un algorithme et le texte du programme qui implémente cet algorithme : la construction d'une représentation graphique d'un algorithme (dit de *haut niveau*) n'est pas automatiquement réalisable uniquement à partir de son implémentation (de *bas niveau*). Cette différence justifie alors la construction de paradigmes comme les *événements intéressants* ou les *visualisations déclarées* recréant artificiellement ce *lien* manquant entre un programme et les différentes étapes de l'algorithme.

Cette distinction entre *bas* et *haut niveaux*, respectivement l'algorithme et le code source du programme, est également un point particulièrement de leur classification : déterminant ce qu'ils nomment le *contenu*[139] qui permet alors de distinguer les *systèmes de visualisation d'algorithmes,* dont la fonction est « *de renseigner un utilisateur sur un algorithme* », des systèmes de *visualisation de programmes*, « *renseignant sur l'implémentation d'un algorithme* ».

Comme nous le montrons dans la section II.3 de ce mémoire, avec les exemples de représentations analogiques construites avec Zeugma, nous nous situons à ce qu'ils nomment *la croisée des systèmes d'animation d'algorithmes et des systèmes de visualisation de programmes*. En effet, les visualisations d'algorithmes que nous présentons *représentent* effectivement les différentes étapes d'algorithmes de tris et les visualisations par ville et par araignées que nous présentons renseignent sur des aspects de l'implémentation des programmes.

Une dimension, fondamentale dans notre système, qui n'apparaît pas dans la taxonomie présentée par Blain Price et al. est ce que nous nommons la *prise de distance* par rapport aux programmes dans la spécification des représentations. Cette dimension permet aux représentations analogiques que nous construisons d'être *indépendantes* d'un contexte, un programme, particulier et de pouvoir s'appliquer à de multiples programmes, qu'ils soient connus ou inconnus de l'utilisateur et du concepteur des représentations. Mais elle permet également, par la spécification du concepteur des représentations analogiques de nouveaux *aspects* de programmes, non seulement aux utilisateurs de notre système d'utiliser les paradigmes qui leur sont familiers mais également d'étendre le champ de compétence de notre système à des applications utilisant des techniques spécifiques d'analyse des programmes.

Finalement, même si la spécification de visualisations d'algorithmes représentant *effectivement* les différentes étapes reste une tâche difficile, les exemples que nous présentons montrent qu'il est possible de construire des représentations à partir desquelles les algorithmes sont déductibles. De plus, les représentations que nous proposons ne sont pas liées à un algorithme particulier mais à un *type* d'algorithme manipulant des listes de nombres et utilisant la récursivité : il est donc possible de construire la visualisation d'algorithmes uniquement à partir d'une étude de la composition et du comportement des programmes.

[139] Content.

Notre système Zeugma peut ainsi être qualifié de *Software visualization system* au sens de Blain Price car il permet de représenter tous les éléments du programme présents dans la figure 4.1. Nous nommons, comme annoncé à la page 50 de ce mémoire, ces systèmes des *environnements de programmation analogique*.

Chapitre V

Correspondances

La Nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles ;
L'homme y passe à travers des forêts de symboles
Qui l'observent avec des regards familiers.

Comme de longs échos qui de loin se confondent
Dans une ténébreuse et profonde unité,
Vaste comme la nuit et comme la clarté,
Les parfums, les couleurs et les sons se répondent.

Il est des parfums frais comme des chairs d'enfants,
Doux comme les hautbois, verts comme les prairies,
Et d'autres, corrompus, riches et triomphants,

Ayant l'expansion des choses infinies,
Comme l'ambre, le musc, le benjoin et l'encens,
Qui chantent les transports de l'esprit et des sens.

Baudelaire
[Beaudelaire, page 62]

V Conclusion et perspectives

V.1 Nos principales contributions

Nous avons présenté une méthode originale de création de liens entre des programmes et des représentations permettant la spécification de représentations *analogiques*. Basée sur la prise en compte *d'aspects* des programmes (désignant toute information calculable autant sur sa composition que sur son comportement) reliés à la description des représentations, elle permet la génération d'animations d'algorithmes comme de visualisation de la composition ou du comportement des programmes. Notre méthode se distingue des paradigmes de l'animation d'algorithmes par le type de relations établies entre les programmes et les animations (en se basant sur une étude des *caractéristiques* des programmes) ou de la représentation de programme par le type de représentations générées (en permettant la création de représentations graphiques complexes et animées). De plus, le système d'aide à la conception de représentations de programmes que nous avons conçu, Zeugma, permet également l'utilisation des représentations comme éléments d'un environnement de programmation. Les exemples que nous avons présentés illustrent ces deux dimensions de notre contribution : la construction de représentations à partir d'abstractions calculées sur les programmes (les *aspects de programmes*) et leur utilisation comme éléments d'un environnement de programmation, peut visualiser des aspects de programmes difficiles à percevoir autrement.

V.1.1 Représentations d'*aspects* de programmes

Les paradigmes de construction de représentations graphiques d'algorithmes se basent actuellement sur des particularismes des programmes qui les implémentent (cf. section IV.2). Même si ces méthodes permettent de construire des animations illustrant le comportement de tel ou tel algorithme, elles demandent aux utilisateurs de bien connaître ces programmes et ne garantissent pas l'efficacité des représentations, illustrant parfois uniquement certaines étapes des algorithmes que ces représentations sont sensées visuellement expliciter[140].

Notre méthode de construction de représentation du *comportement* des programmes, basé sur la mise en relations d'*aspects* calculés sur les programmes, permet de spécifier des animations dépendant d'événements com-

[140] Bien entendu, il existe des contextes dans lesquels il est souhaitable de limiter à quelques étapes particulières, comme l'illustre notre exemple de l'animation du tri par insertion présenté à la section II.3.4.5, notre méthode permet de spécifier les étapes particulières prises en compte dans une animation d'un algorithme.

putationels (par exemple, l'entrée dans une fonction ou l'évaluation paramètres) communes à tous les programmes Lisp. L'utilisation de telles propriétés dans la construction de visualisations d'algorithmes permet, comme nous l'avons montré dans la section II.3.3, de construire des représentations applicables à des classes d'algorithmes qui répondent à une série de critères minimaux (utilisant des appels de fonction), critères par ailleurs toujours présents dans l'écriture de programmes en Lisp. Ainsi, les représentations que notre méthode permet de construire ne seront plus liées à *un* algorithme ou *un* programme particulier : elles sont utilisables pour illustrer tout algorithme. L'utilisation d'une représentation graphique animée commune pour l'illustration d'algorithmes permet aux utilisateurs d'une part une facilité dans la lecture des représentations et d'autre part une meilleure évaluation de l'utilité de ces représentations.

De plus, ces aspects de programmes considèrent aussi bien le comportement que la composition des programmes. Notre méthode de construction de représentations analogiques peut également être appliquée la construction de représentations de la composition ou du comportement des programmes (comme le montrent les exemples présentés aux sections II.3.1 et II.3.2). Cette méthode unifie donc, dans une même démarche, des domaines généralement distincts : l'animation d'algorithmes et la visualisation de programmes.

V.1.2 Représentations *analogiques* de programmes

En conséquence, les aspects des programmes, considérés comme le domaine origine dans la construction des représentations, sont mis en relation avec des descriptions de représentations, le domaine cible, suivant des relations de type *analogique*. Cette considération originale des liens entre programmes et représentations se traduit suivant les deux dimensions des analogies : les *relations* analogiques entre domaine d'origine et domaine cible, et la *perception* de représentations analogiques.

V.1.2.1 Liens analogiques entre programmes et représentations

L'utilisation du terme *relation analogique* entre deux domaines se réfère aux travaux de Derdre Gentner définissant, dans la Structure Mapping Theory, la présence de relations *analogiques* par le partage de propriétés de structure entre des objets appartenant à des domaines distincts. Notre méthode emprunte ce modèle de relations entre objets appartenant à des domaines différents et l'applique à une *perception de haut niveau* (cf. les travaux de Hofstadter et al.) des programmes et des représentations.

Cette notion de perception de haut niveau est l'expression d'une distance relative à une abstraction entre l'aspect des programmes utilisé comme domaine d'origine des représentations et les programmes eux-mêmes. Ainsi, les représentations que nous construisons peuvent se baser sur une étude des

expressions, des flots de contrôle ou de données syntaxiques, ou encore sur des particularités de l'exécution des programmes (comme ceux que nous avons cités plus haut). Notre méthode laisse à l'utilisateur la liberté de :

- définir ses propres abstractions (*aspects*) sur les programmes et de les utiliser comme description du domaine d'origine,
- de spécifier la description des représentations générées en définissant des ensembles d'objets graphiques et de propriétés comme le placement dans l'espace ou la couleur.

L'utilisation combinée de ces deux modèles des analogies, comme relations de structures et comme perceptions de haut niveau, permet à notre méthode de fournir un descriptif complet des éléments mis en relations et de laisser un important degré de liberté dans la description du domaine d'origine, comme des textes répondant à une syntaxe particulière, comme des textes décrivant un processus et finalement comme des processus actifs.

V.1.2.2 Perceptions analogiques

La seconde implication de l'utilisation du terme *analogique* fait référence à la notion de *perception analogique*. Comme nous l'avons exprimé, notre méthode, laisse une entière liberté autant dans le choix des aspects des programmes représenté que dans les éléments de la représentation. Toutefois, considérant la question de la *perception* comme centrale dans l'élaboration de systèmes de visualisation de programmes, l'originalité de notre démarche réside également dans le type de représentations présentées. Les représentations de programmes comme des araignées sur une toile ou comme des villes utilisent à dessein des éléments graphiques avec des comportements ayant induisant une compréhension en dehors du contexte de la programmation, pour que la lecture des représentations puisse être guidée par cette même compréhension intuitive des éléments présents et de leur comportement. La construction d'une représentation analogique, mis à part le schéma de lecture induit par notre méthode de spécification, réside alors dans le choix d'éléments graphiques ou d'animations dont les effets visuels permettrons de déduire visuellement telle ou telle particularité des programmes. Les exemples présentés montrent que des références architecturales ou animalières peuvent effectivement représenter, ou permettre de visualiser, la complexité, l'organisation et des aspects du comportement des programmes difficilement déductible d'une lecture de leur code source.

V.1.3 Zeugma

Nous avons développé le système Zeugma afin d'offrir un outil d'aide à la conception, à la mise au point, à l'utilisation et à l'expérimentation de représentations analogiques de programmes.

Ce système intègre ainsi les différentes étapes de notre méthode et permet d'intégrer aisément de nouveaux aspects de programmes alors directement utilisables dans ce processus.

Les différents outils que notre système propose au cours de ce processus de conception et de mise au point des représentations analogiques, décrits dans le chapitre III de ce mémoire, constituent également une originalité de notre système par rapport aux divers systèmes de conceptions d'animations d'algorithmes ou de représentations de programmes. La plupart des systèmes d'animation d'algorithmes supposent des modifications manuelles des programmes ou algorithmes et nécessitent d'y adjoindre des appels à des fonctions d'une librairie écrite dans un langage particulier. Dans le cas de systèmes de visualisation de programmes qui permettant l'utilisation d'aspects calculés, ils se basent sur l'utilisation d'outils externes, tels que *make* ou *rcs*, demandant encore à l'utilisateur d'apprendre des outils et des langages supplémentaires. Notre système unifie, dans une même interface, l'ensemble de des activités de conception et d'exécution des programmes, de spécification et de calculs des aspects de programmes, de mise au point et d'utilisation des représentations analogiques.

V.1.4 Zeugma comme outil d'expérimentation des représentations

Une autre originalité de notre système est qu'il est à la fois un outil de conception de représentations analogiques et un environnement de programmation qui intègre des outils d'aide à la lecture des représentations générées, au contrôle pas à pas des exécutions ou à la navigation dans l'historique d'exécutions. Ainsi, l'utilisateur pourra :

interroger les représentations générées afin d'obtenir des informations sur l'objet programmatoire représenté par tel ou tel objet graphique,

- décider d'un objet ou d'un type d'objet programmatoire provoquant un arrêt temporaire de l'exécution des programmes étudiés,
- naviguer a posteriori dans l'historique d'une exécution en visualisant en parallèle les animations de sa représentation analogique ce qui lui permettra d'une part une double lecture (textuelle et analogique) de celle-ci et d'autre part, de par la possibilité d'effectuer des retours en arrière, de reconstruire les étapes ayant abouti à un état particulier d'une exécution.

Comme le montrent les exemples de représentations analogiques décrites, il est souvent nécessaire d'avoir des outils informant l'utilisateur sur le lien entre un élément de la représentation et l'élément du programme correspondant. De même, si un outil de contrôle de l'exécution pas à pas est maintenant un élément commun de tout environnement de programmation, la diversité de choix quant à l'élément du programme originaire de ces arrêts

reste une particularité de notre système. La possibilité de parcourir librement l'historique d'une exécution parallèlement à la visualisation des animations analogique de sa représentation demeure également un outil présent dans peu d'environnements de programmation.

V.2 Faiblesses et extensions de notre système

Dans l'étape actuelle du développement de Zeugma, nous avons un environnement qui tourne et fonctionne bien. Comme tout système très récent, il a principalement été utilisé par un utilisateur : son constructeur, nous-même. Cette utilisation a permis sa mise au point, a montré sa puissance dans la diversité des représentations qu'il permet de construire ou des programmes qu'il permet d'observer. Naturellement, cette utilisation a fait apparaître également des faiblesses et – surtout – un grand catalogue d'améliorations et d'extensions que nous prévoyons dans la suite de cette recherche.

Notons également que les représentations analogiques que nous avons expérimentées (les villes et les araignées) ont été construites, non point pour montrer leur adéquation, mais pour évaluer l'utilisation de Zeugma d'une part dans l'exploration de programmes (les villes comme des lieux à découvrir en parallèle à la découverte d'un programme) et d'autre part dans l'observation du comportement d'un programme (les déplacements et les regroupements des araignées à observer en parallèle aux appels de fonctions et à la formation de groupe fonctionnel).

Ces deux aspects de notre travail nous conduisent à formuler les critiques et les perspectives de développement suivantes :

1) Notre méthode, ainsi que le système Zeugma, demande à être utilisée par un nombre important d'utilisateurs de niveaux de compétences (dans la programmation) différents.
2) Notre système permet la représentation de tout programme écrit en Lisp mais il réside actuellement des limitations dans la taille maximale des programmes qu'il peut prendre en compte et dans la complexité des représentations générées.
3) Notre système Zeugma est actuellement limité à la construction de représentations analogiques de programmes, alors que notre méthode devrait également permettre la construction de représentations analogiques de données.
4) La description des représentations, même si elle permet la spécification de représentations analogiques complexes, n'est pas totalement en accord avec la méthode, elle reste liée à un schème générateur de représentation, à une représentation analogique particulière.

5) Les représentations permettent aujourd'hui d'aller d'un programme vers une représentation mais ne permettant pas d'interagir sur un programme à partir de sa représentation.

Voyons maintenant en détail ces critiques et les perspectives qu'elles impliquent.

V.2.1 Validation de Zeugma par son utilisation

La première critique de notre travail est la méthode de validation que nous proposons. Considérant notre travail comme novateur, nous l'illustrons par des exemples d'analogies ou par la comparaison de Zeugma avec d'autres systèmes. Ces exemples étant en nombre limité et uniquement élaboré par le concepteur du système, une approche qui permettrait d'étendre ces deux points serait de valider Zeugma auprès d'un panel d'utilisateur, composé, par exemple, d'ensembles de programmeurs de niveaux différents et de leur demander de construire des représentations analogiques de programmes.

Nous pensons structurer cette validation de la manière suivante :

1) De proposer le système Zeugma comme élément d'un cours d'introduction à la visualisation de programmes, confrontant alors la présentation de notre méthode à celle des autres systèmes présents.

2) Nous comptons, en collaboration avec un collègue ergonome, organiser des deux types de sessions destinées à des programmeurs :

   - à des programmeurs considérés comme *novices*, des sessions de créations de représentations analogiques afin, d'une part les interroger sur ce qu'ils *perçoivent* des programmes, quelles analogies ils utiliseraient pour décrire, par exemple, le fonctionnement ou l'implémentation d'un algorithme particulier, et, d'autre part, leur présenter ces visualisations construites en utilisant notre système Zeugma afin qu'ils puissent exprimer les différences entre images subjectives et exemples de représentations construites à partir de ces images.

   - à des étudiants plus confirmés dans la programmation (en tous les cas pouvant clairement comprendre les notions d'aspects de programmes mis en œuvre dans notre méthode), des sessions de création de représentations analogiques avec Zeugma afin de tester la robustesse et l'utilisation autant de note méthode que de notre système.

A la différence de systèmes de visualisations d'algorithmes comme PAVANE ou Balsa, qui ne demandent pas une expertise de la programmation, notre système Zeugma demande actuellement une connaissance des

différents niveaux d'abstractions (les aspects) présents dans les programmes et c'est ce point qui, à notre avis, justifie encore la distinction entre novices et experts dans la validation de notre système auprès de programmeurs. Ainsi, l'expérience que nous avons de l'enseignement de la visualisation de programmes (à des étudiants de niveaux Licence et Maîtrise) nous permet d'insister sur la difficulté principale de ce domaine : la visualisation de programmes (comme d'algorithmes) demande une connaissance de la programmation et, par rapport à la problématique du questionnement de la validité de telle ou telle représentation ou animation, une réflexion sur la programmation, sur ce qu'est un programme ou un algorithme. Ainsi, et c'est un des points qui sont à l'origine même de notre travail, les systèmes actuels ne prennent pas en compte cette *culture* du programmeur, ses propres représentations internes qui l'aident dans la conception, la mise au point ou la découverte de programmes.

V.2.2 Limitation de la taille des programmes représentés

Nous avons actuellement testé notre système Zeugma pour représenter nos propres programmes mais également pour représenter des programmes qui nous étaient inconnus. Les limites de complexité auxquelles nous nous sommes trouvés confrontés sont de deux ordres : l'un par rapport à la taille des programmes représentés et le second par rapport à la complexité des représentations générées.

La limite de la taille des programmes que Zeugma peut actuellement représenter est, selon le plus grand programme que nous avons testé de l'ordre de 2000 lignes de Lisp (sans commentaires). Cette limite est le fait de contraintes techniques liées à la taille de la mémoire de travail du système sous-jacent (Xbvl) et à la limitation du nombre de transformations graphiques encapsulables liées à librairie graphique (OpenGL) utilisée. Ces limites sont extensibles mais demande d'une part la réécriture d'une partie de Xbvl et d'autre part de concevoir une gestion locale à Xbvl de cette pile de matrice de taille actuellement limitée.

La limite due à la complexité des représentations générées : si la représentation par des araignées permet l'étude de programmes composés de plus de 30 fonctions différentes, les images résultantes de son utilisation deviennent rapidement complexes et demandent une étude *par partie* du programme, invalidant temporairement l'animation d'une partie des araignées, ce que les outils proposés par Zeugma permettent de faire sans difficulté (cf. section III.2.1.5). La représentation analogique par des villes pose un type de problème différent, la complexité apparente des maisons générées étant en soit une information sur la composition du programme. Toutefois, il manque dans Zeugma un outil permettant de décidé du niveau général de granularité d'une représentation analogique, permettant une réduction globale de sa complexité après sa génération.

Une autre voie de résolution du problème de la complexité des représentations analogiques générées, que nous avons évoqué à plusieurs reprises dans ce mémoire, pourrait être l'utilisation d'aspects de programmes considérant de manière unitaire un ensemble de fonction ou un groupe d'expression suivant, par exemple, des critères fonctionnels. Ces aspects permettraient de générer des représentations plus générales illustrant par exemple l'architecture ou le *résumé* de la composition des fonctions des programmes.

V.2.3 Application de la méthode à la représentation de données

Notre système utilise actuellement comme matériaux de base de construction des représentations analogiques les programmes. Pourtant l'émergence de systèmes permettant la représentation analogique de données apparaît de plus en plus nécessaire au vu de l'accroissement de la taille des diverses statistiques ou banques de données disponibles nécessitant une présentation *analogique* des données qu'elles contiennent[141]. Notre méthode de construction de représentations analogiques de programmes, considérant les programmes comme des données, semble permettre une extension directe vers la manipulation de données *indifférenciées* dans le sens où elles ne décrivent pas un processus mais sont le résultat d'un processus. Il apparaît ainsi qu'un simple changement de point de vue par rapport aux données manipulées rendrait notre méthode tout à fait adaptée à la construction de représentations analogiques de données.

L'application de Zeugma à la représentation de données, vu les multiples travaux comme ceux de Eben Harber et al. [Harber94] ou Tiziana Catarci et al. [Catarci95] qui se positionnent actuellement, tout comme notre méthode, dans la nécessité de la création de liens *analogiques* ou *métaphoriques* entre données et bases de données et représentations graphiques, donnera alors une dimension supplémentaire à notre système, sans pour autant nécessiter une remise en cause fondamentale de son implémentation, de sa structure, ou du processus de génération des représentations.

De fait, pour que Zeugma puisse utiliser des données issues d'une base comme base de construction de représentations analogiques, la combinaisons des deux méthodes suivante nous semblent nécessaires :

- La définition de programmes spécialisés dans récupération des données dont le Zeugma représentera analogiquement le comportement (les animations d'algorithmes que nous présentons dans ce mémoire offrent un modèle de spécification de telles représenta-

[141] Ces données, quelles que soient les méthodes utilisées pour les collecter, correspondent à un domaine particulier mais peuvent nécessiter une présentation relative à l'utilisation qui en sera faite, rejoignant ainsi directement notre méthode de spécification de représentation analogique de programmes.

tions),

- La définition de nouveaux aspects de programmes, spécifiques à l'examen des données, permettant alors de spécifier des schèmes générateurs relatifs à telle ou telle structure de base de données.

Par rapport à la spécification d'aspects relatif à des données, il serait alors nécessaire de préciser si la représentation analogique vise à illustrer le type de base de donnée utilisé (relationnel, orienté objet, etc.), les structures des bases de données, la structure de telle ou telle base particulière ou encore un certain type de données spécifiques, démarche comparable à la spécification des différents niveaux d'abstractions calculées sur les programmes que nous définissons dans la présentation de notre méthode.

V.2.4 Description des représentations

Notre méthode se base sur la prise de distance entre un programme particulier et les éléments pris en compte dans la spécification d'une représentation analogique. Toutefois, la spécification des représentations est, elle, liée aux différentes caractéristiques relatives à ces aspects et est relative à un schème générateur de représentation. Dans l'implémentation de notre méthode, la conception d'une représentation ne demande pas la définition d'*aspects des représentations* comparables aux définitions des aspects des programmes. Cette faiblesse dans notre méthode, considérant différemment les programmes et les représentations, provient essentiellement de la démarche que nous avons mise en œuvre dans son élaboration visant la construction d'un environnement de programmation incluant des représentations analogiques animées des programmes et focalisant ainsi particulièrement notre étude sur ces derniers.

Les effets de cette faiblesse sont, par exemple, qu'une représentation avérée efficace pour représenter un aspect des programmes ne peut aisément être transposée pour illustrer un autre aspect dont les caractéristiques diffèrent dans leur organisation[142]. Il serait possible de palier à cette faiblesse en permettant la définition d'aspects spécifiant les représentations de la même manière nous permettons la définition d'aspects de programmes. Ces définitions d'aspect (spécifiant par exemple *les* araignées ou *les* villes, que nous pourrions nommer des *situations*) spécifieraient un ensemble de caractéristiques décrivant la génération des objets graphiques, les positionnements, les couleurs, …, relatifs à des situations. De la même manière que notre système

[142] Si les aspects ne diffèrent pas dans leur organisation (c'est-à-dire que les données qu'ils décrivent sont de structure similaire) les schèmes générateurs sont de structure identique. Le transport des caractéristiques graphiques d'une représentation est alors possible : nous utilisons, par exemple, cette possibilité dans la représentation de programmes par des araignées afin de construire l'image initiale selon diverses organisations de l'ensemble des fonctions des programmes.

distingue les aspects liés à l'étude de la composition des programmes des aspects liés a leurs comportements, ces aspects descriptifs des représentations devraient également distinguer la spécification d'aspects liés au dessin des objets graphiques de leurs animations.

Outre la définition de nouveaux aspects, liés à l'étude des représentations graphiques générées, cette nouvelle approche des représentations nécessiterait également une modification de la méthode de construction des schèmes générateurs de représentations afin qu'ils intègrent et permettent l'utilisation de cette nouvelle forme de spécification.

V.2.5 Unidirectionnalité du système

Un autre aspect critique de notre système réside dans l'unidirectionnalité des représentations. Il permet actuellement uniquement de générer des représentations graphiques à partir de programmes mais ne permet pas le chemin inverse, d'interagir sur les programmes à partir des représentations. Cette réversibilité permettra à Zeugma non seulement d'être un environnement de construction et d'utilisation de représentations analogiques de programmes mais également d'être un environnement de programmation permettant la construction de langages de programmations visuels, analogiques.

De fait, Zeugma possède aujourd'hui le potentiel permettant d'interagir sur les programmes à partir de leurs représentations, les structures des représentations, telles que nous les présentons à la section III.2.2.2, décrivent la corrélation entre objet graphique et objet du programme représenté. La représentation interne des représentations analogiques, que ces structures rendent apparentes, contient ainsi les informations nécessaires à une action rétroactive sur les programmes. Le problème principal dans l'implémentation d'une telle rétroactivité analogique sur des programmes réside dans la distance, fondamentale dans notre méthode, entre les programmes et les aspects utilisés dans la construction des représentations. Cette distance (les aspects décrivent des *informations* calculées *sur* les programmes) nécessite alors une fonction *inverse* permettant, à partir d'une caractéristique d'un aspect, de modifier le programme sous-jacent. Cette fonction inverse dépendra du niveau d'abstraction présent entre l'aspect et le programme : si la fonction inverse de la composition des fonctions ne semble pas posée de problèmes insolubles, quelle peut être la fonction inverse d'un flot de contrôle, de données, ou encore des aspects liés à l'exécution des programmes ?

La résolution de cette question amènera également à considérer la problématique de l'interaction sur les représentations. Pour reprendre les aspects précédemment cités et les exemples de représentations analogiques que nous présentons dans notre mémoire pour les illustrer, l'interaction sur les éléments composant une ville seront dépendantes du type d'élément représenté :

- les éléments présents à l'intérieur d'une maison, décrivant les expressions la composant, pourraient aisément être considérés comme éléments d'un langage visuel,
- les critères de construction des quartiers, liés à l'étude de la composition, auraient un statu différent et engageraient une réorganisation automatique de la ville selon les modifications effectuées sur la composition des maisons,
- le mouvement du personnage dans la ville, expression de la progression dans l'exécution du programme, pourra difficilement devenir un élément actif dans la modification des programmes[143].

La définition de ces différents modes d'interaction est nécessaire afin de conserver la cohérence *analogique* tant au niveau des relations entre programmes et représentations qu'au niveau de la *perception analogique* d'une représentation et des éléments qui la compose. Cette définition pourra ainsi guider la construction d'une méthodologie de spécification des éléments graphiques sous la forme d'aspects et l'avancée dans sa résolution se fera ainsi en parallèle avec la résolution de la faiblesse de Zeugma évoquée dans la section précédente. Ceci amènera Zeugma à devenir un outil permettant non seulement de travailler sur les programmes à partir de leur représentation analogique mais également à devenir une interface permettant la conception de nouveaux langages de programmations *analogiques* particulièrement nécessaires pour les interfaces homme – machines actuellement développées.

[143] Une voie possible de recherche dans l'utilisation d'éléments actifs dans la spécification de programmes par des éléments actifs peut venir des systèmes de programmation par l'exemple dont Tinker [Lieberman93] est un exemple.

Bibliographie

[Abchiche99] Abchiche N., *Elaboration, implémentation et validation d'une approche distribuée pour l'intégration de modèles de raisonnement hétérogènes : application au diagnostic de pannes électriques*, Thèse de Doctorat, Université Paris 8, janvier 1999.

[Ali Chérif 95] Arab Ali Chérif, *Réalisation, programmation et modélisation d'une société de robots*. Thèse de doctorat, Université Paris 8, 1995.

[Aristote] Aristote, *La Poétique*, Traduction Roselyne Dupont-Roc et Jean Lallot, Edition du Seuil, Paris, 1980.

[Arnowitz97] Arnowitz J., Willems E., Faber L. et Priester R., *Mahler, Mondriaan, and Bauhaus: improve application using artistic ideas to usability*, DIS'97 Amsterdam, Hollande, 1997, pages 13 – 21.

[Balmas95a] Balmas F., *Classifying Programs: a Key for Program Understanding*, Dans les actes de 7th Int. Conference on Software Engineering and Knowledge Engineering, Rockville, MD, 1995.

[Balmas95b} Balmas F., *Contribution à la conceptualisation de programmes : modèle, implémentation, utilisation et validation,* Thèse de doctorat, université Paris 8, Saint Denis, décembre 1995.

[Balmas97] Balmas F., *Toward a Framework for Conceptual and Formal Outlines of Programs*. 4th Working Conference on Reverse Engineering, Amsterdam (Hollande), octobre 1997.

[Baecker81] Baecker R.M. (assisté de Sherman D.), *Sorting Out Sorting*, film couleur/sons, Université de Toronto. Distribuée par Morgan Kaufmann, San Francisco, 1981.

[Baecker90] Beacker R.M. et Marcus A., *Human Factors and Topography for More Readable Programs*, Addison-Wesley, Reading, MA, 1990.

[Baecker97a] Beacker R.M., DiGiano C., Marcus A., *Software Visualization for Debugging*, Communications of the ACM, Vol. 40, No. 4, avril 1997.

[Baecker97b] Baecker, R.M. *Sorting Out Sorting: A case study of software visualization for teaching computer science*. Dans Software Visualization: Programming as a Multimedia Experience, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, Cambridge, Mass., 1997.

[Beaudelaire] Beaudelaire, *Les Fleurs du Mal*, Flammarion, 1991.

[Bazik97] Bazik J., Tamassia R., Reiss S. et van Dam A., *Software Visualisation in Teaching at Brown University*, Dans Software Visualization: Programming as a Multimedia Experience, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, Cambridge, Mass., 1997, pages 383 – 398.

[Beshers93] Beshers, C. et Feiner, S. *AutoVisual: Rule-based design of interactive multivariate visualizations*. IEEE Computer Graphics and Applications, 13(4), Juillet 1993, pages 41-49.

[Blackwell96a] Blackwell A., *Metaphor or Analogy: How Should We See Programming Abstractions?*, Dans P. Vanneste, K. Bertels, B. De Decker & J.-M. Jaques (Eds.), Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group, avril 1996, pages 105-113.

[Blackwell96b] Blackwell A., *Metacognitive Theories of Visual Programming: What do we think we are doing ?* Dans Proc. IEEE Symposium on Visual Languages, septembre 1996, pages 240 – 246.

[Blackwell97] Blackwell A., *Diagrams about thoughts about thoughts about diagrams*. In M. Anderson (Ed.), Reasoning with Diagrammatic Representations II: Papers from the AAAI 1997 Fall Symposium. Technical Report FS-97-02. Menlo Park, CA: AAAI Press, pages 77 – 84.

[Bongard70] Bongard M., *Pattern Recognition*, Hayden Book Co. Rochelle Park, N.J., 1970.

[Brayshaw88] Brayshaw M. and Eisenstadt M., *Adding data and procedure abstraction to the Transparent Prolog Machine (TPM).* In Kowalski, R. and Bowen, K, (Eds.), Logic Programming. Cambridge, MA, MIT Press, 1988.

[Brayshaw91] Brayshaw, M., and Eisenstadt, M. *A practical graphical tracer for Prolog.* International Journal of Man-Machine Studies, 35 (5), 1991, pages 597-631.

[Brown84] Brown M. et Sedgewick R., *A System for Algorithm Animation*, ACM Computer Graphics, Vol. 18, N°3, pages 177-186, juillet 1984.

[Brown85] Brown M. et Sedgewick R., *Techniques for Algorithm Animation*, IEEE Software, Vol. 2, N°1, pages 28-39, janvier 1985.

[Brown88a] Brown M., *Exploring Algorithm Animation using Balsa-II*, Computer, Vol. 21, N°5, pages 14-36, mai 1988.

[Brown88b] Brown M., *Perspectives on Algorithm Animation*, Dans Proceedings de ACM SIGCHI'88 Conference on Human Factors in Computing Systems, ACM, pages 33-38, mai 1988.

[Brown91] Brown M., *ZEUS: A System for Algorithm Animation and Multi-View Editing*, Dans Proceedings du Workshop IEEE 1991 sur les Langages Visuels, Kobe, Japon, pages 4-9, octobre 1991.

[Brown92] Brown M. et Hershberger J., *Color and Sound in Algorithm Animation*, Computer, Volume 25, N°12, décembre 1992, pages 52-63.

[Brown97a] Brown M. et Hershberger J., *Fundamental Techniques for Algorithm Displays,* Dans J.Stasko, J.Domingue, M.H.Brown, and B.A.Price (Eds.) Software visualization: programming as a multimedia experience. Cambridge, MA, MIT Press, 1997.

[Brown97b] Brown M. et Najork M., *Algorithm Animation Using Interactive 3D Graphics,* Dans J. Stasko, J. Domingue, M.H. Brown, and B.A. Price (Eds.) Software visualization: programming as a multimedia experience. Cambridge, MA, MIT Press, 1997.

[Brown97c] Brown M. et Sedgewick R., *Interesting Events*, Dans J. Stasko, J. Domingue, M.H. Brown, and B.A. Price (Eds.) Software visualization: programming as a multimedia experience. Cambridge, MA, MIT Press, 1997.

[Bukley79] Buckley S., *Sun up to sun down,* New York : MacGraw-Hill, 1979.

[Catarci95] Catarci T., Costabile M. et Maristella M., *Visual Metaphors for Interacting with Databases*, SIGCHI Bulletin, avril 1995.

[Chamlers95] Chamlers D., French R. et Hofstadter D., *High-Level Perception, Representation and Analogy: A Critique of Artificial-intelligence Methodology*, Dans Fluid Concepts and Creative Analogies, Douglas Hofstadter, The Penguin Press, Basic-Books, 1995, pages 169 – 193.

[Chandy88] Chandy K. M. and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, New York, 1988.

[Cox92a] Cox K. et Roman G.-C., *Abstraction in Algorithm Animation*. Dans les proceedings de 1992 Workshop on Visual Languages, Seattle, WA, septembre 1992.

[Cox92b] Cox K. et Roman G.-C., *Experiences with the PAVANE Program Visualization Environment*, Rapport Technique WUCS-92-40, Washington University, St Louis, MO, octobre 1992.

[Cox94] Cox K. et Roman G.-C., *A Characterization of the Computational Power of Rule-Based Visualization,* Journal of Visual Languages and Computing, vol. 5, N°1, janvier 1994, pages 5-27.

[Delisle84] Delisle N, Menicosy D. et Schwartz M., *Viewing a programming environment as a single tool,* SIGPLAN Notices, Vol. 19, N°5, mai 1984, pages 49-56.

[Domingue92] Domingue J. et Eisenstadt M. *A new metaphor for the graphical explanation of forward-chaining rule execution*. In M. Eisenstadt, M. Keane, and T. Rajan (Eds.) Novice programming environments: explorations in human-computer interaction and artificial intelligence. London, Lawrence Erlbaum Associates, 1992.

[Dumeur94] Dumeur R., *Synthèse de comportements individuels et collectifs par Algorithmes Génétiques*, Thèse de Doctorat à l'Université de Paris 8, janvier 1994.

[Eco80] Eco U., *Le signe*, Editions Labor, Bruxelles, 1980.

[Eco92] Eco U., *Les limites de l'interprétation,* Editions Grasset et Fasquelle, Paris, 1992.

[Eisenstadt87] Eisenstadt M. et Brayshaw M., *Graphical Debugging with the Transparent Prolog Machine (TPM).* Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Milan, Italy, 1987.

[Eisenstadt88] Eisenstadt M. et Brayshaw M., *The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming*. Journal of Logic Programming, 5 (4), pages 277-342, 1988.

[Eisenstadt89a] Eisenstadt M. et Brayshaw M., *An integrated textbook, video, and software environment for novice and expert Prolog programmers*. Dans E. Soloway and J. Spohrer (Eds.), Understanding The Novice Programmer. Hillsdale, N.J., Lawrence Erlbaum Associates, 1989.

[Eisenstadt89b] Eisenstadt M. et Brayshaw M., *AORTA diagrams as an aid to visualising the execution of Prolog programs*. In A. Kilgour (Ed.), Graphics Tools for Software Engineering. Cambridge, U.K., Cambridge University Press, 1989.

[Eisenstadt90] Eisenstadt M. et Brayshaw M., *A fine-grained account of Prolog execution for teaching and debugging*. Instructional Science., 19(4/5),1990, pages 407-436.

[Eisenstadt91] Eisenstadt M., Brayshaw M. et Payne J., *The Transparent Prolog Machine: visualising logic programs*. Dordrecht, Kluwer, 1991.

[Eisenstadt92a] Eisenstadt M., *Design Features of a Friendly Software Environment for Novice Programmers*. Dans M. Eisenstadt, M. Keane, and T. Rajan (Eds.) Novice programming environments: explorations in human-computer interaction and artificial intelligence. London, Lawrence Erlbaum Associates, 1992.

[Eisenstadt92b] Eisenstadt M. et Lewis M.W., *Novice Programmers' Syntax Errors: Causes and Cures*. Dans M. Eisenstadt, M. Keane et T. Rajan (Eds.) Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence. London, Lawrence Erlbaum Associates, 1992.

[Eisenstadt97a] Eisenstadt M., *"My hairiest bug" war stories*. Comm. ACM 40 (4), avril 1997.

[Eisenstadt97b] Eisenstadt M. et Brayshaw M., *The truth about Prolog execution.* In J. Stasko, J. Domingue, M.H. Brown et B.A.Price (Eds.) Software visualization: programming as a multimedia experience. Cambridge, MA, MIT Press, 1997.

[Fernandez75] Fernandez D., *Eisenstein*, Ramsay Poche Cinéma, 1975.

[Falkenhainer86] Falkenhainer B., Forbus K. et Gentner D., *The Structure Mapping Engine: Algorithm and examples*. Artificial Intelligence, N°41, 1989, pages 1 – 63.

[Ford93] Ford L., *How Programmers visualize programs*, Rapport Technique R271, Department of Computer Science, Université Exeter, UK, 1993.

[French95] French R., *The Subtlety of Sameness - A Theory and Computer Model of Analogy-Making*, MIT Press, MA., 1995.

[Fry95] Fry C. et Lieberman H., *Programming as Driving: Unsafe at Any Speed?* Demonstration, ACM Conference on Computers and Human Interface [CHI-95], Denver, avril 1995.

[Gentner87] Gentner D., Falkenhainer B. et Skorstad J., *Metaphor: The good, the bad and the ugly.* Actes de Third Conference on Theorical Issues in Natural Language Processing, Las Cruces, NM, janvier 1987, pages 155 – 159.

[Gentner89] Gentner D. *The mechanisms of analogical learning*, Dans Vosniadou S. et Ortony A. (eds.), Similarity and Analogical Reasoning, Cambridge University Press, MA, 1989, pages 199 - 241.

[Gerstendörfer87] Gerstendörfer M. and Rohr G., *Which task in which representation on what kind of interface.* Human-Computer Interaction, INTERACT '87, IFIP 1987.

[Glinert84] Glinert E. et Tanimoto S., *Pict : An Interactive Graphical Programming Environment*, IEEE Computer, Vol 17, N°11, novembre 1984, pages 7 – 25.

[Glinert90a] Glinert E., *Nontextual Programming Environments*, Dans Shi-Kuo Chang (ed.), Principles of Visual Programming Systems, Prentice-Hall International Editions, Englewood Cliffs, NJ, 1990, pages 144 – 231.

[Goldberg83] Goldberg A. et Robson D., *Smalltalk-80 : the language and its implementation*, Addison-Wesley, Reading, MA, 1983.

[Graham92] Graham S., Harrison M. et Munson E., *The Proteus Presentation System*, Software Engineering Notes, Vol. 17, N°5, décembre 1992, pages 130-138.

[Greussay76] Greussay P., *VLISP : Structures et extensions d'un système Lisp pour mini-ordinateur*, RT 20-76, Département Informatique, Université Paris 8, janvier 1976.

[Greussay77] Greussay P., *Contribution à la définition interprétative et à l'implémentation des Lambda-langages*, Thèse de doctorat d'Etat N°78-2, novembre 1977.

[Greussay82] Greussay P., *Le système VLISP-UNIX*, Département d'Informatique, Université Paris 8, février 1982.

[Hall79] Hall E., *Au-delà de la culture*, Point Seuil, Paris, 1979.

[Harber94] Harber H., Ioanndis Y. et Livny M., *Foundations of Visual Metaphors for Schema Display*, Journal of Intelligent Information Systems, N°3, 1994, pages 1 – 38.

[Hirakawa86] Hirakawa M., Monden N., Yoshimoto I., Tanaka M. et Ichikawa T., *HI-VISUAL: A Language Supporting Visual Interaction in Programming,* Dans S. K. Chang, T. Ichikawa, et A. Ligomenides (eds), Visual Languages, Plenum Press, 1986, pages 233-259,

[Hirakawa90] Hirakawa M., Tanaka M. et Ichikawa T., *An Iconic Programming System, HI-VISUAL,* IEEE Transactions on Software Engineering, Vol. 16, No. 10, octobre 1990, pages 1178-1184.

[Holt96] Holt R. C. et Pak J., *GASE: Visualizing Software Evolution-in-the-Large*. WCRE 96: Working Conference on Reverse Engineering, Monterey, novembre 1996.

[Hofstadter85] Hofstadter D., *Gödel, Escher et Bach*, InterEditions, Paris, 1985.

[Hofstadter95] Hofstadter D. et Mitchell M., *The Copycat Project: A Model of Mental Fluidity and Analogy-making*, Dans Fluid Concepts and Creative Analogies, Douglas Hofstadter, The Penguin Press, BasicBooks, 1995, pages 205 – 267.

[Hoare62] Hoare C. A. R., *QuickSort*, Computer Journal N°5, 1962, pages 10-15.

[Ichikawa87] Ichikawa T. et Hirakawa M., *HI-VISUAL: Toward Realization of user-friendly Iconic Programming,* Proc., Fall Joint Computer Conference (FJCC'87), octobre 1987, pages 129-137.

[Johnson77] Johnson S. C., *YACC: Yet Another Compiler-Compiler*. UNIX Manual, vol. 2B, 1977.

[Judge94] Judge A., *Les métaphores comme véhicules transdisciplinaires de l'avenir*, Dans *L'homme, la science et la nature, regards transdisciplinaires*. Coll. Science et Conscience. Ed. Le Mail, juin 1994, pages 168 – 204.

[Kaiser88] Kaiser G., Feiler P. et Popovich S., *Intelligent assistance for software development and maintenance*, IEEE Software, Vol. 5, N°3, mai 1988, pages 40-45.

[Kado92] M. Kado, M. Hirakawa, and T. Ichikawa, "HI-VISUAL for Hierarchical Development of Large Programs," Proc., IEEE Workshop on Visual Languages (VL'92), septembre 1992, pages 48-54.

[Kamada89] Tomihisa Kamada, Visualizing *Abstract Objects and Relations, A Constraint-Based Approach*. World Scientific, Singapore, 1989.

[Kamada91] Tomihisa Kamada et Satoru Kawai, *A General Framework for Visualizing Abstract Objects and Relations*. ACM Transactions on Graphics, 10(1) :1-39, janvier 1991.

[Kehoe96] Kehoe C. et Stasko J., *Using Animation to Learn about Algorithms: An Ethnographic Case Study*, Georgia Institute of Technologie, Technical Report GIT-GVU-96-20, 1996.

[Kernighan88] Kernighan B. et Ritchie D., *The C Programming Language*, deuxième édition, Prentice Hall Software Series, 1988.

[Koike95a] Hideki Koike, *Fractal Views: A Fractal-Based Method for Controlling Information Display,* ACM Transaction on Information Systems, Vol. 13, No. 3, ACM, juillet1995, pages 305 - 323.

[Koike95b] Hideki Koike et Manabu Aida, *A Bottom-Up Approach for Visualizing Program Behavior*, Proc. du 11th IEEE International Symposium on Visual Languages, IEEE/CS, 1995, pages 91-98.

[Kopache90] Kopache M. E. et Glinert E. P., *C²: A Mixed Textual/Graphical Environment for C*, Dans Ed. Ephraim Glinert (ed), Visual Programming Environments: Paradigms and Systems, IEEE, Los Alamitos, 1990, pages 305 – 312.

[Kyrou85] Kyrou A., *Le surréalisme au cinéma*, Ramsay Poche Cinéma, 1985.

[Lakoff80] Lakoff G. et Johnson M., *Metaphor we live by*, The University of Chicago Press, 1980.

[Lakoff93] Lakoff G., *Contemporary theory of metaphor*, Dans Metaphor and Thought, ed. Andrew Orthony, Cambridge University Press, 1993, pages 202 - 251.

[Le Corbusier25] Le Corbusier 1925, *Urbanisme*, Editions Vincent, Fréal & Cie, Collection de « l'esprit nouveau », 1966.

[Le Guern75] Le Guern M., *Sémantique de la métaphore et de la métonymie*, Edition Larousse Université, Col. Langue et Langage, Paris, 1975.

[Levin77] Levin S., *The Semantics of Metaphor*, The Johns Hopkins Press, Baltimore, 1977.

[Lieberman93] Lieberman H., *Tinker*: *A programming by demonstration system for beginning programmers*. Dans A. Cypher, Ed., *Watch What I Do: Programming by Demonstration*, MIT Press, 1993.

[Lieberman95] Henry Lieberman et Christopher Fry, *Bridging the Gap Between Code and Behavior in Programming*, ACM Conference on Computers and Human Interface [CHI-95], Denver, avril 1995.

[Lieberman97] Henry Lieberman et Christopher Fry, *ZStep 95, A Reversible, Animated Source Code Stepper*, Dans John Stasko, John Domingue, Blaine Price, Marc Brown, (eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, 1997. Pages 276-292.

[Madsen94] Madsen K., *A Guide to Metaphorical Design*, Communication of the ACM, Vol. 37, N° 12, décembre 1994, pages 57 – 62.

[Makigushi30] Makigushi T., *Education for Creative Living: ideas and proposals*, (première publication en 1930), trad. Alfred Birnbaum, Iowa State University Press / Ames, 1991.

[Miller95] Miller E. C., Kado M., Hirakawa M. et Ichikawa T., *HI-VISUAL as a User-Customizable Visual Programming Environment*, Proc., 1995 IEEE Symposium on Visual Languages (VL'95), octobre 1995, pages 107-113.

[Mitchell90] Mitchell M., *Copycat: A computer model of high-level perception and conceptual slippage in analogy-making*, Mémoire de Doctorat, Université du Michigan, 1990.

[Mitchell93] Mitchell M., *Analogy-Making as Perception - A Computer Model*, MIT Press, MA., 1993.

[Miyashita92] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi, Akinori Yonezawa et Tomihisa Kamada. *Declaractive Programming of Graphical Interfaces by Visual Examples*. Dans Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92), vol. 5, novembre 1992, pages 107-116.

[Miyashita94] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi et Akinori Yonezawa. *Interactive generation of graphical user interface by multiple visual examples*. Dans Proceedings of the ACM Symposium on User Interface Software and Technology, vol. 7, 1994, pages 85-94.

[Morrison95] Morrison C. et Dietrich E., *Structure Mapping vs. High-level Perception: The Mistaken Fight Over The Explanation of Analogy,* Dans Cognitive Science Society Conference, 1995.

[Mulholland97] Mulholland P. et Eisenstadt M., *Using Software to Teach Computer Programmingf : Past, Present and Future*, Dans J. Stasko, J. Domingue, M. Brown et B. Price (eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, Cambridge, Mass., 1997, pages 399 – 408.

[Muller93] Hausi A. Muller, O.A. Mehmet, S.R. Tilley, and J.S. Uhl, *A Reverse Engineering Approach to Subsystem Identification*, Software Maintenance and Practice, Vol 5, 1993, pages 181 - 204.

[Myers83] Myers B., *Incense : A System for Displaying Data Structures*, Dans Proceedings of ACM SIGGRAPH'83, juillet 1983, pages 115 - 125.

[Myers96] Myers B., *A Brief History of Human Computer Interaction Technology*, Rapport Technique CMU-CS-96-163, Carnegie Mellon University, décembre 1996.

[Najork91] Najork Marc et Kaplan Simon, *The Cube Language*. Dans IEEE Workshop on Visual Languages, Kobe, Japon, 1991, pages 218 - 224.

[Najork92] Najork Marc et Kaplan Simon, *A Prototype Implementation of the Cube Language*. Dans IEEE Workshop on Visual Languages, Seattle, WA, 1992, pages 270 - 272.

[Najork93] Najork Marc, *Programming in Three Dimensions*. Ph.D. Thesis. Rapport Technique UIUCDCS-R-93-1838, Dept. of Computer Science, Univ. of Illinois, octobre 1993.

[Nassi73] Nassi I. et Shneiderman B., *Flowchart Techniques for Structured Programming*, ACM SIGPLAN Notices, Vol. 8, N° 8, août 1973, pages 12 – 26.

[Ninio89] Ninio J., *L'empreinte des sens*, Edition Odile Jacob, Paris, 1989.

[Notkin85] Notkin D., Ellison R., Kaiser G., Kant E., Haberman A., Ambriola V. et Montanegero C., *Special Issue on the GANDALF Project*, Journal of Systems and Software, Vol. 5, N°2, Mai 1985.

[Orthony93] Orthony A., *Metaphor, language and thought,* Dans Metaphor and Thought, ed. Andrew Orthony, Cambridge University Press, 1993, pages 1 - 16.

[Petre92] Petre M. et Price B., *Why Computer Interfaces Are Not Like Paintings : the user as a deliberate reader*, Dans Proc. East-West HCI'92: : The St. Petersburg International Conference on Human-Computer Interaction, Volume I, Russie, 1992, pages 117 – 224.

[Petre97] Petre M., Blackwell A. et Green T., *Cognitive Questions in Software Visualization*. Dans J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, Cambridge, Mass., 1997, pages 453 – 480.

[Petrie93] Petrie H. et Oshlag R., *Metaphor and learning*, dans Dans Metaphor and Thought, ed. Andrew Orthony, Cambridge University Press, 1993, pages 610 – 620.

[Pineda88a] Pineda L. A., *A compositional semantics for graphics*. Dans Proceedings of Eurographics '88. Elsevier Science, Amsterdam, 1988.

[Pineda88b] Pineda L. A. et Andchater N., *GRAFLOG: Programming with interactive graphics and Prolog*. Dans Proceedings of CG International '88. Springer, New York, 1988, pages 469-478.

[Pineda88c] Pineda L. A., Klein E. et Andlee J. *GRAFLOG: Understanding drawings through natural language*. Computer Graphic Forum 7, 2 juin 1988, pages 97-103.

[Ploix93] Ploix D., *VISAL : Un système de visualisation statique et dynamique de programmes Lisp*, Mémoire de DEA, Rapport de recherche, Département d'Informatique, RR 931010, Université Paris 8, octobre 1993.

[Ploix96a] Ploix D., *Xbvlisp Aujourd'hui*, Rapport technique, Département d'Informatique, Groupe APARU, APARU960301, Université Paris 8, mars 1996.

[Ploix96b] Ploix D., *Building Program Metaphors,* Dans les actes du Workshop Psychology of Programming Interest Group (PPIG'96), Matlock, UK, septembre 1996. pages 125-129.

[Ploix97] Ploix D., *Observation de Programmes par la Combinaison d'Analogies*, Actes de la conférence Intelligence Artificielle et Complexité, St Denis, France, février 1997, pages 150-156.

[Ploix98] Ploix D., *The Zeugma Programming Environment*, Rapport de Recherche, Département d'Informatique, RR980123, Université Paris 8, janvier 1998.

[Pong83] Pong M. et Ng N., *PIGS – A System for Programming with Interactive Graphical Support*, Software – Practice and Experience, Vol. 13, N°9, septembre 1983, pages 847 – 855.

[Price97] Price B., Baecker R. et Small I., *An Introduction to Software Visualization*, Dans J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), Software Visualization : Programming as a multimedia experience , MIT Press, 1997, pages 3 – 28.

[Reiss84] Reiss S., *Graphical Program Development with PECAN Program Development System*, Dans Proceedings de ACM SIGSOFT/SIGPLAN Symposium sur Practical Software Development Environments, avril 1984.

[Reiss85] Reiss S., *PECAN : Program Development System that Support Multiple Views,* IEEE Transactions on Software Engineering, Vol. SE-11, mars 1985, pages 276-284.

[Reiss90a] Reiss S., *Interacting with the FIELD Environment*, Software Practice and Experience, Vol. 20, N°S1, juin 1990, pages 89-115.

[Reiss90b] Reiss S., *Connecting Tools Using Message Passing in the FIELD Environment*, IEEE Software, Vol. 7, N°4, juillet 1990, pages 57-67.

[Reiss94a] Reiss S., *FIELD : A Friendly Integrated Environment for Learning and Development,* Kluwer, Norwell, MA. 1994

[Reiss94b] Reiss S., *3-D Visualization for Program Information*, SIGCHI'94 Software Visualization Workshop, Boston, MA, 1994.

[Reiss97] Reiss S., *Visualization for Software Engineering – Programming Environments.* Dans J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), Software Visualization : Programming as a multimedia experience, MIT Press, 1997, pages 259-276.

[Reiter89] Reiter R. et Mackworth A., *A Logical Framework for Depiction and Image Interpretation*, Artificial Intelligence, Vol 41, 1989, pages 125 – 155.

[Rich81] Rich C. et Waters R., *A formal representation for plans in the Programmer's Apprentice*, Dans Proc. 7th Joint Conf. Artificial Intelligence, Vancouver, BC, Canada, 1981, pages 1044 – 1052.

[Roman89] Roman G.-C. et Cox K., *A Declarative Approach to Visualizing Concurrent Computations.* Computer, 22(10), 1989, pages 25-36.

[Roman90] Roman G.-C. et Cunningham H.C., ``Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency,'' IEEE Transactions on Software Engineering Vol. 16, No. 12, décembre 1990, pages 1361- 1373.

[Roman92] Roman G.-C., Cox K., Wilcox C. et Plun J., *PAVANE : A System for Declarative Visualization of Concurrent Computation*, Journal of Visual Languages and Computing, Vol. 3, N°1, janvier 1992, pages 161-193.

[Roman97] Roman G.-C., *Declarative Visualization*, Dans J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), Software Visualization : Programming as a multimedia experience, MIT Press, 1997, pages 173-186.

[Sendoya92] Ernesto Sendoya, *Etude et réalisation d'une couche à objets graphiques pour bVLISP*, Mémoire de D.E.A., Université Paris 8, octobre 1992.

[Soloway84] Woloway E. et Ehrlich K, *Empirical studies of programming knowledge,* IEEE Transactions on Software Engineering, vol. 10, N°5, 1984, pages 595 – 609.

[Shu85a] Shu N., *Visual Programming Languages : A Dimensional Analysis*, Dans Proc. De IEEE International Symposium on New Directions in Computing, août 1985, pages 326 – 334.

[Shu85b] Shu N., *FORMAL : A Forms – Oriented and Visual-Directed Application System,* IEEE Computer, Vol 18, N°8, août 1985, pages 38 – 49.

[Stepich88] Stepich D. et Newby T., *Analogical instruction within the information processing paradigm : Effective means to facilitate learning*, Instructional Science, Vol. 17, 1988, pages 129 – 144.

[Stasko89] Stasko J., *TANGO : A Framework and Systems for Algorithm Animation,* Thèse de Phd, Brown University, Providence, RI, mai 1989.

[Stasko92] Stasko J., *Animating Algorithms with XTANGO*, SIGACT News, Vol. 23, N°2, printemps 1992, pages 66-71.

[Stasko93] Stasko J., *A Methodology for Building Application-Specific Visualization of Parallel Programs.* Journal of Parallel and Distributed Computing, vol. 18; N°2, juin 1993, pages 258-264.

[Stasko97a] Stasko J., *Using Student-Built Algorithm Animations as Learning Aids*, Proceedings du ACM Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, CA, Février 1997, pages 25-29.

[Stasko97b] Stasko J., *Building Software Visualizations through Direct Manipulation and Demonstration*. Dans J. Stasko, J. Domingue, M. Brown, and B. Price (eds.), Software Visualization : Programming as a multimedia experience, MIT Press, 1997, pages 173-186.

[Takahashi91] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa et Tomihisa Kamada, *A general Framework for Bidirectional Translation between Abstract and Pictorial Data*. Dans Proceedings of the ACM Symposium on User Interface Software and Technology, vol. 4, novembre 1994, pages 165-174.

[Takahashi94] Shin Takahashi, Ken Miyashita, Satoshi Matsuoka, et Akinori Yonezawa. *A Framework for constructing animations via declarative mapping rules.* Dans Proceedings of the 1994 IEEE Symposium on Visual Languages, vol. 10, 1994, pages 314-322.

[Takahashi95] Shin Takahashi, Satoshi Matsuoka, Ken Miyashita, Hiroshi Hosobe, Akinori Yonezawa et Tomihisa Kamada. *A Constraint-Based Approach for Visualization and Animation*. In Proceedings of international Workshop on Constraints for Graphics and Visualization (CGV'95), septembre 1995, pages 103-117.

[Taylor89] Taylor R. et al., *Foundations for the Arcadia Environment Architecture*, SIGPLAN Notices, Vol. 24, N°2, février 1989, pages 1-13.

[Tufte83] Tufte E., *The Visual Display of Quantitative Information,* Graphics Press, Cheshire, CT, 1983

[Ungar97] Ungar D., Lieberman H. et Fry C., *Debugging and the Experience of Immediacy*, Communications of the ACM, Vol. 40, No. 4, avril 1997.

[vanDam98] van Dam A., *Frontiers in User-Computer Interaction*, EUROGRAPHICS'98, Portugale, septembre 1998.

[Vinge87] Vernor Vinge, *True Names*, Bean Books, N.Y. 1987.

[Vosniadou89] Vosniadou S., *Analogical reasoning in knowledge aquisition*, Dans Vosniadou S. et Ortony A. (eds.), Similarity and Analogical Reasoning, Cambridge University Press, MA, 1989, pages 413 - 437.

[Wertz76] Wertz H., *Understanding Lisp programs is improving Lisp programs*, Dans Informatik Fachberichte N°5, ed J. Neuhold, Springer Verlag, Heidelberg, Wien, New-York, 1976, pages 427-441.

[Wertz78] Wertz H., *Un système de compréhension, d'amélioration et de correction de programmes incorrects*, Université Paris 6, thèse de 3ème cycle, juillet 1978.

[Wertz79] Wertz H., *A System to improve incorrect programs*, dans Proc. 4th International Conference on Software Engineering, München (RFA), septembre 1979, pages 286 – 293.

[Wertz83] Wertz H., *Etude, Réalisation et Evaluation d'un Environnement de Programmation Utilisant des Représentations Multiples pour le Développement Continu de Logiciels Très Evolués*, Thèse de Doctorat d'Etat, 1983.

[Wertz93] Wertz H. et Ploix D., *« Liberating programming from the ASCII view »*, Rapport de Recherche, Département d'Informatique, Laboratoire d'Intelligence Artificielle, RR 930702, Université Paris 8. juillet 1993

[Williams88] Williams P., *Going west to east : Using metaphors as instructional tools*. Journal of Children in Contemporary Society, Vol 20, 1988, pages 127 – 129.

[Wills92] Wills L., *Automated Program Recognition by Graph Parsing*, Rapport Technique N° 1358 du Laboratoire d'Intelligence Artificielle du MIT, juillet 1992.

Annexe I Code source de Zeugma

Annexe I.1 Fichier de chargement du système : Zeugma.vlisp

```
;
; Construction du système
;
(careful nil)
; répertoires
(setq ZEUGMA-PARENT    (or (getenv "ZPARENT") "/usr/people/damien/work/"))
(setq ZEUGMA-DIRECTORY (strcat ZEUGMA-PARENT "Zeugma/"))

; librairie utilisées par Zeugma
(mapc '(X-Menus GL-drive xfile widget i-val)
     (lambda (f) (princ ":") (eval `(include ,(strcat VLISPDIR f)))))

; définition de Zeugma
(mapc '(Zmpv Zmeta-editor Zprolog
   ZORS Zfunction-info
   Zgraphiques Zanalyses
   Zsuivit     Zin-out Zdump
   Zfile Zuser
   Zhelp Zdata Zmenus
   Zprc-compile)
     (lambda (f) (princ ".") (eval `(include ,(strcat ZEUGMA-DIRECTORY f)))))

; définition des parcours initiaux
(mapc '(Zprc-def Zprc-analyses Zdefs)
     (lambda (f) (princ "-") (eval `(include ,(strcat ZEUGMA-DIRECTORY f)))))

; lancement de la construction des menus pour les parcours définis durant le chargement de Zeugma
(PRC-Make-Menus)
;
; Fonction de lancement de Zeugma
;
(de Zeugma (obj w)                                                    Zeugma
  (if (and (null obj)
      (boundp 'root-object)
      root-object) (setq obj root-object))
  (if obj (put obj 'obj obj))
  (Create-Stepper-Widget)
  (ifn (and (boundp 'MPV-Root-Window)
       (iswidget MPV-Root-Window)) (MPV-InitWindow)))
```

Annexe I.2 Définition des données globales : Zdefs.vlisp

```
;
; Fichier de definitions des globales utiliées dans Zeugma
;
(ifn (boundp  'Objects-List) (setq Objects-List nil)) ; liste des ORS definis

                                                ; ******* ; ;
                                                ; Donnees ; ;
                                                ; ******* ; ;
;
; repertoires :
;
(setq bitmap-directory       (strcat ZEUGMA-PARENT "bitmap/"))   ; repertoire des bitmaps
(setq default-dump-directory (strcat ZEUGMA-PARENT "dump/"))
(setq default-data-directory (strcat ZEUGMA-PARENT "Meta-Donnees/"))
(setq default-lisp-directory (strcat ZEUGMA-PARENT "lisp-expls/"))
(setq Zeugma-Help-File       (strcat ZEUGMA-DIRECTORY "Zeugma-Help"))
;
; Classes de widgets
```

```
;
(setq W-AShell   "ApplicationShell") (setq W-Box      "awBox")
(setq W-Command  "awCommand")        (setq W-Dialog   "awDialog")
(setq W-Draw     "Drawxbvl")         (setq W-Form     "awForm")
(setq W-Label    "awLabel")          (setq W-Text     "awText")
(setq W-Tree     "awTree")           (setq W-BTree    "brTree")
(setq W-Viewport "awViewport")       (setq W-Xbvlisp  "Xbvlisp")

(setq W-ClBck    "callback")    (setq W-Btn1     "<Btn1Down>")
(setq W-Btn2     "<Btn2Down>")  (setq W-Btn3     "<Btn3Down>")

(setq W-BColor   "#272") (setq W-ComFg    "#000") (setq W-ComBg    "#888")
(setq W-RColor   "#f0f")  ; code en cours d'exécution
(setq W-MColor   "#f00")  ; code marqué
(setq W-MRColor  "#ff0") ; code marqué en cours d'exécution
(setq W-Black    "#000")
(setq W-White    "#fff")
(setq W-Colors '("#0ff" "#0ee" "#0dd" "#0cc" "#0bb" "#0aa" "#099" "#088"
                 "#077" "#066" "#055" "#044" "#033" "#022" "#011" "#000"))

(setq Meta-Font  "a14")  (setq default-font "a14") (setq FontHeight 14) (setq FontWidth 9)
(setq Zinfo-font "7x13") (setq Zinfo-font-size-y 13) (setq Zinfo-font-size-x 7)
;
; Taille initiale des fenetres
;
(setq XW-Gene   520) (setq YW-Gene   100)
(setq XW-Nom    225) (setq YW-Nom    100)
(setq XW-Points 100) (setq YW-Points 100)
(setq XW-Vars   100) (setq YW-Vars   100)
(setq XW-Cond   600) (setq YW-Cond   600)
;
(setq main-bvlisp (xwinp))
; pour la gestion des objets graphiques
(setq scale-factor        1000)          ; multiplicateur des position pour eviter les prob. de precision
(setq init-list  0)                      ; premier objet
(setq object-list 0)
(setq curent-object-list 0)
(setq curent-to-object 0)
(setq curent-list 0)
(setq max-list 0)                  ; objet courant et maximum...
(setq gl-Windows nil)                ; pile des fenetres ouvertes...
(setq list-range 15000)
; pour le contrôle de la génération des analogies
(setq Flow-Uniq nil)
(setq Flow-Verbose nil)
(setq Flow-Curent-Instruction nil)
(setq Flow-Curent-Method nil)
;
; Variables de l'analyse numérique
;
(setq  Composition-Variables '(loop conds test local global modif
                                    num str lst plst io x gl param lambda
                                    total 1 2 3 4 5 6 7 8 9 maxlg))
;
(setq ga-cond nil)
(setq GA-Build-Stack nil)
(setq ga-build nil)
(setq ga-p-build t)
(setq Curent-Object-Transformation nil)
;
; Mio-dump.vlisp
;
(setq DD-Root-Window nil)
(setq exe-dynamic-dump nil)
(setq exe-dump-file nil)
(setq dump-file nil)
(setq dump-root nil)
(setq dump-data-file nil)
(setq dump-include-file nil)
(setq dump-func-file nil)
(setq dump-func nil)
(setq dump-initial nil)
(setq dump-stack nil)
(setq dump-f-list 0)
(setq dump-l-list 0)
(setq is-dump-back nil)
```

```
(setq is-in-dump nil)
(setq dump-step 0)
;
; Mgraphiques.vlisp
;
 ; variables utilises pour la construction des graphiques
 ;
(setq GA-Trans-Stack nil)        ; pile des transformations
(setq GA-Name-Stack nil)         ; pile des nom graphiques
(setq GA-Stack nil)              ; pile de construction
;
; gestion de la pile de construction
;
(defmacro GA-Push (value)  `(newl GA-Stack ,value))                    GA-Push
(defmacro GA-Pop  ()       '(nextl GA-Stack))                          GA-Pop
;
;
; pour l'éditeur de représentations
(setq GA-G-Tree nil)  (setq GA-G-Parent nil)  (setq GA-G-Done nil)
(setq GA-Erase nil)   (setq GA-Root nil)      (setq GA-Tree nil)
(setq GA-Tree-Root nil)
;
; pour le calcul de l'emplacement exact a partir des transformations par matrices
;
(setq GA-trans-x 0) (setq GA-trans-y 0) (setq GA-trans-z 0)
(setq GA-rot-x 0)   (setq GA-rot-y 0)   (setq GA-rot-z 0)
(setq GA-scale-x 0) (setq GA-scale-y 0) (setq GA-scale-z 0) (setq GA-scale-w 0)
;
; position de l'element
(setq GA-X 0) (setq GA-Y 0) (setq GA-Z 0) (setq actuel nil) (setq GA-Values nil)
; position du pere
(setq p-x 0) (setq p-y 0) (setq p-z 0) (setq pere nil)      (setq p-values nil)
; delta entre pere et fils
(setq GA-Delta nil)
; utilise pour le mouvement unique et le deplacement des elements
(setq exe-translation '(GLtranslate 0 0 0))
;
; pour la gestion des patterns graphiques
;
(setq Node-Bitmap (strcat bitmap-directory "node"))
(setq Obj-Bitmap  (strcat bitmap-directory "gr-obj"))
(setq Node-Obj-Bitmap (strcat bitmap-directory "node-obj"))
(setq Tree-Bitmap (strcat bitmap-directory "s-tree"))
(setq Kill-Bitmap (strcat bitmap-directory "gr-kill"))
;
(setq Match-Type '((partiel . "Partiel = structure même inclue")
                   (exact   . "Exact   = même structure       ")))
;
(setq Base-Colors '((15 0 0)  (0 15 0)  (0  0 15)
                    (15 0 15) (0 15 15) (15 15 0)))
(setq N-Base-Colors 6)
(setq ME-Pattern-List nil)
```

Annexe I.3 Définition des menus : Zmenus.vlisp

```
                                        ; ***************************** ; ;
                                        ; Définition des Menus de Zeugma ; ;
                                        ; ***************************** ; ;
;
; Menu de l'interface principale
;
(setq MPV-Menu '((("new.bmp") .      (() . MPV-New-Root)) ;  Zmpv
                 (("load-lsp.bmp") . (() . Load-Lisp-File)) ; Zfile...
                 (("load.bmp") .     (() . Get-File-Name)) ; Zfile...
                 (("save.bmp") .     (() . MPV-Save)) ; menu-g "Sauvegarde des objets definis"
;                (("status.bmp") .   (() . Z-Status)) pour la phase de débug
                 (("g-flow.bmp") .   (() . ME-Init-Window)) ; Zmeta-editor
                 (("exe-dump") .     (() . DD-Load))
                 (("gr-exe.bmp") .   (() . Initialize-Follow-GL)) ; Zgraphiques
                 (("dismiss.bmp") .  (() . MPV-Quit-Zeugma)) ;
                 (("help.bmp") .     (() . MPV-Help))))
```

```
;
; Etats de Zeugma
;
(setq MPV-Status '("Répertoires          "
                   ("Dumps                " . (dump . Get-String-Value))
                   ("Définition des analogies" . (data . Get-String-Value))
                   ("Programmes Lisp      " . (lisp . Get-String-Value))
                   ("Validation des répertoires" . (() . Valid-directories))
                   "Suivit de l'exécution"
                   ("Pas à Pas          " .   (io-text . Select-Yes-No-Value))
                   "DEBUG"
                   ("Génération Verbeuse" . (is-verbose . Select-Yes-No-Value))
                   ("Sauvegarde         " . (() . Z-Save-Definitions))))
;
; Création d'une nouvelle représentation
;
(setq MPV-New-Root '(("Nom de l'objet racine     " . (nom . MPV-New-Name))
                     ("Domaine d'application     " . (domain . MPV-New-Select-Domain))
                     ("Titre de la représentation" . (titre . Get-String-Value))
                     (("ok.bmp") . (() . MPV-New-Confirm))
                     (("dismiss.bmp") . (() . MPV-New-Cancel))
                     (("help.bmp") . (() MPV-Help-new-root))))

;
; Menu des propriétés des ORS racine
;
(setq Initial-Menu '((("new-ors.bmp") .              (() . Create-Object)) ; menu
                     (("ors.bmp") .                  (() . Edit-One-Object))
                     (("o-flow.bmp") .               (() . Draw-ORS-Graph))          ; graphiques "Graphe des objets   "
                     (("dismiss.bmp") .              (() . Quit-Spec-Menu)) ;
                     (("help.bmp") .                 (() . Root-ORS-Help))
                     ("Description de l'ORS" .   (() . Describe-Object))
                     ("Nom de l'ORS racine       " . (name . Select-Object)) ; menu
                     ("Lien avec une fonction    " . (func . Select-Analysed-Function)) ;
                     ("Variables initiales du FDD" . (variable . Select-DEP-Variable))
                     ("Dessin unique des elts.   " . (unique . Select-Yes-No-Value))))
;
; Menu des propriétés d'un ORS
;
(setq C-Menu       '(("Nom de l'objet          " . (name . Set-Object-Name))
                     "Actions liées à la construction"
                     ("Données graphiques" . (gl-data . Edit-GL-Data)) ; menu-g
                     ("Reconstruction    " . (gl-recons . Select-Yes-No-Value))
                     ("ORS Générés       " . (generation . Select-Multiple-Objects)) ; menu-g
                     "  Actions liées à l'exécution  "
                     ("Graphiques dyn. Entrée  " . (gl-dyn-in  . Edit-GL-Data)) ; menu-g
                     ("Graphiques dyn. Sortie  " . (gl-dyn-out  . Edit-GL-Data)) ; menu-g
                     ("Action graphique        " . (action-g . Select-Run-Action)) ; menu-g
                     ("Transformation graphique" . (gl-trans . Select-Run-Transformation))
                     ("Attachement(s) automatiques(s)" . (() . Select-Run-Link))
                     (("dismiss.bmp") .              (() . Dismiss-C-Menu)) ;
                     (("help.bmp") .                 (() . ORS-Help))))

(setq Link-Menu '("Attachements automatiques"
                  ("A l'entrée des fonctions   " . (o-in . Select-Yes-No-Value))
                  ("A la sortie des fonctions  " . (o-out . Select-Yes-No-Value))
                  ("A l'entrée des expressions " . (i-in . Select-Yes-No-Value))
                  ("A la sortie des expressions" . (i-out . Select-Yes-No-Value))
                  ("A l'accès aux variables    " . (v-obj . Select-Variable-Link))
                  ("Test attaché aux variables " . (v-test . Enter-Variable-Test))
                  (("dismiss.bmp") . (() . Dismiss-Link-Menu))
                  (("help.bmp") . (() . Link-Help))))

(setq Variables-Link '((local . "Contexte local")
                       (global . "Contexte global")))

(setq GA-GMenu '((GA-Graph-FULL .     "            Complet          ")
                 (GA-Graph-NO-TRANS . "Sans Transformations Graphiques")))

;
; Actions à la génération
;
(setq Gen-Types '((parcour    .     "Activation :")
                  (objet      .     "ORS :")
                  (transformations . "Transformation :")))
;
```

```
; Actions a l'Exécution
;
; Pour la partie graphique
;
(setq GL-Run-Actions '((recons          . "Reconstruction                  ")
                       (eff-recons      . "Effacement et reconstruction    ")
                       (transfo         . "Transformation                  ")
                       (mv-uniq         . "Repos. Unique sans reconstruction ")
                       (mv-uniq-recons  . "Repos. Unique avec reconstruction ")
                       (nil             . "Aucun                           ")))
;
; Donnees des transformations
;
(setq T-Menu       '("Transformations" ; menu-g
                     ("Translation" . (translate . Enter-Translation))
                     ("Rotation   " . (rotate   . Enter-Rotation))
                     ("Scaling    " . (scale . Enter-Scaling))
                     ("Couleur    " . (color . Enter-Color))
                     (("dismiss.bmp") . (() . Dismiss-T-Menu))
                     (("help.bmp") . (() . Trans-Help))))
;
; Menu des fonctions
;
(setq Func-Menu '((("analyses.bmp") .          (() . Draw-Analyses-Results)) ; "Resultats des analyses"
                  (("f-flow.bmp") .            (() . FM-Function-Graph)) ; "Graphe des fonctions"
                  (("v-flow.bmp") .            (() . FM-Variables-Graph)) ; "Graphe des variables"
                  (("f-comp.bmp") .            (() . FM-Display-Program))
                  (("dismiss.bmp") .           (() . FM-Quit)) ;
                  (("help.bmp") .              (() . FM-Help))
                  ("Point d'entrée" .          (func . ()))
                  ("Fichier source" .          (file . ()))
                  "Attachements d'activités supplémentaires :"
                  ("En entrée de fonction    " . (o-in . ()))
                  ("En sortie de fonction    " . (o-out . ()))
                  ("En entrée d'expression   " . (i-in . ()))
                  ("En sortie d'expression   " . (i-out . ()))
                  ("Sur l'accès aux variables" . (v-obj . ()))))

;
; Informations sur les fonctions
;
(setq Func-Info-Menu  '("Activités supplémentaires"
                        "Sur la fonction"
                        ("Arrêt en entrée" . (s-in    . Switch-Stop))
                        ("Arrêt en sortie" . (s-out   . Switch-Stop))
                        ("ORS en entrée  " . (o-in    . Select-Object))
                        ("ORS en sortie  " . (o-out   . Select-Object))
                        "Sur les expressions de la fonction"
                        ("Arrêt en entrée" . (e-s-in  . Switch-Stop))
                        ("Arrêt en sortie" . (e-s-out . Switch-Stop))
                        ("ORS en entrée  " . (i-in    . Select-Object))
                        ("ORS en sortie  " . (i-out   . Select-Object))
                        (("repercute.bmp") . (() . Dispatch-Func-Data))
                        (("ok.bmp")        . (() . Confirm-Func-Info-Menu))
                        (("dismiss.bmp")   . (() . Dismiss-Func-Info-Menu))
                        (("help.bmp")      . (() . FM-Func-Help))))
;
; Informations sur les variables
;
(setq Var-Info-Menu '("Activités supplémentaires"
                      ("Arret lors de l'exécution" . (v-stop . Switch-Stop))
                      ("Test sur la variable     " . (v-test . Enter-Variable-Test))
                      ("Lien contextuel à la valeur" . (v-value . Select-Yes-No-Value))
                      ("ORS initial              " . (v-obj  . Select-Object))
                      (("repercute.bmp") . (() . Dispatch-Var-Data))
                      (("ok.bmp")        . (() . Confirm-Var-Info-Menu))
                      (("dismiss.bmp")   . (() . Dismiss-Var-Info-Menu))
                      (("help.bmp")      . (() . FM-Var-Help))))
;
; Information sur les objets graphiques
;
(setq Gr-Info-Menu '((("dismiss.bmp") . (() . Confirm-Info-Menu))))
;
```

```
; Procédure de placement et de récupération des données sur les
; objets tels que les fonctions ou les variables traitées dans Zeugma

; récupération de données

(de GetPrcData (prc obj what)                                                    GetPrcData
  (and (not (or (numbp obj) (stringp obj) (null obj)))
       (cdr (assoc what (or (and (listp obj) (get-car-value obj prc))
                            (and (atom obj) (cdr (assoc prc (ival obj)))))))))

(de GetData     (obj what)  (GetPrcData ga-cond obj what))                       GetData
(de GetFuncData (func what) (GetPrcData 'func func what))                        GetFuncData
(de GetVarData  (var what)  (GetPrcData 'var var what))                          GetVarData

; placement de données

(de PutPrcData (type obj what value)                                             PutPrcData
  (setq __aux__ nil)
  (cond
   ((or (numbp obj) (stringp obj) (null obj)) nil)
   ((and (listp obj) (null (setq __aux__ (get-car-value obj type))))
    (add-in-car obj type `((,what . ,value))))
   ((not (or __aux__ (setq __aux__ (cdr (assoc type (ival obj))))))
    (ival obj (cons `(,type (,what . ,value)) (ival obj))))
   ((null (setq __aux__1 (assoc what __aux__)))
    (rplacd (last __aux__) `((,what . ,value))))
   (t
    (rplacd __aux__1 value)))
  value)

(de PutData     (obj what value)  (PutPrcData ga-cond obj what value))           PutData
(de PutFuncData (func what value) (PutPrcData 'func func what value)))           PutFuncData
(de PutVarData  (var what value)  (PutPrcData 'var var what value)))             PutVarData


; et effacement...

(de DelPrcData (prc obj what)                                                    DelPrcData
  (if (and (not (or (numbp obj) (stringp obj) (null obj)))
      (setq __aux__ (assoc what (or (and (listp obj) (get-car-value obj prc))
                                    (and (atom  obj) (cdr (assoc prc (ival obj))))))))
      (rplacd __aux__ nil)))

(de DelData     (obj what) (DelPrcData ga-cond obj what))                        DelData
(de DelFuncData (obj what) (DelPrcData 'func obj what))                          DelFuncData
(de DelVarData  (obj what) (DelPrcData 'var obj what))                           DelVarData
```

Annexe I.5 Définition des aspects prédéfinis : Zprc-def.vlisp

```
                                        ; ************************** ; ;
                                        ; Définition des points de vues ; ;
                                        ; ************************** ; ;


                                        ; ********************************************************** ; ;
                                        ; point de vue génériques (pas liées à un flot particulier) ; ;
                                        ; ********************************************************** ; ;

(define-PDV Point-de-vue-Generiques
  (nom generiques)  (titre "Point-de-vue génériques          ")
  (analyse (GEN-analyse . generiques))
  (tests incond test))

(define-Test incond
  (titre "Branchement inconditionnel")
  (test-eval t)
  (next-flow flow))

(define-Test test
```

```
  (titre "Test Lisp")
  (test-read "Entrez le test Lisp")
  (test-eval (eval (GetTestData))
  (next-flow flow)))

                                        ; ******************************************************* ; ;
                                        ; points de vues liés à l'exécution des programmes :    ; ;
                                        ; permet le parcours des valeurs des données manipulées ; ;
                                        ; ******************************************************* ; ;
; élément d'initiation de l'interruption
(define-PDV Execution-element
  (nom EXE) (titre "Objet de l'exécution")
  (analyse (EXE-elem . EXE))
  (tests exe-elem-var exe-elem-func exe-elem-instr exe-entree exe-sortie))

(define-Test exe-elem
  (titre "Element - inconditionel")
  (test-eval t)
  (next-flow flow))

(define-Test exe-elem-var
  (titre "l'élément est une variable")
  (test-eval exe-is-var)
  (next-flow flow))

(define-Test exe-elem-func
  (titre "l'élément est une fonction")
  (test-eval (and (null exe-is-var) (atom exe-func))))

(define-Test exe-elem-instr
  (titre "l'élément est une expression")
  (test-eval (and (null exe-is-var) (listp exe-func))))

(define-Test exe-entree
  (titre "Entrée dans une fonction avec des paramètres non nuls")
  (test-eval (and (null exe-is-var)
                  (atom exe-func)
                  exe-in
                  (let (args exe-args)
                    (and args
                         (or (eval (car args))
                             (self (cdr args)))))))
  (next-flow flow))

(define-Test exe-sortie
  (titre "Sortie d'une fonction avec un retour non nul")
  (test-eval (and (null exe-is-var)
                  (atom exe-func)
                  exe-out
                  exe-sortie))
  (next-flow flow))

; valeur numérique
(define-PDV Execution-numerique
  (nom EXENUM) (titre "Données de l'exécution - numérique")
  (analyse  (EXE-var-value . EXE-value))
  (tests num-cmp num-zero num-mod))

(define-Test num-cmp
  (titre "Test sur une Variable numérique          ")
  (test-read "Donnez le test à effectuer")
  (test-eval (and exe-is-var (eval (GetTestData))))
  (next-flow flow))

(define-Test num-zero
  (titre "Variable numérique égale à zéro          ")
  (test-eval (and exe-is-var (= flow 0)))
  (next-flow flow))

(define-Test num-mod
  (titre "Applique une opération sur la valeur     ")
  (test-read "Donnez la modification à effectuer (valeur dans flow)")
  (test-eval exe-is-var)
  (next-flow (eval (GetTestData))))

; valeur liste
```

```
(define-PDV Execution-liste
  (nom EXELST) (titre "Données de l'exécution - liste")
  (analyse  (EXE-var-value . EXE-value))
  (tests lst-CAR lst-CDR lst-ATOM))

(define-Test lst-CAR
  (titre "Variable Liste CAR                    ")
  (test-eval (and exe-is-var (listp flow)))
  (next-flow car-flow))

(define-Test lst-CDR
  (titre "Variable Liste CDR                    ")
  (test-eval (and exe-is-var (listp flow) (cdr flow)))
  (next-flow (cdr flow)))

(define-Test lst-ATOM
  (titre "Variable Liste ATOM                    ")
  (test-eval (and exe-is-var (listp flow) (atom car-flow)))
  (next-flow car-flow))

; valeur chaine
(define-PDV Execution-chaine
  (nom EXESTR) (titre "Données de l'exécution - chaîne")
  (analyse  (EXE-var-value . EXE-value))
  (tests str-car str-cdr str-cmp str-len))

(define-Test str-car
  (titre "Premier caractère d'un variable chaîne    ")
  (test-eval (and exe-is-var (stringp flow)))
  (next-flow (strcar flow)))

(define-Test str-cdr
  (titre "Derniers caractères d'un variable chaîne  ")
  (test-eval (and exe-is-var (stringp flow)))
  (next-flow (strcar flow)))

(define-Test str-cmp
  (titre "Comparaison d'une variable chaîne         ")
  (test-read "Donnez la chaîne à comparer")
  (test-eval (and exe-is-var (strcmp flow (GetTestData))))
  (next-flow flow))

(define-Test str-len
  (titre "Longueur d'une variable chaîne            ")
  (test-read "Donnez la longueur de la chaîne")
  (test-eval (and exe-is-var (= (GetTestData (strlen flow)))))
  (next-flow flow))



                                        ; ********************************************** ; ;
                                        ; Les Point-de-vue de la composition des programmes ; ;
                                        ; nécessaires au fonctionnement de Zeugma :     ; ;
                                        ; - FDC : construction du flot de contrôle      ; ;
                                        ; - FDD : construction du flot de données       ; ;
                                        ; - COMP : étude de la composition             ; ;
                                        ; ********************************************** ; ;

; ********************************************** ; ;
; définition du Point-de-vue du flot de contrôle ; ;
; ********************************************** ; ;

(define-PDV Flot-de-controle
  (nom FDC)  (titre "Flot de contrôle              ")
  (analyse (analyse-FDC . FDC)) ; fonction de l'analyse et champ dans lequel est placé le flot à traiter
  (Point-de-vue-type "Meta-Donnees/sources/fdc.vlisp" traverse-fdc)
  (global (Flow-NLG . PRC-Make-Flow-NLG))
  (compteurs fdc-ipr fdc-ilg fdc-iilg fdc-nilg fdc-nlg)
  (tests fdc-CAR fdc-CDR fdc-ATOM))

(define-Test fdc-CAR
  (titre "Flot de contrôle - Profondeur")
  (local-var (next-nilg (setq next-nilg (length car-flow))))
  (test-eval (listp car-flow))
  (next-flow (car flow))
```

```
    (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

  (define-Test fdc-CDR
    (titre "Flot de contrôle - Largeur")
    (test-eval (cdr flow))
    (next-flow (cdr flow))
    (ilg (PRC-Next-ILG)) (iilg (1+ iilg)))

  (define-Test fdc-ATOM
    (titre "Flot de contrôle - Elément")
    (local-var (Fdc-Constructed (progn (newl Fdc-Constructed car-flow) (Set-Composition-Numeration car-
flow))))
    (test-eval (and (litatom car-flow) (not (and Flow-Uniq (member car-flow Fdc-Constructed)))))
    (next-flow car-flow)
    (iilg 1) (nilg 0))

  ; ****************************** ; ;
  ; Point-de-vue du flot de données ; ;
  ; ****************************** ; ;

  (define-PDV Flot-de-donnees
    (nom FDD) (titre "Flot de données              ")
    (analyse (analyse-FDD . FDD))
    (Point-de-vue-type "Meta-Donnees/sources/fdd.vlisp" traverse-fdd)
    (global (Flow-NLG . PRC-Make-Flow-NLG))
    (compteurs fdd-ipr fdd-ilg fdd-iilg fdd-nilg fdd-nlg)
    (tests fdd-CAR fdd-CDR fdd-VAR fdd-FUNC))

  (define-Test fdd-CAR
    (titre "Flot de données - profondeur")
    (local-var (next-nilg (setq next-nilg (length car-flow))))
    (test-eval (listp car-flow))
    (next-flow car-flow)
    (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))


  (define-Test fdd-CDR
    (titre "Flot de données - largeur   ")
    (test-eval (or (and (atom car-flow) (car (cddr flow)))
                   (and (listp car-flow) (cdr flow))))
    (next-flow (or (and (atom car-flow) (cddr flow))
                   (cdr flow)))
    (ilg (PRC-Next-ILG)) (iilg (1+ iilg)))

  (define-Test fdd-VAR
    (titre "Elément - Variable")
    (local-var (Fdd-Constructed (newl Fdd-Constructed
                                      ['fdd-VAR (and (null (Is-Global-Context (cadr flow))) (car flow)) (cadr
flow)]))
          (fdd-var       (setq fdd-var (cadr flow)))
          (fdd-func      (progn (setq fdd-func (car flow))
                                (Set-Composition-Numeration fdd-func))))

    (test-eval (and (litatom car-flow) (litatom (cadr flow))
                    (or (null (car flow))
                        (Is-Global-Context (cadr flow))
                        (Var-In-Context (cadr flow) (car flow)))
                    (not (and Flow-Uniq (member ['fdd-VAR (and (null (Is-Global-Context (cadr flow))) (car flow))
(cadr flow)]
                                                Fdd-Constructed)))))

    (next-flow (cadr flow))
    (iilg 1) (nilg 0))

  (define-Test fdd-FUNC
    (titre "Elément - Fonction")

    (local-var (Fdd-Constructed (newl Fdd-Constructed
                                      ['fdd-FUNC (and (null (Is-Global-Context (cadr flow))) (car flow)) (cadr
flow)]))
          (fdd-var       (setq fdd-var (cadr flow)))
          (fdd-func      (progn (setq fdd-func (car flow))
                                (Set-Composition-Numeration fdd-func))))

    (test-eval (and (litatom car-flow) (litatom (cadr flow))
                    (or (null (car flow))
```

```
                    (Is-Global-Context (cadr flow))
                    (Var-In-Context (cadr flow) (car flow)))
                  (not (and Flow-Uniq (member ['fdd-FUNC (and (null (Is-Global-Context (cadr flow))) (car flow))
(cadr flow)]
                                              Fdd-Constructed)))))

        (next-flow (car flow))
        (iilg 1) (nilg 0))


                                              ; ************************************************** ; ;
                                              ; Troisième partie : le Point-de-vue de la composition ; ;
                                              ; ************************************************** ; ;

      (define-PDV composition
        (nom COMP)
        (titre "Composition des fonctions Lisp")
        (analyse (analyse-COMP . COMP))
        (Point-de-vue-type "Meta-Donnees/sources/composition.vlisp" gener)
        (global
                                              ; utilisées pour la numération sur les fonctions
         (com-local . 0)  (com-loop . 0)   (com-total . 0) (com-param . 0) (com-maxlg . 0)
         (com-lambda . 0) (com-global . 0) (com-modif . 0) (com-num . 0) (com-lst . 0)
         (com-str . 0) (com-plst . 0) (com-io . 0) (com-conds . 0) (com-test . 0) (com-x . 0) (com-gl . 0)
                                              ; pour l'étude sur la distance temporele d'exécution
         (com-eval-ray . 0) (p-com-eval-ray . 0) (ray-instr . nil) (has-ray . nil)
         (com-lg-ray . 0)
                                              ; instruction et fonction en cours
         (com-instr . nil) (com-func . nil)

         (Flow-NLG . PRC-Make-Flow-NLG))

        (compteurs com-ipr com-ilg com-iilg com-nilg com-nlg)
        (tests com-CDR
          com-CAR
          i-com f-com
          com-is-str com-is-num com-is-lst
          com-is-io com-is-loop com-is-cond
          com-is-test com-is-plist com-is-gl
          com-is-x
          com-ATOM com-NUMBP com-STRINGP com-NIL
          composition-fonction))

      (define-Test composition-fonction
        (titre "Lien fonction -> composition  ")
        (local-var (next-nilg (setq next-nilg (length flow)))
             (as-CAR (setq as-CAR nil)))
        (test-eval (and (litatom flow) (fval flow)))
        (next-flow car-flow)
        (ipr 1) (ilg 1) (iilg 1) (nilg next-nilg) (nlg (PRC-Get-NLG)))

      (define-Test com-CAR
          (titre "Composition CAR        ")
        (test-eval (listp car-flow))
        (next-flow (car flow))
        (local-var (Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow))
             (next-nilg (setq next-nilg (length car-flow)))
             (as-CAR (setq as-CAR t)))
        (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

      (define-Test com-CDR           (titre "Composition CDR        ")
        (local-var (as-CAR (setq as-CAR nil)))
        (test-eval (and (listp flow) (cdr flow)))
        (next-flow (cdr flow))
        (ilg (PRC-Next-ILG)) (iilg (1+ iilg)))

      (define-Test com-ATOM         (titre "Composition ATOM       ")
        (local-var (as-CAR (setq as-CAR t))
             (Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
        (next-flow car-flow)
        (test-eval (litatom car-flow)))

      (define-Test com-NUMBP       (titre "Composition NUMBERP    ")
        (local-var (Flow-Curent-Instruction (setq Flow-Curent-Instruction flow))
             (as-CAR (setq as-CAR t)))
        (next-flow car-flow)
```

```
  (test-eval (numbp car-flow)))

(define-Test com-STRINGP    (titre "Composition STRINGP     ")
  (local-var (Flow-Curent-Instruction (setq Flow-Curent-Instruction flow))
       (as-CAR (setq as-CAR t)))
  (next-flow car-flow)
  (test-eval (stringp car-flow)))

(define-Test com-NIL          (titre "Composition NIL        ")
  (test-eval (null car-flow))
  (next-flow car-flow))

(define-Test i-com
  (titre "Composition CAR + Instr.")
  (test-read "Entrez l'instruction")
  (local-var (Flow-Curent-Instruction (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow
flow
))
       (next-nilg             (setq next-nilg (length car-flow)))
       (as-CAR                 (setq as-CAR t)))
  (test-eval (and (null as-CAR) (listp car-flow) (eq (car car-flow) (GetTestData))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(define-Test f-com
  (titre "Composition CAR + ftype ")
  (test-read "Entrez le f-type")
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))
  (test-eval (and (null as-CAR) (listp car-flow) (litatom (car car-flow))
                  (= (ftyp (car car-flow)) (GetTestData))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Str   '(strcat strcar strcdr strcmp stringp strincp))

(define-Test com-is-str
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Str   ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Str))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Num   '(1+ 1- + - / * div mod sin cos tan rem))

(define-Test com-is-num
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Num   ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Num))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Lst   '(cons car caar caaar caadr cadr cadar caddr cdr cdar cddr cdaar cddar cdadr cdddr copy
nth
                        rplac rplaca rplacd rplacb last list listp mapc mapcar mapct memq mcons nconc
nmemq
                        nextl sublis quote))
(define-Test com-is-lst
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Lst   ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Lst))))
  (next-flow (car flow))
```

```
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Plist '(put get remprop))

(define-Test com-is-plist
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Plist ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Plist))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Atom  '(set setq))

(define-Test com-is-atom
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Atom ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Atom))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Lambda '(let letf lambda))

(define-Test com-is-lambda
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Lambda ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Lambda))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Loop  '(while mapc mapcar mapct until self repeat))

(define-Test com-is-loop
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Loop  ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Loop))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Conds '(if cond ifn when))

(define-Test com-is-cond
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Conds ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Conds))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-Test  '(stringp numbp atom listp null not and or eq neq ge le gt = > < numbp member))

(define-Test com-is-test
  (local-var (Flow-Curent-Instruction
        (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
       (next-nilg (setq next-nilg (length car-flow)))
       (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + Test  ")
```

```
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-Test))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-IO    '(read readlin readline readch princ print princh prin1 input output open close tyi tyo))

(define-Test com-is-io
  (titre "Composition CAR + IO    ")
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
      (next-nilg (setq next-nilg (length car-flow)))
      (as-CAR (setq as-CAR t)))
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-IO))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-GL    '(GLnumpush GLwinopen GLwinset GLcallback GLlmdef GLlmbind GLlmcolor GLswapbuf-
fer GLflush GLmakeob
   GLcallobj
                                GLfreeobj GLobjdel GLobjmodify GLobjtrans GLobjscale GLobjrotate GLtag
GLgentag GLinsert GLreplace
                                GLdelete GLcolor GLortho GLwindow GLperspective GLlookat GLpolarview
GLloadname GLpushname
                                GLpopname GLmmode GLpushmatrix GLpopmatrix GLloadidmatrix
GLscale GLtranslate GLrotate GLimtranslate
                                GLimrotate GLimscale GLclear GLbgn GLend GLvertex GLnormal GLset-
nurbs GLcurve GLsurface GLswaptmesh
                                GLsphmode GLsphere GLrect GLrectf GLpoly GLpolymode GLellipse
GLgetXYangulaire GLgetXYZ GLstring
                                GLfont GLfontpath GLchar GLstr GLtextsize GLboxtext GLboxfit GLtextangl
GLfixedwidth GLcentertext
                                GLrigthjustify GLleftjustify))

(define-Test com-is-gl
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
      (next-nilg (setq next-nilg (length car-flow)))
      (as-CAR (setq as-CAR t)))

  (titre "Composition CAR + GL    ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-GL))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(setq Instr-X     '(xscroll xEnable xDisable xRealize xUnrealize xManage xUnmanage xSetValues xGetVa-
lues
                        xGetResourceType xGetResourceList xGetNewResourceList xGetAllResource-
Lists xCircSubWidgets
                        xDrawLines xDrawRays xDrawPoints xDrawRectangles xFillRectangles xDra-
wArcs xFillArcs
                        xDrawString xGetStringDimension xClearArea xFillPolygon xSetForeground
xSetBackground
                        xDisplayBitmap xModifyGC xGetGCValue xGetCursorpos xchgsrc xwsave xCrea-
teResource
                        xsetRessource xCreateWidget xWidgetParent xAddCallback xAugment xOverride
xCheckEvent xIsRealized
                        xAddTreeLink xChangeWin xPosPointer xGetPosPointer iswidget xMap xUnmap
xRemoveWidget xwst xGrab
                        xPopup xPopdown xUngrab xflush sleep xwdim xBeep xwinp xwindowp isobjet
isressource home cleol
                        poscur cleos left shift_left right shift_right back up down cursorpos xMoveWidget
                        xResizeWidget xInitXbvlisp popupDisplay xCreateXbvlisp xGetSupWidget xGe-
tHierarchie xGetSousArbre))

(define-Test com-is-x
  (local-var (Flow-Curent-Instruction
       (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow flow)))
      (next-nilg (setq next-nilg (length car-flow)))
      (as-CAR (setq as-CAR t)))
  (titre "Composition CAR + X     ")
  (test-eval (and (null as-CAR) (listp car-flow) (member (car car-flow Instr-X))))
  (next-flow (car flow))
  (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

(define-Test com-no-CAR
  (local-var (Flow-Curent-Instruction  (setq Flow-Curent-Instruction (if (get-car-value car-flow 'sortie) car-flow
```

```
flo
        )))
              (next-nilg              (setq next-nilg (length car-flow)))
              (as-CAR                 (setq as-CAR t)))
          (titre "Composition Sans CAR")
          (test-eval (null as-CAR))
          (next-flow car-flow)
          (ipr (1+ ipr)) (ilg (PRC-Next-ILG)) (iilg 1) (nilg (length car-flow)) (nlg (PRC-Get-NLG)))

                                                  ; ********************************************** ; ;
                                                  ; Point-de-vue supplémentaires :                ; ;
                                                  ; - FUNCS : liste des fonctions du programme    ; ;
                                                  ; - TYPES : liste des fonctions classée par types ; ;
                                                  ; ********************************************** ; ;

        ; ************************************************** ; ;
        ; Point-de-vue de la liste des fonctions du programme ; ;
        ; ************************************************** ; ;

        (define-PDV Fonctions-du-programme
          (nom FUNCS) (titre "Fonctions du programme       ")
          (analyse (analyse-FUNCS . liste-func))
          (Point-de-vue-type "Meta-Donnees/sources/funcs.vlisp" traverse-funcs)
          (global (funcs-nelem . length))
          (compteurs funcs-ipr funcs-ilg funcs-iilg funcs-nilg)
          (tests funcs-CDR funcs-ATOM))

        (define-Test funcs-CDR
          (titre "Fonctions - largeur")
          (test-eval (cdr flow))
          (next-flow (cdr flow))
          (ilg (1+ ilg)) (iilg (1+ iilg)))

        (define-Test funcs-ATOM
          (titre "Fonctions - élément")
          (local-var (Dummy-Var (Set-Composition-Numeration car-flow)))
          (test-eval (and car-flow (litatom car-flow)))
          (next-flow car-flow))

        ; ********************************************************************************************** ; ;
        ; Point-de-vue de la liste des fonctions du programme triée par types d'instruction utilisées ; ;
        ; ********************************************************************************************** ; ;
        (define-PDV Fonctions-du-programme-typees
          (nom TYPES) (titre "Fonctions par types          ")
          (analyse (analyse-TYPES . liste-type))
          (Point-de-vue-type "Meta-Donnees/sources/types.vlisp" traverse-types)
          (global (types-nelem . length))
          (compteurs nil types-ilg types-iilg types-nilg)
          (tests types-CDR types-ATOM))

        (define-Test types-CDR
          (titre "Types - largeur")
          (test-eval (cdr flow))
          (next-flow (cdr flow))
          (ilg (1+ ilg)) (iilg (1+ iilg)))

        (define-Test types-ATOM
          (titre "Types - élément")
          (local-var (Dummy-Var (Set-Composition-Numeration car-flow)))
          (test-eval (and car-flow (litatom car-flow)))
          (next-flow car-flow))
```

Annexe I.6 Définition des analyses liées aux points de vues : Zprc-analyses.vlisp

```
; Définition des analyses liées aux parcours ; ;

(de GEN-analyse (elem)                                                    GEN-analyse
  (PutPrcData 'generiques elem 'generiques elem))

; analyse liée à l'exécution des programmes : placement des activités sur le fonctions et variables
```

```
(de analyse-Dynamique (func vars)                                    analyse-Dynamique
  (unless (GetFuncData func 'vars)
    (unless (GetPrcData 'FDC func 'liste-func) (analyse-FDC func))
    (unless (GetPrcData 'COMP func 'COMPOSITION) (analyse-COMP func))
                                          ; construction de la liste de toutes les couples
                                          ; [fonction variables] presents dans le programme...
    (MPV-Message (strcat "Liens automatiques avec " func))
    (mapc (GetPrcData 'FDC func 'liste-func)
     (lambda (x)
       (mapc '(i-in i-out o-in o-out)
             (lambda (prop)
               (if (and (null (GetFuncData x prop))
                        (setq aux (GetGlobalObjects prop)))
                   (PutFuncData x prop aux))))
       (mapc `(sortie ,@(getival x 0) ,@(getival x 1))
             (lambda (y)
               (when y
                 (mapc (GetGlobalObjects 'v-obj)
                       (lambda (z)
                         (cond
                          ((cadr z) (PutVarData y 'v-obj (cons [(car z) x] (GetVarData y 'v-obj))))
                          ((not (member z (GetVarData y 'v-obj)))
                                (PutVarData y 'v-obj (cons z (GetVarData y 'v-obj)))))))
                 (ifn (member [x y] vars) (setq vars (cons [x y] vars))))))))
    (PutFuncData func 'vars vars)
    (MPV-Reset-Label)))

; construction du flot de contrôle d'un programme ; ;

; construit les différents flots de contrôle : sous forme de graphe
; et sous la forme d'une liste contenant les fonctions utilisées
(de analyse-FDC (func)                                               analyse-FDC
  (when (and (litatom func) (fval func) (null (GetPrcData 'FDC func 'FDC)))
    (MPV-Message  (strcat "Analyse FDC de " func))
    (PutPrcData 'FDC func 'FDC (Make-Control-Tree func))
    (PutPrcData 'FDC func 'liste-func (reverse (cdr (GetPrcData 'FDC func 'FDC))))
    (setq fdc-nelem (length (GetPrcData 'FDC func 'liste-func)))
    (mapc (GetPrcData 'FDC func 'liste-func)
     (lambda (aFunc)
       (ifn (get aFunc 'entree) (Install-ival aFunc))))
    (PutPrcData 'FDC func 'FDC (car (GetPrcData 'FDC func 'FDC)))
    (MPV-Reset-Label)))

; récupére la liste des fonctions utilisées
(de analyse-FUNCS (func)                                             analyse-FUNCS
  (unless (GetPrcData 'FUNCS func 'liste-func)
    (ifn (GetPrcData 'FDC func 'liste-func) (analyse-FDC func))
    (PutPrcData 'FUNCS func 'liste-func (GetPrcData 'FDC func 'liste-func))
    (setq funcs-nelem (length (GetPrcData 'FDC func 'liste-func)))))

; tris la liste des fonctions utilisées suivant l'ordre suivant :
; 1 - les fonctions contenant des boubles
; 2 -                    lambdas
; 3 -                    tests
; 4 -                    entrées/sorties
; 5 -                    opérations sur les numériques
; 6 -                    opérations sur les chaines
; 7 -                    opérations sur les listes
; 8 - les autres
(de analyse-TYPES (func)                                             analyse-TYPES
  (unless (GetPrcData 'TYPES func 'liste-type)
    (MPV-Message  (strcat "Analyse TYPES de " func))
    (ifn (GetPrcData 'FDC func 'liste-func) (analyse-FDC func))
    (ifn (GetPrcData 'COMP func 'COMPOSITION) (analyse-COMP func))
    (let ((fcs (GetPrcData 'FDC func 'liste-func))
     (bcl) (lmb) (tst) (io) (num) (str) (lst) (autres))
      (ifn fcs (PutPrcData 'TYPES func 'liste-type (Zappendn bcl lmb tst io num str lst autres))
      (Set-Composition-Numeration (car fcs))
      (cond
       (com-loop              (self (cdr fcs) (cons (car fcs) bcl) lmb tst io num str lst autres))
       (com-lambda              (self (cdr fcs) bcl (cons (car fcs) lmb) tst io num str lst autres))
       ((or com-conds com-test)  (self (cdr fcs) bcl lmb (cons (car fcs) tst) io num str lst autres))
       ((or com-io com-x com-gl) (self (cdr fcs) bcl lmb tst (cons (car fcs) io) num str lst autres))
       (com-num                 (self (cdr fcs) bcl lmb tst io (cons (car fcs) num) str lst autres))
       (com-str               (self (cdr fcs) bcl lmb tst io num (cons (car fcs) str) lst autres))
       (com-lst               (self (cdr fcs) bcl lmb tst io num str (cons (car fcs) lst) autres))
```

```
      (t                    (self (cdr fcs) bcl lmb tst io num str lst (cons (car fcs) autres))))))
   (MPV-Reset-Label)))

(de Zappendn (l . lsts)                                                    Zappendn
   (cond
    ((null lsts) l)
    ((null l)    (apply 'Zappendn lsts))
    (t (append l (apply 'Zappendn lsts)))))

(de analyse-FDD (func)                                                     analyse-FDD
  (unless (GetPrcData 'FDD func 'FDD)
    (MPV-Message  (strcat "Analyse FDD de " func))
    (PutPrcData 'FDD func 'FDD (Make-D-Tree func))
    (MPV-Reset-Label)))

; utilisées pour l'initialisation des variables globales ; ;
(de Zself (x) x)                                                           Zself
(de Znil () nil)                                                           Znil
(de Zzero () 0)                                                            Zzero

(de analyse-COMP (func)                                                    analyse-COMP
  (unless (GetPrcData 'COMP func 'COMPOSITION)
    (ifn (GetPrcData 'FDC func 'liste-func) (analyse-FDC func))
    (mapc (GetPrcData 'FDC func 'liste-func)
     (lambda (x)
      (MPV-Message  (strcat "Analyse COMP de " x))
      (ifn (GetPrcData 'COMP x 'COMPOSITION) (Make-Composition-Numeration x))
      (MPV-Message  (strcat "Analyse chrono de " x))
      (ifn (GetPrcData 'COMP x 'e-time) (Make-Func-Chrono x))
      (MPV-Message  (strcat "Analyse chaines de " x))
;     (ifn (GetPrcData 'COMP x 'chaines)
;               (PutPrcData 'COMP x 'chaines (Make-Func-String x)))
      (ifn (GetPrcData 'COMP x 'COMP) (PutPrcData 'COMP x 'COMP (cdr (fval x))))))
   (MPV-Reset-Label)))

(de Set-Composition-Numeration (func)                                      Set-Composition-Numeration
  (if (setq aux (GetPrcData 'COMP (or func com-instr) 'COMPOSITION))
     (setq com-local (get aux 'local)
      com-total (get aux 'total)
      com-param (get aux 'param)
      com-global (get aux 'global)
      com-modif (get aux 'modif)
      com-loop (get aux 'loop)
      com-lambda (get aux 'lambda)
      com-num   (get aux 'num)
      com-lst   (get aux 'lst)
      com-str   (get aux 'str)
      com-plst  (get aux 'plst)
      com-conds (get aux 'conds)
      com-test  (get aux 'test)
      com-io    (get aux 'io)
      com-x     (get aux 'x)
      com-gl    (get aux 'gl))))
```

Annexe I.7 Définitions des analyses liées aux points de vues (2) : Zanalyses.vlisp

```
                                        ; ******************************** ; ;
                                        ; Construction du flux de controle ; ;
                                        ; ******************************** ; ;
;
; construction des appels avec les arguments a des fonctions utilisateur
; presents dans les sources d'une fonction
;
(de get-calls (f)                                                          get-calls
   (let (ret)
     (cond
      ((null (%find (ival f) 4)) nil)
      ((setq ret (%find (ival f) '**calls**)) (car ret))
      (t
  (addival f `(**calls** ,(setq ret (get-function-calls (cdr (fval f)) (%find (ival f) 4)))))
  ret))))
```

```
(de get-function-calls (body calls all)                                    get-function-calls
   (cond
    ((not (listp body)) all)
    ((not (listp (car body))) (get-function-calls (cdr body) calls all))
    ((and  (atom (caar body))
      (member (caar body) calls)
      (not (member (car body) all)))
     (get-function-calls (cdr body) calls (cons (car body) (if (not (listp all)) [all] all))))
    (t
     (let ((c1 (get-function-calls (car body) calls nil))
      (c2 (get-function-calls (cdr body) calls all)))
  (append c1 (if (not (or (listp c1) (listp c2))) [c2])))))))

(de Make-Control-Tree (root-func)                                          Make-Control-Tree
   (setq flis ())
   (cons (cons root-func [(Control-Tree-call-tree root-func (Control-Tree-get-calls root-func))])
    flis))

(defmacro is-good-func (f) `(or (= (ftyp ,f) 7) (= (ftyp ,f) 8)))          is-good-func

(defmacro cadddr (x) `(caddr (cdr ,x)))                                    cadddr

;
; recuperation des appels dans un defmacro (ftyp = 9)
;
(de Identity (x) x)                                                        Identity

(de Control-Tree-get-calls (func aux)                                      Control-Tree-get-calls
  (cond
   ((or (null func) (= (ftyp func) 0)) ())
   ((setq aux (get func 'assert))
    (let ((ret nil) (phrase nil))
      (while aux
   (setq phrase (cdar aux))
   (while phrase
     (cond
      ((member (caar phrase) ret) nil)
      ((get (caar phrase) 'assert) (setq ret (cons (caar phrase) ret)))
      (t
       (let (phrase-body (cdar phrase))
         (cond
          ((listp phrase-body) (self (car phrase-body)) (self (cdr phrase-body)))
          ((and (not (member phrase-body ret)) ;(member phrase-body all_functions); ;
                    (listp (fval phrase-body))) (setq ret (cons phrase-body ret)))))))
    (nextl phrase))
   (nextl aux))
     ret))
   ((is-good-func func)  ; de ou df ; ;
    (let ((fcs (%find (ival func) 4))
     (ret nil))
      (while fcs
   (if (is-good-func (car fcs)) (newl ret (car fcs)))
   (nextl fcs))
     ret))
   ((= (ftyp func) 9)                          ; defmacro ; ;
    (let (body (cdr (cadr (caddr (cadddr (cadr (fval func)))))))
      (if (listp (car body)) (setq body (car body))) ;reajustement du quote ; ;
      (setq calls ())
      (let (recurse (if (listp (car body)) (car body) body))
   (cond
    ((atom recurse) calls)
    ((listp (car recurse))
     (self (car recurse)) (self (cdr recurse)))
    ((and (member (car recurse) all_functions)
          (not (member (car recurse) calls))
          (> (ftyp (car recurse)) 0))
     (setq calls (cons (car recurse) calls))
     (self (cdr recurse)))                     ; old was recursing on cadr too... ; ;
    (t
     (self (cdr recurse)))))))))
;
; construction effective
;
(de Control-Tree-call-tree (x %exp %arb)                                   Control-Tree-call-tree
  (unless (memq x flis)
```

```
   (setq flis (cons x flis))
   (PutFuncData x 'fonction-racine root-func))
  (cond
   ((null %exp) %arb)
   ((memq (car %exp) flis)
    (Control-Tree-call-tree x (cdr %exp)
                            (if %arb (append %arb (car %exp)) [(car %exp)])))
   ((setq aux (Control-Tree-call-tree (car %exp) (Control-Tree-get-calls (car %exp))))
    (Control-Tree-call-tree x (cdr %exp) (append %arb [(car %exp) aux])))
   (t (Control-Tree-call-tree x (cdr %exp)
                              (if %arb (append %arb (car %exp)) [(car %exp)])))))


                                                ; ************************************** ; ;
                                                ; Analyse de la constitution d'une fonction ; ;
                                                ; ************************************** ; ;

; mise en place d'un appel a f-entree en entree de la fonction et f-sortie en sortie

(de Install-ival (func)                                                    Install-ival
  (remprop func 'entree)
  (entree func `(f-entree ',func 'func))
  (entree func `(setq func-is-first-entree (chemin-entree ,func)))
  (remprop func 'sortie)
  (sortie func `(f-sortie ',func 'func))
  (sortie func `(setq func-is-last-sortie (chemin-sortie ,func)))
  (mapc (append (%find (ival func) 0) (%find (ival func) 1))
   (lambda (var)
     (replaceival var `(entree (f-entree ',var 'var))))))

;
; le resultat est place dans la pliste de l'atome correspondant a la fonction
;
(de Make-Composition-Numeration (f)                          Make-Composition-Numeration
  (if (and f (listp (fval f)))
      (let (var (or (GetPrcData 'COMP f 'COMPOSITION) (gensym)))
    (newl Numeration-Results var)
    (PutPrcData 'COMP f 'COMPOSITION var)
    (rplacd var nil)
    (rplacd var (Calcul-expr-numeration (cdr (fval f))))
    (put var 'func f))))


(de Calcul-expr-numeration (expr ret)                                Calcul-expr-numeration
  (let ((n-local-call 0)  (n-plst 0)          (n-conds 0) (n-test 0)
    (n-str 0)             (n-loop 0)          (n-num 0)
    (n-io 0)              (n-lst 0)           (n-calls 0)
    (n-x  0)              (n-gl 0) (ft) (ret))
    (let (ex-l expr)
      (when (listp ex-l)
                                                ; (rplacd (getcar ex-l) nil)
                                                ;           (print ex-l (car ex-l))
                                                ;           (break a)
    (when (and (listp (car ex-l)) (not (listp (caar ex-l))))
                                                ;(or (listp (caar ex-l))
                                                ;(getival ex-l 'entree)
                                                ;(getival ex-l 'sortie))
      (replace-car ex-l 'entree `((f-entree ',ex-l 'instr ',f)))
      (replace-car ex-l 'sortie `((f-sortie ',ex-l 'instr ',f))))
    (cond
     ((listp   (car ex-l))
      (self (car ex-l)) (self (cdr ex-l)))
     ((numbp   (car ex-l)) (incr n-num) (self (cdr ex-l)))
     ((stringp (car ex-l)) (incr n-str) (self (cdr ex-l)))
     ((null    (car ex-l)) (incr n-lst) (self (cdr ex-l)))
     ((= (setq ft (ftyp (car ex-l))) 0) (self (cdr ex-l)))
     ((> ft 6)
      (if (member (car ex-l) (%find (ival f) 4)) (incr n-local-call))
      (put var ft (+ 1 (get var ft)))
      (incr n-calls)
      (self (cdr ex-l)))
     (t
      (cond
       ((member (car ex-l) Instr-Plist)(incr n-plst))
       ((member (car ex-l) Instr-Conds)(incr n-conds))
       ((member (car ex-l) Instr-Test) (incr n-test))
```

```
            ((member (car ex-l) Instr-Lst)  (incr n-lst))
            ((member (car ex-l) Instr-Str)  (incr n-str))
            ((member (car ex-l) Instr-Loop) (incr n-loop))
            ((member (car ex-l) Instr-Num)  (incr n-num))
            ((member (car ex-l) Instr-IO)   (incr n-io))
            ((member (car ex-l) Instr-X)    (incr n-x))
            ((member (car ex-l) Instr-GL)   (incr n-gl)))
           (put var ft (+ 1 (get var ft)))
           (incr n-calls)
           (self (cdr ex-l))))))
                                                ; local et careful ont un sens en cas de lecture sans careful et
execution
                                                ; avec...
          (setq ret nil)
          (if (> n-num 0)  (setq ret `(num ,n-num)))          (if (> n-lst 0)  (setq ret `(lst ,n-lst ,@ret)))
          (if (> n-str 0)  (setq ret `(str ,n-str ,@ret)))    (if (> n-plst 0)  (setq ret `(plst ,n-plst ,@ret)))
          (if (> n-io 0)  (setq ret `(io ,n-io ,@ret)))       (if (> n-conds 0)  (setq ret `(conds ,n-conds ,@ret)))
          (if (> n-test 0)  (setq ret `(test ,n-test ,@ret)))  (if (> n-local-call 0)  (setq ret `(local ,n-local-call ,@ret
        ))
          (if (> n-loop 0)  (setq ret `(loop ,n-loop ,@ret)))  (if (> n-x 0)  (setq ret `(x ,n-x ,@ret)))
          (if (> n-gl 0)  (setq ret `(gl ,n-gl ,@ret)))       (if (> n-calls 0)  (setq ret `(total ,n-calls ,@ret)))
          (if (> (length (car (fval f))) 0)  (setq ret `(param ,(length (car (fval f))) ,@ret)))
          (if (> (- (length (%find (ival f) 0)) (length (car (fval f)))) 0)
         (setq ret `(lambda ,(- (length (%find (ival f) 0)) (length (car (fval f)))) ,@ret)))
          (if (> (length (%find (ival f) 1)) 0)  (setq ret `(global ,(length (%find (ival f) 1)) ,@ret)))
          (if (> (length (%find (ival f) 2)) 0)  (setq ret `(modif ,(length (%find (ival f) 2)) ,@ret)))
          (setq ret `(,@ret ,@(cdr var)))
          ret))

        ; construction de la représentation du code source des fonctions :
        ; placement dans FuncData de 'l_chaines représentant les différentes lignes du code source
        ; Ce programme est le fruit du travail de :
        ; Harald Wertz : pretty printer original
        ; Benjamin Drieu : première version
        ; Damien Ploix : version finale

        ; On utilise les packages pour eviter les "clashes"
        (package make_func_string)

        ; Cette fonction "marque" une fonction passee en argument.  C'est a
        ; dire que le programme la connait desormais, et que cette fonction
        ; va recevoir des activites supplementaires (entre autres)
        (de |Make-Func-String (f l_chaines lmargin)                                    |Make-Func-String
                                                ; Test de la validite de la fonction passee en argument (il y a
                                                ; peut-etre plus beau, mais ca marche bien)
          (unless (or (= 0 (fval f))
                (|GetPrcData '|COMP f '|chaines))
                                                ; On remet a jour differentes variables
            (setq lmargin 0
             prev_ligne nil                ; ligne de code correspondant à la ligne construite
             l_chaines nil                 ; la liste des lignes imprimées
             chaine ""                     ; la ligne en cours
             NB 1)                         ; le numéro de ligne en cours

                                           ; On execute un clone de pretty-print pour avoir une
                                           ; belle mise en page de la fonction dans la widget
            (let (osp (status print))
              (status print 3)             ; bit 0 : impression des guillemets, bit 1 : pas d'espace avant une
impression
              (setq lmargin 0)
              (setq x (cdr (assoc (ftyp f) '((7 . de) (8 . df) (9 . dm)))))
              (and x (my-print [x f . (fval f)]))
                                           ; Ceci pour avoir la derniere chaine (je ne peut pas
                                           ; savoir quand on sort pour la derniere fois de my-print,
                                           ; ou alors ca sera tres laid)
              (SaveLast)
              (eval `(status print ,osp)))
            l_chaines)))

        ; construit une chaine de n espaces (marge de gauche...)
        (de nespace (n)                                                                nespace
          (if (= n 1) " " (strcat " " (nespace (1- n)))))

        (de SaveLast (l) (NewLine))                                                    SaveLast

        (de NewLine ()                                                                 NewLine
```

```
                                        ; on va changer de chaine
  (if l_chaines ; l_chaines va contenir les chaines et un ptr vers la ligne de code ...
      (rplacd (last l_chaines) (list (cons chaine (cons prev_ligne nil))))
    (setq l_chaines (list (cons chaine (cons prev_ligne nil)))))

  (if prev_ligne (|replace-car prev_ligne '|l_nb NB))

  (incr NB)
  (setq prev_ligne (and (not (eq l prev_ligne)) l)
   chaine (or (and (> (outpos) 0) (nespace (outpos))) "")))

; Cette fonction ajoute un élément Lisp dans la chaine en respectant le status de print :
; bit 0 à 1 : impression d'une chaine entourée de guillemets,
; bit 1 à 1 : impression d'un espace avant la chaine
(de ajoutChaine (elem force_no_string)                                  ajoutChaine
  (cond
   (force_no_string (setq chaine (strcat chaine elem)))
   (t
    (if (= (logand (status print 2) 0)) (setq chaine (strcat chaine " ")))
    (setq chaine
     (strcat chaine
                (if (and (stringp elem) (= (logand (status print) 1) 1))
                   (strcat "\"" elem "\"")
                 elem))))))

; Cette fonction tres utile renvoie la ligne de code située à la
; position nb
(de |get_nth_line (nb)                                                  |get_nth_line
  (cdar (nth nb l_chaines)))

; Cette fonction est ma propre version de (pretty|p-p).  J'ai
; legerement modifie le code, c'est-a-dire principalement remplace
; les (print) par des (ajoutChaine et NewLine).
(de my-print  (l x)                                                     my-print
  (cond
   ((null l) (ajoutChaine "()" t))
   ((atom l) (ajoutChaine l))
   ((and (eq (car l) quote) (null (cddr l)))
    (ajoutChaine "'" t) (my-print  (cadr l)))
   (t
    (ajoutChaine "(" t)
    (selectq (and (litatom (setq x (print-car))) (get x 'ptyp))
        (1 (p-progn))
        (2 (p-p1) (p-progn t))
        (3 (p-p1) (p-p1) (p-progn t))
        (4 (p-cond))
        (5 (p-p1) (p-cond))
        (6 (p-setq))
        (7 (p-log))
        (t
         (outpos (t+ 3))
         (while (listp l) (p-p1))
         (outpos (t- 3))))
    (and l (ajoutChaine " . " t) (ajoutChaine l))
    (ajoutChaine ")" t))))

; impression du car de l avec vérification des commentaires
(de print-car ()                                                        print-car
  (if (not (or (member '|PCOM (getcar l))
          (member '|COM (getcar l))))
      (my-print (car l))
    (p-comm t '|PCOM)
    (my-print (car l))
    (p-comm t '|COM))
  (nextl l))

; traitement des commentaires
(de p-comm (-x- COMMENT -y- -Z-)                                        p-comm
  (setq -y- t) ; point virgule sans espace
  (p-comm1 -x-)
  (when (and -Z- (eq COMMENT '|PCOM)) (ajoutChaine " " t)))

(de p-comm1 (-x-)                                                       p-comm1
  (let ((z (reverse (cdr (getcar l)))) (lmargin (outpos)) (ipos -1))
    (status print 2) ; on ne montre plus les "
    (while z                                    ; dans le bon sens
```

```
      (cond
       ((equal (cadr z) COMMENT)
    (if (= ipos -1) (setq ipos (strlen chaine))
     (outpos ipos)
     (NewLine))          ; plusieurs lignes de commentaires pour
                                          ; une seule ligne de code...
    (ajoutChaine (if (and -y- (eq COMMENT '|PCOM))
                     (progn (setq -y- ()) ";") " ; ;") t)
    (let ((INDICATEUR (status print)))
     (status print 2)
     (my-print (car z))
     (eval `(status print ,INDICATEUR)))
    (setq -Z- t)
    (setq z (cddr z))
    nil)
       (t
    (setq z (cddr z)))))
     (outpos lmargin)
     (status print 3)))

(|defmacro t+ (n) `(incr lmargin ,n)) ; décalage
(|defmacro t- (n) `(decr lmargin ,n)) ; replacement

(de p-p1 ()  (ajoutChaine " " t)  (print-car))                                    p-p1

(de p-cond () ; formatage de l'impression pour les formes de type "cond"          p-cond
  (outpos (t+ 3))                     ; décalage vers la droite
  (while (listp l)                    ; tant qu'il y a des clauses à imprimer
    (NewLine)                                 ; nouvelle ligne...
    (ajoutChaine "(" t)               ; ouverture de la clause
    (let (l (nextl l)) ; traitement de l'élément du cond
      (when l
    (print-car) ; le test
    (outpos (t+ 1))
    (while l (NewLine) (print-car)) ; le reste
    (outpos (t- 1))))
     (ajoutChaine ")" t)) ; rien dedans ???
  (outpos (t- 3)))                    ; on se repositionne

; formatage des formes progn
(de p-progn (+++)                                                                 ; +++ indic
  (if (not (or (cdr l) +++)) (p-p1) ; un seul argument
    (outpos (t+ 3))                   ; plusieurs
    (while (listp l)
      (NewLine)
      (print-car))
    (outpos (t- 3))))

; formatage des formes setq multiples (un couple donne - variable par ligne)
(de p-setq ()                                                                     p-setq
  (outpos (t+ 5))
  (p-p1) (p-p1)
  (while l (NewLine) (p-p1) (p-p1))
  (outpos (t- 5)))

; formatage des formes opération logique and ou or
(de p-log ()                                                                      p-log
  (let ((oldpos (outpos))
    (npos (setq lmargin (strlen chaine))))
    (outpos npos)
    (p-p1)
    (while l (NewLine) (p-p1))
    (outpos (setq lmargin oldpos))))

(|mapc '( progn prog1 exit) (lambda (x) (put x 'ptyp 1)))
(|mapc '(lambda letf when escape if ifn let |mapc |mapcar while until) (lambda (x) (put x 'ptyp 2)))
(|mapc '(de df dm dmc) (lambda (x) (put x 'ptyp 3)))
(|mapc '( cond) (lambda (x) (put x 'ptyp 4)))
(|mapc '(selectq) (lambda (x) (put x 'ptyp 5)))
(|mapc '(setq) (lambda (x) (put x 'ptyp 6)))
(|mapc '(and or) (lambda (x) (put x 'ptyp 7)))

; Oh, no!  Everything looks SO boring, now...
(package)

                                          ; ********************************** ; ;
```

```
                                        ; Construction des arbres d'influence ; ;
                                        ; ********************************** ; ;

(de Make-D-Tree (func vars aux)                                                    Make-D-Tree
  (let (deps (mapcar (if vars vars (GetFuncData func 'vars))
                     (lambda (x)
                       (setq influence-done nil)
                       (MPV-Message (strcat "Influence de " (car x) " / " (cadr x)))
                       `(,@x
                             ,(if (and (atom (car (setq aux (set-var-influence (cdr (fval (car x))) [(cadr x)] (car x)))))
                                       aux)
                                  [aux] aux)))))
    (MPV-Message "Vérification des influences")
    (Check-D-Tree deps deps)))

(de Check-D-Tree (d-chk d-tree)                                                    Check-D-Tree
  (ifn d-chk d-tree
       (Check-D-Tree (cdr d-chk)
                     (Remove-if-multiple (car d-chk) d-tree))))


(de Remove-if-multiple (elem lst)                                                  Remove-if-multiple
  (if (let (l lst) (cond
                     ((null l) t)
                     ((equal elem (car l)) (self (cdr l)))
                     ((is-included elem (car l)) nil)
                     (t (self (cdr l)))))
      lst
    (if (equal elem (car lst)) (cdr lst)
      (let (l lst)
    (cond
     ((null l) nil)
     ((equal elem (cadr l)) (rplacd l (cddr l)))
     (t (self (cdr l)))))
      lst)))

(de is-included (a as)                                                             is-included
  (or (equal a as)
      (and (listp as)
       (or (is-included a (car as))
           (is-included a (cdr as))))))



; fonctions annexes...

; recherche d'au moins un member d'une liste atome dans un arbre
(de present-in-tree (lat tree)                                                     present-in-tree
  (cond
   ((null tree) nil)
   ((atom tree) (member tree lat))
   (t (or (present-in-tree lat (car tree))
      (present-in-tree lat (cdr tree))))))

; recherche du placement d'une (ou plusieurs) variable(s) dans un appel
(de present-in-call (var call)                                                     present-in-call
  (let ((p-call call) (indice 1) (ret nil))
    (ifn p-call ret
    (self (cdr p-call)
          (1+ indice)
          (if (and (or (not (listp (car p-call)))
                       (neq (caar p-call) 'quote))
                   (present-in-tree var (car p-call)))
              (cons indice ret)
            ret)))))

(defmacro c-newl (v val) `(if ,v (if (atom (car ,v)) (setq ,v (append [,v] [,val]))   c-newl
                                     (setq ,v (append ,v [,val])))
                                 (setq ,v ,val)))

(defmacro Make-Lambda (indices vars corps)                                         Make-Lambda
  `(progn
     (GA-Push v)                        ; push des anciennes variables
     (mapc ,indices                     ; ajout des variables de la lambda
      (lambda (v-indice)
```

```
            (ifn (member (v-indice ,vars) v) (newl v (v-indice ,vars)))))
          (setq aux (set-var-influence ,corps v)) ; parse du corps de la lambda
          (if aux (c-newl pars aux))
          (setq v (GA-Pop))))

(defmacro Make-Func-Call (func n-var)                                          Make-Func-Call
  `(unless (member [,func (,n-var (car (fval ,func)))] influence-done)
     (c-newl pars [,func
                   (,n-var (car (fval ,func)))
                        (set-var-influence (cdr (fval ,func))
                                           [(,n-var (car (fval ,func)))]
                                           ,func)
                   ])))

;
; set-var-influence : point d'entree de la recherche de l'influence d'une variable dans un programme
; les arguments sont :  f = le corps de la fonction
;                       v = le ou les variables recherchees,
;                       touched = les fonctions influencees
;                  is-args = parsing d'arguments (t) ou d'expression (nil)
;

(de set-var-influence (body v func aux tmp) ; determination des liens d'influence dans les appels ; ;set-var-influence
  (if func (newl influence-done [func (car v)]))
  (let ((expr body) (vars v) (pars))            ; parcour du code de la fonction
    (ifn expr pars
    (cond
     ((atom (car expr)) nil)
     ((eq (caar expr) 'let)            ; cas du let : exploration des liens des nouvelles variables
      (GA-Push (setq tmp v))
      (mapc (if (listp (car (cadar expr))) (cadar expr) [(cadar expr)]) ; present dans l'edition de liens
                (lambda (x)
                  (when (or (present-in-call v (cdr x))
                            (and (atom (cadr x))
                                 (member (cadr x) v)))
                        (if (setq aux (set-var-influence (cdr x) v)) (c-newl pars aux))
                        (if (not (member (car x) v)) (newl v (car x))))))
      (if (setq aux (self (cddr (car expr)) v)) (c-newl pars aux))
      (setq v (GA-Pop)))
     ((eq (caar expr) 'lambda)      ; cas du lambda
      (setq tmp (present-in-call v (cdddr (car expr)))) ; verification de la presence dans les valeurs des va-
riables
      (Make-Lambda tmp
                   (cadar expr)
                   (cdr (cdar expr)))
      (if (setq aux (self (cdddr (car expr)) v nil)) (c-newl pars aux)))
     ((member (caar expr) '(mapc mapcar)) ; cas des map???
      (cond
       ((null (present-in-call v [(cadar expr)])) nil)
       ((eq (car (cadr (cdar expr))) 'quote) ; cas d'un appel quote
        (if (> (ftyp (setq tmp (cadr (cadr (cdar expr))))) 6)
               (Make-Func-Call tmp 1)))
       ((eq (car (cadr (cdar expr))) 'lambda) ; cas d'un appel par une lambda expression
        (Make-Lambda '(1)
                     (cadr (cadr (cdar expr)))
                     (cddr (cadr (cdar expr)))))))
     (t                                        ; autres cas
      (if (and (listp (caar expr))
               (setq aux (self (car expr) vars nil))) (c-newl pars aux))
      (setq tmp (present-in-call v (cdar expr)))
      (cond
       ((null tmp) t)                   ; pas de reference a une variable dans les donnees de l'appel...
                                        ; des references...
       ((member (caar expr) '(incr decr setq rplac rplacd rplaca set newl)) ; instruction modificatrice d'une va-
riable..


        (if (and (litatom (cadar expr))
                 (not (equal tmp '(1)))
                 (not (member (cadar expr) v)))  (newl v (cadar expr))))
       ((and (litatom (caar expr))   ; appel a une fonction utilisateur
             (> (ftyp (caar expr)) 6))
        (mapc tmp
              (lambda (v-indice)
                (Make-Func-Call (caar expr) v-indice)))))
      (if (and tmp (setq aux (set-var-influence (cdar expr) v))) ; parse des arguments de l'appel
```

```
      (c-newl pars aux))))
   (self (cdr expr) vars pars))))

                                        ; ********************************************* ; ;
                                        ; Construction des positionnement chronologiques ; ;
                                        ; ********************************************* ; ;


; types de fonctions Lisp
;
(setq SUBR0 1) (setq SUBR1 2) (setq SUBR2 3) (setq SUBR3 4)
(setq NSUBR 5) (setq FSUBR 6)
(setq EXPR  7) (setq FEXPR 8)
(setq MACRO 9) (setq ESCAPE 10) (setq MACOUT 11)

; traitement du corps de la fonction pour le calcul de la "distance"
; d'evaluation par rapport a l'entree dans la fonction
;
; Temps de l'evaluateur
; t0 = temps initial
; t(x) = temps initial du calcul du temps des composants de x
; T(x) = temps de x
; Cas :
;  if test if-true if-false   -> t(test) = t0+1, t(if-true) = t0+T(test), t(if-false) = t0+T(test)
;                         -> T(if) = T(test) + max(T(if-true), T(if-false))
;  cond expr1 expr2          -> t(expr1) = t0+1, t(expr2) = t0+T(test-expr2),
;                         -> T(cond) = somme(T(test-expri)) + max(T(expri))
;  lambda expr data          -> t(expr) = t0+t(data), t(data) = t0+1
;  let data expr             -> t(data) = t0+1, t(expr) = t0+T(data)
;  ((expr1) expr2     )      -> t(expr1) = t0+1, t(expr2) = t0+T(expr1),
;
;  FSUBR, FEXPR              -> T(expr) = 1
;  atom                   -> T(atom) = 1

(defmacro Set-Com-Ray ()                                                    Set-Com-Ray
  `(ifn (eq ray-instr Flow-Curent-Instruction)
   (setq p-com-eval-ray com-eval-ray ray-instr Flow-Curent-Instruction
        com-eval-ray (get-car-value Flow-Curent-Instruction 'chrono))))

(de Make-Func-Chrono (func)                                                 Make-Func-Chrono
  (when (listp (fval func))
    (PutPrcData 'COMP func 'chrono (NExpr-Chrono (cdr (fval func)) 0))))

; parcour du corps d'une fonction ou d'un progn
(de NExpr-Chrono (body time)                                                NExpr-Chrono
  (ifn body time
      (NExpr-Chrono (cdr body) (PutPrcData 'COMP body 'chrono (Expr-Chrono (car body) time)))))

(de OneExpr-Chrono (body time)                                              OneExpr-Chrono
  (PutPrcData 'COMP body 'chrono (Expr-Chrono (car body) time)))

(setq Lisp-Exeptions '(quote quasiquote GLpushAtrrib))

; traitement d'une expression
(de Expr-Chrono (expr time)                                                 Expr-Chrono
  (if (atom expr) (1+ time)
    (PutPrcData 'COMP expr 'chrono
                (cond
                 ((or (eq (car expr) 'if) ; cas du if
                      (eq (car expr) 'ifn)) ; t(test)

                  (incr time (OneExpr-Chrono (cdr expr) time))
                                          ; t(if-true)
                  (let ((if-t-time (if (caddr expr) (OneExpr-Chrono (cddr expr) time) 0))
                            (if-f-time (if (cdddr expr) (NExpr-Chrono (cdddr expr) time) 0)))
                    (+ time (if (> if-t-time if-f-time) if-t-time if-f-time)))) ; temps du if

                 ((or (eq (car expr) 'cond) ; cas du cond
                      (eq (car expr) 'selectq))

                  (Cond-Chrono (cdr expr) time 0)) ; temps du cond

                 ((eq (car expr) 'lambda) ; cas du lambda

                  (OneExpr-Chrono (cddr expr)
```

```
                                        (if (cdddr expr) (NExpr-Chrono (cdddr expr) time) 0)))


                    ((eq (car expr) 'letf)

                     (OneExpr-Chrono (cdr expr) 0) ; definission d'une nouvelle fonctions
                     (1+ (NExpr-Chrono (cddr expr) time)))

                    ((listp (car expr))    ; cas de la liste en foncteur

                     (NExpr-Chrono (cdr expr)
                                   (OneExpr-Chrono (car expr) time)))

                    ((and (litatom (car expr)) ; cas de FSUBR et FEXPR
                          (or (= (ftyp (car expr)) FEXPR)
                              (member (car expr) Lisp-Exeptions)))

                     (1+ time))

                    (t                                  ; autres cas : arguments evalues
                     (1+ (NExpr-Chrono (cdr expr) time)))))))

        (de Cond-Chrono (sub-expr sub-expr-time max-expr-time aux aux1)                    Cond-Chrono
          (if (atom (car sub-expr)) (+ max-expr-time sub-expr-time)
            (setq aux (OneExpr-Chrono (car sub-expr) sub-expr-time)) ; temps du test
            (setq aux1 (NExpr-Chrono (cdar sub-expr) aux)) ; temps de l'expr correspondante
            (PutPrcData 'COMP sub-expr 'chrono aux1) ; placons le temps de l'expression
            (Cond-Chrono (cdr sub-expr) aux (if (> (- aux1 aux) max-expr-time) (- aux1 aux) max-expr-time)))) ; nou-
velle expre
        sion

        ; Test

        (defmacro Is-Global-Context (var)                                          Is-Global-Context
          `(let (v (GetVarData ,var 'v-obj))
            (cond
             ((null v) nil)
             ((null (cadar v)) t)
             (t (self (cdr v))))))

        (defmacro Var-In-Context (var func)                                        Var-In-Context
          `(or FDD-All-Couples
               (let (v (GetVarData ,var 'v-obj))
              (cond
               ((null v) nil)
               ((or (null (cadar v)) (eq ,func (cadar v))) t)
               (t (self (cdr v)))))))

        ; Execution

        ; modification

        (de Update-Value (what context value test)                                 Update-Value
          (if (or (null (ifn (setq aux (member what context)) nil (setq old-value (cadr aux)) t))
                  (null test)
                  (and test (eval test)))
              (cond
               (aux     (if (cdr aux) (rplaca (cdr aux) value) (rplacd aux (cons value (cddr aux)))))
               (context (rplacd (last context) (cons what (cons value nil)))))))


        (de Update-Var-Context (var func value)                                    Update-Var-Context
          (let (context (if (GetVarData var func) (GetVarData var func) (PutVarData var func '(())) (GetVarData var
func)))
            (Update-Value 'actual context value)
            (Update-Value 'min context value '(let ((v1 (or (and (numbp value) value)
                                                            (and (listp value) (length value))))
                                                    (v2 (or (and (numbp old-value) old-value)
                                                            (and (listp old-value) (length old-value)))))
                                                (and v1 (or (not v2) (< v1 v2)))))

            (Update-Value 'max context value '(let ((v1 (or (and (numbp value) value)
                                                            (and (listp value) (length value))))
                                                    (v2 (or (and (numbp old-value) old-value)
                                                            (and (listp old-value) (length old-value)))))
                                                (and v1 (or (not v2) (> v1 v2)))))))
```

```
; reinitialisation

(de Clear-Var-Context (var)                                          Clear-Var-Context
   (if (litatom var)
  (mapc (append (getival var 'localvar) (getival var 'globalvar))
       (lambda (x)
              (remprop var x)))))
```

Annexe I.8 Génération des analogies : Zprc-compile.vlisp

```
; compilation des définitions de parcours

; ****************************************** ; ;
; première partie : intégration des données ; ;
; ****************************************** ; ;

; syntaxe :
; définition des parcours :
; (define-Parcours <nom-générique>
;   (nom <nom-abbrege>)
;   (titre <titre-des-menus>)
;   (analyse (<fonction> . <indicateur-de-champ>))
;   (global  (<variable> . <fonction-de-calcul>))  : la fonction de calcul recoit le flot calculé en argument
;   (compteurs nom1 nom2 ...)
;   (tests     test1 test2 ...))
;
; définition des tests :
; (define-Test <nom-générique>
;   (nom <nom-déclaré>)
;   (titre <titre-des-menus>)
;   (test-read)  : lecture d'une donnée particulière (<nom-déclaré> . valeur)
;   (test-eval <test>)
;   (next-flow <expression>)
;   (local-var <variable> <expression>)
;   (<compteur> <expression>))
;
; changement de parcours (en next-flow) :
; (change-Parcours NouveauParcours ElementdeCalculduNouveauFlot)
;
; liste des parcours et des tests définis dans Zeugma
(setq Z-Parcours nil)
(setq Z-Tests    nil)
;
; Les domaines d'application :
;
(setq MPV-Application-Domain '((cancel . "   Anulation  ")
                               (remove . "Remise à zéro")
                               (empty  .      "                          ")
                               (fdc-dyn .     "Comportement des fonctions    ")
                               (donnees-dyn . "Comportement des données      ")
                               (instr-dyn .   "Comportement des expressions  ")
                               (empty .       "                          ")))
;
; Fichiers de définitions suivant le domaine
;
(setq MPV-Domain-Source-File '((fonctions-dyn "Meta-Donnees/sources/dynamic.vlisp")
                               (donnees-dyn "Meta-Donnees/sources/aa-data.vlisp")
                               (expr-dyn "Meta-Donnees/sources/animation-instructions.vlisp")))


(deff define-PDV (args)                                              define-PDV
  (if (member (car args) Z-Parcours) (print "Warning : Redéfinition du parcours" (car args))
    (let (p-obj (nextl args))
      (newl Z-Parcours p-obj)
      (mapc args
       (lambda (AnArg)
         (if (listp AnArg) (put p-obj (car AnArg) (cdr AnArg)))))
      (when (get p-obj 'Point-de-vue-type)
```

```
  (rplacd (last MPV-Application-Domain) (cons (cons (car (get p-obj 'nom))
                                                    (car (get p-obj 'titre))) nil))
  (rplacd (last MPV-Domain-Source-File) (cons (cons (car (get p-obj 'nom))
                                                    (get p-obj 'Point-de-vue-type) nil)
                                              nil))))))

(deff define-Test (args)                                                    define-Test
  (if (member (car args) Z-Tests) (print "Warning : Redéfinition du test" (car args))
    (let (t-obj (nextl args))
      (newl Z-Tests t-obj)
      (mapc args
       (lambda (AnArg)
         (if (listp AnArg) (put t-obj (car AnArg) (cdr AnArg))))))))

;
; macros d'accès aux données des parcours et des tests
;
(defmacro PRC-Read-Label (obj) `(car (get ,obj 'test-read)))                PRC-Read-Label
(defmacro PRC-Nom (obj) `(car (get ,obj 'nom)))                                  PRC-Nom
(defmacro PRC-Analyse-Func  (obj) `(caar (get ,obj 'analyse)))              PRC-Analyse-Func
(defmacro PRC-Analyse-Field (obj) `(cdar (get ,obj 'analyse)))              PRC-Analyse-Field
(defmacro PRC-Titre (obj) `(car (get ,obj 'titre)))                              PRC-Titre
(defmacro PRC-Get-Parcours (test) `(get ,test 'parcours))                   PRC-Get-Parcours
(defmacro PRC-Get (obj val) `(car (get ,obj ,val)))                              PRC-Get
(defmacro PRC-Next-Flow (test)                                               PRC-Next-Flow
  `(if (get ,test 'next-flow) (eval (car (get ,test 'next-flow)))
     flow))
(de PRC-Get-From-Nom (nom)                                                PRC-Get-From-Nom
  (let (P Z-Parcours)
    (cond
     ((null P) nil)
     ((eq (PRC-Nom (car P)) nom) (car P))
     (t (self (cdr P))))))

(de PRC-Eval-Local-Vars (aTest)                                          PRC-Eval-Local-Vars
  (mapc (get aTest 'local-var) (lambda (aVar) (eval (cadr aVar)))))


; ************************************************************* ; ;
; Deuxième partie : construction des menus à partir des parcours ; ;
; ************************************************************* ; ;

(de PRC-Make-Menus ()                                                    PRC-Make-Menus
  (setq Read-Test-List nil)
  (setq Conditions-Type
   (let (menu nil)
     (mapc Z-Parcours
               (lambda (aPrc)
                 (PRC-Make-Prc-Menu aPrc)
                 (newl menu (cons (PRC-Nom aPrc) (PRC-Titre aPrc)))))
     menu)))

(de PRC-Make-Prc-Menu (aPrc)                                          PRC-Make-Prc-Menu
  (eval `(setq ,(PRC-Nom aPrc)
         ',(let (pMenu nil)
                 (mapc (get aPrc 'tests)
                           (lambda (aTest)
                             (put aTest 'parcours aPrc)
                             (if (PRC-Read-Label aTest) (newl Read-Test-List aTest))
                             (newl pMenu (cons aTest (PRC-Titre aTest)))))
                 pMenu))))


; ********************************************** ; ;
; troisième partie : construction des analogies ; ;
; ********************************************** ; ;

; lancement de la génération
(de Test-Graphic-Object (obj)                                        Test-Graphic-Object
  ; mode DEBUG ?
  (setq Flow-Verbose (get MPV-Root-Window 'is-verbose))
  (if Flow-Verbose (print "New Analogy for" (get obj 'name)))

  (if exe-dynamic-dump (DD-Dump-Generation obj))

  (cond
```

```
        ((and (get obj 'name) (get obj 'func))
         (MPV-Message (strcat "Génération à partir de l'ORS " (get obj 'name) (get obj 'func)))
         (let (flow (get obj 'func))
           (PutFuncData flow 'root-obj obj)
           ; analyses liées à l'exécution (et donc indépendante du parcours de génération
           (analyse-Dynamique flow)
           ; détection du premier type de parcours (et donc récupération du premier flow parcouru)
           (ifn ga-cond (setq ga-cond 'generiques))
           (PRC-Detect-PRC (get obj 'generation) t)
           ; initialisation des variables globales
           (PRC-Init-Global-Variables (get obj 'func))
           ; maj des valeurs des compteurs
           (PRC-Set-Var-Values)
           ; lancement de la génération
           (PRC-Generate-ORS obj
                             (or (GetData (get obj 'func) (PRC-Analyse-Field (PRC-Get-From-Nom ga-cond)))
                                 (get obj 'func))
                             nil nil)                ; lancement de la generation proprement dite
           (MPV-Reset-Label)))
        (t  (MPV-Message "Ne peut tester cet objet"))))


; évaluation d'un ORS
(de PRC-Generate-ORS (obj flow c-prcs is-replace)                    PRC-Generate-ORS

  (if Flow-Verbose (print 'GORS ga-cond (get obj 'name) flow c-prcs))
  ; partie graphique
  (when (get obj 'gl-data)
    (if (and is-replace (setq aux (IO-Get-List (get obj 'name) flow)))
    (GA-Eval-GL obj aux is-replace) ; depuis Zin-out uniquement... et si l'objet graphique correspondant existe
      (when (setq aux (PRC-Build-GL obj flow)) ; si non on le construit normalement
    (if Flow-Verbose (print "Built GL" aux))
    (if Curent-Object-Transformation (putcar aux Curent-Object-Transformation))
    (ifn ga-build (setq ga-build aux)
         (rplacd (last ga-build) aux)))))
  ; liens vers d'autres ORS
  (when (get obj 'generation)
    (GA-Push ga-build) (setq ga-build nil)
    (GA-Push Curent-Object-Transformation) (setq Curent-Object-Transformation nil)

    (PRC-Generate-From-ORS (or (and (listp flow) (car flow)) flow)
                           (get (get obj 'generation) 'parcour)
                           (get (get obj 'generation) 'objet)
                           (get (get obj 'generation) 'transformations))

    (setq Curent-Object-Transformation (GA-Pop))
    (setq aux (GA-Pop))                                  ; aux contiendra les objets graphiques générés avant la
génération

    (if Flow-Verbose (print (get obj 'name) "aux" aux "ga-build" ga-build))
    ; construction de la pile des objets graphiques construits
    (cond
     ((null ga-build) (setq ga-build aux))
     (t
      (when Curent-Object-Transformation
    (setq ga-build [ga-build])
    (ifn (atom (caar ga-build))
         (putcar ga-build Curent-Object-Transformation))
    (setq Curent-Object-Transformation nil))
      (when aux
    (rplacd (last aux) ga-build)
    (setq ga-build aux))))))


; génération à partir d'un objet

; GetTestData est utilisé dans le cas de données liées à un test de parcours
(defmacro GetTestData () '(if (listp (car prcs)) (cdar prcs)))                    GetTestData

; procédure de génération à partir d'un flot et d'un ORS
(de PRC-Generate-From-ORS (car-flow prcs objs trans)                    PRC-Generate-From-ORS
  (while prcs
    ; le test courant
    (setq Flow-Curent-Method (or (and (listp (car prcs)) (caar prcs)) (car prcs)))

    (if Flow-Verbose (print (get obj 'name) "-> ?" Flow-Curent-Method objs car-flow))
```

```
            (when (and (or (neq ga-cond 'EXE)
                           (eq ga-cond (PRC-Nom (PRC-Get-Parcours Flow-Curent-Method))))
                       (eval (PRC-Get Flow-Curent-Method 'test-eval)))
              ; maj des compteurs
              (PRC-Set-Values)
              ; maj des variables locales
              (PRC-Eval-Local-Vars Flow-Curent-Method)
              ; maj des transformations
              (if (car trans) (GA-Push-Transformation (car trans) obj))
              ; génération à partir de l'ORS
              (cond
               ((PRC-Changing-PRC Flow-Curent-Method)
                ; le test courant implique un changement de type de parcours
               (PRC-Change-PRC (GetObjectByName (car objs)) Flow-Curent-Method (PRC-Next-Flow Flow-Curent-
Method is-replace)))
               (t
               (PRC-Generate-ORS (GetObjectByName (car objs)) (PRC-Next-Flow Flow-Curent-Method) Flow-Curent-
Method is-replace)))
              ; pop des transformations
              (if (car trans) (GA-Pop-Transformation (car trans) obj))
              ; réinitialisation des compteurs
              (PRC-Reset-Values))
             (nextl prcs)
             (nextl objs)
             (nextl trans)))

        ; detection d'un parcours initial entrainant la construction d'analyses et d'un flot à parcourir
        ; gen-obj = objet initial
        ; is-init = intialisation d'un parcours
        ; is-test = recherche d'un parcours particulier
        (de PRC-Detect-PRC (gen-obj is-init is-test)                                    PRC-Detect-PRC
          (setq ORS-done nil)
          (PRC-do-detect-PRC gen-obj))

        (de PRC-do-detect-PRC (gen-obj)                                                 PRC-do-detect-PRC
          (let ((Tests (get gen-obj 'parcour)) (aTest))
            (setq aTest (or (and (listp (car Tests)) (caar Tests)) (car Tests)))
            (cond

             ((null Tests)
              ; parcours descendant sur les objets générés
              (let (gen-objs (get gen-obj 'objet))
               (cond
                ((null gen-objs) nil)

                ((and (not (member (car gen-objs) ORS-done))
                      (setq aux (PRC-do-detect-PRC (get (GetObjectByName (newl ORS-done (car gen-objs)))
'generation))))
                 aux)

                (t (self (cdr gen-objs))))))

             ((and is-test (eq is-test (setq aux (PRC-Get-Parcours aTest))))
              (neq (PRC-Nom aux) 'generiques))

             ((and (null is-test)
              (PRC-Get (setq aux (PRC-Get-Parcours aTest)) 'analyse)
              (neq (PRC-Nom aux) 'generiques))

              (or (and is-init (PRC-Init-PRC aTest))
              (PRC-Nom aux)))

              (t (self (cdr Tests))))))

        ; detection d'un changement de parcours
        (de PRC-Changing-PRC (aTest)                                                    PRC-Changing-PRC
          (and (not (eq ga-cond (PRC-Nom (PRC-Get-Parcours aTest))))
               (neq (PRC-Nom (PRC-Get-Parcours aTest)) 'generiques)
               (get (PRC-Get-Parcours aTest) 'analyse)))

        ; changement de structure programmatoire parcourue
        (de PRC-Change-PRC (nextORS aTest flow is-replace)                              PRC-Change-PRC
          (if Flow-Verbose (princ "Changement de parcours" (PRC-Nom (PRC-Get-Parcours aTest)) (get nextORS
'name) flow))
          (MPV-Message (strcat "Nouveau parcours " (PRC-Nom (PRC-Get-Parcours aTest)) " pour " flow))
```

```
  (PRC-Push-PRC)
  (PRC-Init-PRC aTest flow)
  (PRC-Eval-Local-Vars Flow-Curent-Method)
  (PRC-Generate-ORS nextORS flow aTest is-replace)
  (PRC-Pop-PRC)))

; calcul du déplacement dans les arborescences
(de PRC-Make-Flow-NLG (lst)                                          PRC-Make-Flow-NLG
  (let (atm (gensym))
    (put atm 'cont (gensym))
    (let ((l lst) (pr 1))
      (cond
       ((or (atom l) (null l)) t)
       (t
        (put atm pr (+ 1 (get atm pr)))
    (if (listp (car l)) (self (car l) (+ 1 pr)))
    (self (cdr l) pr))))
    atm))

(de PRC-Get-NLG ()  (get Flow-NLG ipr))                              PRC-Get-NLG
(de PRC-Next-ILG ()                                                  PRC-Next-ILG
  (put (get Flow-NLG 'cont) ipr (+ 1 (get (get Flow-NLG 'cont) ipr)))
  (get (get Flow-NLG 'cont) ipr))


; valeurs empilées lors du changement de structure programmatoire parcourue
(de PRC-Push-PRC ()                                                  PRC-Push-PRC
  ; push du nom du parcours
  (GA-Push ga-cond)
  ; push de la structure parcourue
  (GA-Push flow) (GA-Push car-flow)
  ; push des compteurs
  (GA-Push ipr) (GA-Push ilg) (GA-Push iilg) (GA-Push nilg))

(de PRC-Pop-PRC ()                                                   PRC-Pop-PRC
  (setq nilg (GA-Pop)) (setq iilg (GA-Pop)) (setq ilg (GA-Pop)) (setq ipr (GA-Pop))
  (setq car-flow (GA-Pop)) (setq flow (GA-Pop))
  (setq ga-cond (GA-Pop)))

; initialisation des variables globales des parcours définis
(de PRC-Init-Global-Variables (func)                                 PRC-Init-Global-Variables
  (if Flow-Verbose (print "Init Globals"))
  (setq GA-Root obj ; racine de la génération
   ; états de la génération
   Parcours-has-Changed nil ; changement de parcours au milieu d'un parcours
   Flow-Uniq (get obj 'unique) ; génération unique des éléments
   FDD-All-Couples (get MPV-Root-Window 'fdd-all) ; traitement de tous couple var . func

   ; pour le traitement des objets graphiques
   has-drawn nil ga-build nil ; pour le traitement des objets graphiques
   GA-X 0 GA-Y 0 GA-Z 0 ; position des objets
   GA-rot-x 0 GA-rot-y 0 GA-rot-z 0 ; cumul des rotations
   GA-scale-x 1 GA-scale-y 1 GA-scale-z 1 GA-scale-w 1 ; cumul des changements d'échelles
   t-x 0 t-y 0 t-z 0  ; pour le calcul des variables précédentes
   r-x 0 r-y 0 r-z 0
   s-x 1 s-y 1 s-z 1 s-w 1
   ; compteurs globaux
   ipr 0 ilg 0 iilg 0 nilg 0
   ; progression dans le parcours
   actuel nil pere nil new-pere nil flow-instr nil
   flow nil car-flow nil) ; et enfin la déclaration du flot...

  ; parcours des parcours pour traiter les variables globales de chacun...
  (mapc Z-Parcours
   (lambda (prc)
      ; variables globales du parcours
     (mapc (get prc 'global)
           (lambda (g-var)
             (if (and (litatom (cdr g-var)) (> (ftyp (cdr g-var)) 0))
                 (eval `(setq ,(car g-var) (,(cdr g-var) ',(GetPrcData (PRC-Nom prc) func (PRC-Analyse-Field
prc)))))
                 (eval `(setq ,(car g-var) ,(cdr g-var))))))
     ; initialisation des variables locales des tests du parcours
     (mapc (get prc 'tests)
           (lambda (aTest)
             (mapc (get aTest 'local-var)
```

```
                                (lambda (aVar)
                                  (eval `(setq ,(car aVar) nil))))))
        ; initialisation des compteurs
;       (mapc (get prc 'compteurs)
;                     (lambda (aCpt)
;                       (print aCpt)
;                       (eval `(setq ,aCpt 0))))
        )))


; initialisation du parcours d'une structure programmatoire
(de PRC-Init-PRC (aTest aFlow)                                          PRC-Init-PRC
  (setq ga-cond 'generiques)
  (let ((prc (PRC-Get-Parcours aTest))
   (func (or aFlow (and (listp flow) (car flow)) flow)))
    (if Flow-Verbose (print "Init PRC" prc "from" aTest func))
    (when (PRC-Get prc 'analyse)
      (setq ga-cond (PRC-Nom prc))
      (unless (GetData func (PRC-Analyse-Field prc))
    (eval `(,(PRC-Analyse-Func prc) ',func)))
      (setq flow  (GetData func (PRC-Analyse-Field prc))
       car-flow (if (listp flow (car flow) flow))
       ipr 1 ilg 1 iilg 1 nilg (if (listp flow) (length flow) 0)
       nlg nilg)))
  flow)

; gestion des compteurs
;
(de PRC-Set-Var-Values ()                                               PRC-Set-Var-Values
  (let (cpts (get (PRC-Get-From-Nom ga-cond) 'compteurs))
    (if Flow-Verbose (print cpts ipr ilg iilg nilg nlg))
    (if (1 cpts) (eval `(setq ,(1 cpts) ipr)))
    (if (2 cpts) (eval `(setq ,(2 cpts) ilg)))
    (if (3 cpts) (eval `(setq ,(3 cpts) iilg)))
    (if (4 cpts) (eval `(setq ,(4 cpts) nilg)))
    (if (5 cpts) (eval `(setq ,(5 cpts) nlg)))))

(de PRC-Set-Values ()                                                   PRC-Set-Values
  (let (cpt-name '(ipr ilg iilg nilg nlg))
    (mapc '(1 2 3 4 5)
     (lambda (n-var)
       (when (and (PRC-Get Flow-Curent-Method (n-var cpt-name))
                  (n-var (get (PRC-Get-Parcours Flow-Curent-Method) 'compteurs)))
         (eval
          `(progn
                (GA-Push ,(n-var cpt-name))
                (setq ,(n-var cpt-name) ,(PRC-Get Flow-Curent-Method (n-var cpt-name)))
                (setq ,(n-var (get (PRC-Get-Parcours Flow-Curent-Method) 'compteurs)) ,(n-var cpt-name))
                ,(n-var cpt-name)))))))
  (GA-Push Flow-Curent-Method))


(de PRC-Reset-Values ()                                                 PRC-Reset-Values
  (setq Flow-Curent-Method (GA-Pop))
  (let (cpt-name '(ipr ilg iilg nilg nlg))
    (mapc '(5 4 3 2 1) ; ordre inverse pour la place dans la pile...
     (lambda (n-var)
       (when (and (PRC-Get Flow-Curent-Method (n-var cpt-name))
                  (n-var (get (PRC-Get-Parcours Flow-Curent-Method) 'compteurs)))
         (eval
          `(progn
                (setq ,(n-var cpt-name) (GA-Pop))
                (setq ,(n-var (get (PRC-Get-Parcours Flow-Curent-Method) 'compteurs)) ,(n-var cpt-
name)))))))))
```

Annexe I.9 Génération des analogies (partie graphique) : Zgraphiques.vlisp

```
; division entière avec approximation au suppérieur si la division flotante donne
; une première décimale > 5.
(defmacro Mdiv (v d) `(if (> (rem ,v ,d) (/ ,d 2)) (1+ (/ ,v ,d)) (/ ,v ,d)))                    Mdiv


                                        ; ************************** ; ;
                                        ; Generation flow / Graphique ; ;
                                        ; ************************** ; ;

; macros de récupération des informations à partir de la pile des opérations  graphique

; les différentes informations sauvegardées sont :
; si le parcours est de type FDD
(defmacro is-var-dep     (elem) `(or (eq (car ,elem) 'fdd-VAR) (eq (car ,elem) 'fdd-FUNC)))     is-var-dep
(defmacro GL-Get-TDEP    (elem) `(if (is-var-dep ,elem) (1 ,elem)))                             GL-Get-TDEP
(defmacro GL-Get-DDEP    (elem) `(if (is-var-dep ,elem) (2 ,elem)))                             GL-Get-DDEP
; dans tous les cas :
; l'ORS racine de la génération
(defmacro GL-Get-GA-Root (elem) `((or (and (is-var-dep ,elem) 3) 1) ,elem))                     GL-Get-GA-Root
; la fenêtre GL contenant la représentation graphique
; (defmacro GL-Get-Meta-GL (elem) `((ifn (is-var-dep ,elem) 2 4) ,elem))
; l'ORS générateur de l'objet graphique courant
(defmacro GL-Get-Object  (elem) `((or (and (is-var-dep ,elem) 4) 2) ,elem))                     GL-Get-Object
; tag correspondant au nom graphique retourné lors d'une sélection
(defmacro GL-Get-Tag     (elem) `((or (and (is-var-dep ,elem) 5) 3) ,elem))                     GL-Get-Tag
; liste de l'objet graphique :
; la liste est structurée de la manière suivante :
; - liste + curent-to-object = liste correspondante à l'objet graphique lui-même,
;   si une autre liste est appelée par dans liste c'est le nouvel objet qui prendra la place
; - (liste-1) = liste réservée à l'ajout d'objets dans la représentation au cours de l'exécution
; - (liste-1) + curent-to-object = même principe que précédemment
(defmacro GL-Get-LList   (elem) `(nth (or (and (is-var-dep ,elem) 6) 4) ,elem))                 GL-Get-LList
(defmacro GL-Get-List    (elem) `(car (GL-Get-LList ,elem)))                                    GL-Get-List
; transformations grahpiques ayant aboutis au placement de l'objet graphique
(defmacro GL-Get-Transfo (elem) `((or (and (is-var-dep ,elem) 7) 5) ,elem))                     GL-Get-Transfo
; objet graphique suivant dans la pile
(defmacro GL-Get-Next    (elem) `(nth (or (and (is-var-dep ,elem) 8) 6) ,elem))                 GL-Get-Next

; valeurs pour les transformations :
; les valeurs elles-mêmes
(defmacro GL-Get-Transfo-Values (elem) `(3 (GL-Get-Transfo ,elem)))                            GL-Get-Transfo-Values
; liste correspondant au push des transformations
(defmacro GL-Get-Transfo-Tag    (elem) `(car (last (GL-Get-Transfo-Values ,elem))))            GL-Get-Transfo-Tag

; correspondant pour modifier les valeurs
(defmacro GL-Set-GA-Root (elem val) `(rplaca (nth (or (and (is-var-dep ,elem) 3) 1) ,elem) ,val))GL-Set-GA-Root
; (defmacro GL-Set-Meta-GL (elem val) `(rplaca (nth 2 ,elem) ,val))
(defmacro GL-Set-Object  (elem val) `(rplaca (nth (or (and (is-var-dep ,elem) 4) 2) ,elem) ,val))GL-Set-Object
(defmacro GL-Set-Tag     (elem val) `(rplaca (nth (or (and (is-var-dep ,elem) 5) 3) ,elem) ,val))   GL-Set-Tag
(defmacro GL-Set-List    (elem val) `(rplaca (nth (or (and (is-var-dep ,elem) 6) 4) ,elem) ,val))   GL-Set-List
(defmacro GL-Set-Transfo (elem val) `(rplaca (nth (or (and (is-var-dep ,elem) 7) 5) ,elem) ,val))GL-Set-Transfo


(defmacro GI-Get-a-GL (obj)                                                                    GI-Get-a-GL
  `(let ((f-obj ,obj) (Prc Z-Parcours))
     (cond
      ((null Prc) (if (listp f-obj) (self (car f-obj) Z-Parcours)))
      ((setq aux (GI-Get-GL f-obj (PRC-Nom (car Prc)))) aux)
      (t (self f-obj (cdr Prc))))))

(defmacro GI-Get-GL (obj aPRC) `(GetPrcData ,aPRC ,obj 'gl-object))                            GI-Get-GL

(de GI-Is-Dep-Var (obj num)                                                                    GI-Is-Dep-Var
  (if (litatom obj)
      (let (gl-o (GI-Get-GL obj 'FDD))
    (cond ((null gl-o) nil)
          ((and (= num (GL-Get-Tag (car gl-o)))
                (is-var-dep (car gl-o))) (car gl-o))
          (t (self (cdr gl-o)))))))

(de GL-Get-GL-From-ORS (f-elem ors func-on-var)                                                 GL-Get-GL-From-ORS
  (if f-elem
      (let (Prcs Z-Parcours)
    (when Prcs
      (or
       (let (gl-data (GetPrcData (PRC-Nom (car Prcs)) f-elem 'gl-object))
```

```
            (cond
             ((null gl-data)
              (if (listp f-elem) (GL-Get-GL-From-ORS (car f-elem) ors func-on-var)))
             ((and (eq ors (GL-Get-Object (car gl-data)))
                   (or (null func-on-var) ; on ne parle pas de variable
                       (and (is-var-dep (car gl-data))
                            (eq func-on-var (GL-Get-DDEP (car gl-data))))))
                   (car gl-data))
              (t (self (cdr gl-data)))))
            (self (cdr Prcs)))))))

(de GL-Get-GL (f-elem list) (car (GL-Get-All-GL f-elem list)))                    GL-Get-GL

(de GL-Get-All-GL (f-elem list)                                                   GL-Get-All-GL
  (if f-elem
      (let (Prcs Z-Parcours)
    (when Prcs
      (or
       (let (gl-data (GetPrcData (PRC-Nom (car Prcs)) f-elem 'gl-object))
        (cond
         ((null gl-data) nil)
         ((eq list (GL-Get-Tag (car gl-data))) gl-data)
         (t (self (cdr gl-data)))))
       (self (cdr Prcs)))))))

(de GetInNameStack (tag stack)                                                    GetInNameStack
  (while (and stack (not (= tag (car stack))))
    (nextl stack)
    (nextl stack))
  stack)

(de GetMetaObjectsFromGraphic (num-obj)                                           GetMetaObjectsFromGraphic
  (let (stk-obj (cadr (GetInNameStack num-obj current-name-stack)))
    (if (or (listp stk-obj)
        (GL-Get-GL stk-obj num-obj)) stk-obj)))

(de Find-Object-List-From-Tag (tag)                                               Find-Object-List-From-Tag
  (ifn (boundp 'current-name-stack) (setq current-name-stack GA-Name-Stack))
  (GL-Get-List (GL-Get-GL (cadr (GetInName tag current-name-stack)) tag)))



                                                ; ****************************** ; ;
                                                ; opérations graphiques de Zeugma ; ;
                                                ; ****************************** ; ;

; génération d'un graphique
(de PRC-Build-GL (obj elem)                                                       PRC-Build-GL

  (setq instr (setq com-instr (or (and (not (listp elem)) elem) car-flow)))
  (if Flow-Verbose (princ "Building" (get obj 'name) elem com-instr))

                                                ; gl-object est l'objet concervant les informations graphique cons-
truites :

  (let (gl-object (GetData instr 'gl-object))
                                                ; premier objet graphique pour l'élément parcouru
    (if (null gl-object) (PutData instr 'gl-object (setq gl-object (cons nil nil))))
                                                ; calcul de la position actuelle
    (GA-Set-Delta)
    (let ((o-list 0) (tag curent-list))
                                                ; placement de tag (nom graphique)
      (NewList (GLloadname curent-list))
      (cond
       ((setq o-list (GA-Eval-GL obj))  ; liste graphique du dessin de l'objet

    (setq GA-Name-Stack (cons tag (cons instr GA-Name-Stack))) ; sauvegarde dans la pile graphique
    (if exe-dynamic-dump (putcar GA-Name-Stack [ilg ipr iilg nilg])) ; placement des valeurs pour le dump dy-
namique

    (setq tag `(,GA-Root ,(get obj 'name) ,tag ,o-list ,GA-Trans-Stack))

    (when (or (eq Flow-Curent-Method 'fdd-FUNC)
              (eq Flow-Curent-Method 'fdd-VAR))
     (setq tag (cons Flow-Curent-Method (cons fdd-func tag))))
```

```
  (if (car gl-object) (rplacd (last gl-object) [tag])
    (rplaca gl-object tag))
  (ifn has-drawn (setq has-drawn t new-pere GA-Trans-Stack))
  [(GL-Get-Tag tag) (get obj 'name)])
      (t
  (if (equal gl-object '(nil))
      (DelData instr 'gl-object)))
  (decr curent-object-list)
  (decr curent-list)
  nil))))

; construction graphique a partir des instructions

(de GA-Eval-GL (obj is-list is-replace)                                    GA-Eval-GL
  (if Flow-Verbose (print "EVAL-GL" (ifn is-list curent-object-list is-list) is-list is-replace (get obj 'gl-data)))
  (cond
   ; ajout d'un élément graphique supplémentaire
   ((eq is-replace 'a)                          ; ajout d'un dessin dynamique
    (let (old-lists (get-car-value is-list 'gl-lists))
      (if old-lists (rplacd (last old-lists) (cons (setq aux (GLgenlist)) nil))
    (replace-car is-list 'gl-lists (setq old-lists (cons (setq aux (GLgenlist)) nil))))
;       (princ old-lists)
      (GLnewlist aux)
      (mapc (get obj 'gl-data) 'eval)
      (GLendlist)
      (if dump-file (DD-Dump-GL aux))
      (GLnewlist (car is-list))
      (mapc old-lists 'GLcalllist)
      (GLendlist)
      (if dump-file (DD-Dump-GL (car is-list)))))
   ; replacement de l'élément graphique
   ((eq is-replace 'r)
    (ReplaceMkList (or is-list (get obj 'gl-list)) '(GA-Eval-GL-Data (get obj 'gl-data))))
   ; reconstruction inutile
   ((and (not (get obj 'gl-recons)) (get obj 'gl-list))
    (NewList (GLcalllist (get obj 'gl-list)))
    (1- curent-list))
   ; construction d'un nouvel objet graphique
   (t
    (let (obj-list)
      ; si on prevoi des instruction graphiques supplementaires au cours de l'execution
      ; il faut leur creer la liste utilise pour ce nouvel objet graphique
      ; le numero de listes sera alors le numero de la liste de l'instruction graphique -1
      (if (or (get obj 'gl-dyn-in)
        (get obj 'gl-dyn-out)) (NewList (eq))) ; (eq) est l'instruction executee lors de la creation...
      (if is-list (setq obj-list is-list)
    (put obj 'gl-list (setq obj-list (- (incr curent-object-list) 1)))
    (put obj 'gl-eval t))
      (MkList obj-list '(GA-Eval-GL-Data (get obj 'gl-data)))
      (1- curent-list)))))

(de GA-Eval-GL-Data (data)                                                 GA-Eval-GL-Data
  (mapc data
   (lambda (x)
     (cond
      ((not (member (car x) '(GLbgn GLfont GLstr))) (eval x))
      ((or (eq (car x) 'GLbgn) (eq (car x) 'GLfont)) ((car x) (strcat (cadr x))))
      ((eq (car x) 'GLstr)
       (if (eq (car (nth 5 x)) '%) (GLstr (cadr x) (caddr x) (cadddr x) (eval (car (nth 6 x))))
         (GLstr (cadr x) (caddr x) (cadddr x) (strcat (car nth 5 x)))))))))

; Generation et sauvegarde des transformation graphiques appliquees sur l'arborescence

(de MkTransfo (name values def-value)                                      MkTransfo
  (when values
   (let ((v1 (if (1 values) (eval (1 values)) nil))
         (v2 (if (2 values) (eval (2 values)) nil))
         (v3 (if (3 values) (eval (3 values)) nil))
         (v4 (if (4 values) (eval (4 values)) nil)))
     (if (and (or v1 v2 v3 v4)
                 (not (and (= v1 def-value)
                           (= v2 def-value)
                           (= v3 def-value)
                           (= v4 def-value))))
         (cons name
                  (cons (ifn v1 def-value v1)
```

```
                                (cons (ifn v2 def-value v2)
                                      (cons (ifn v3 def-value v3)
                                            (cons (ifn v4 def-value v4) nil)))))
            nil))))

        ; Push des transformation

        (de GA-Push-Transformation (trans obj)                                  GA-Push-Transformation
          (when (and (or (listp trans) (cdr trans)) (not GA-Erase))

            (if (atom trans) (setq trans (cdr trans) obj (get obj 'name)))
            (let ((e-trans) (tra) (old-trans) (x) (y) (z))
              (setq x 0 y 0 z 0)

              (while trans
            (when (and (setq aux (assoc (car trans) '((translate t 0) (color c 0) (scale s 1) (rotate r 0))))
                       (setq tra (MkTransfo (cadr aux) (cadr trans) (caddr aux))))

              (if e-trans (rplacd (last e-trans) [tra])
                (setq e-trans [tra]))

              (cond
               ((eq (car tra) 's)
                (setq GA-scale-x (* GA-scale-x (2 tra))   GA-scale-y (* GA-scale-y (3 tra))
                      GA-scale-z (* GA-scale-z (4 tra))   GA-scale-w (* GA-scale-w (5 tra))))
               ((eq (car tra) 'r) (incr GA-rot-x (2 tra)) (incr GA-rot-y (3 tra)) (incr GA-rot-z (4 tra)))
               ((eq (car tra) 't) (setq x (2 tra) y (3 tra) z (4 tra)))))
              (nextl trans)
              (nextl trans))
                                                    ; calcul de la position :
                                                    ; (0 0 0) = position absolue calculee plus loin
                                                    ; e-trans = transformations effectuees pour y arriver
                                                    ; (curent-list 0) = liste graphique de dessin + liste finale de l'objet
(affecte au pop)
              (ifn e-trans (GA-Push t)
              (NewList (GLPushmatrix)
                       (mapc e-trans
                             (lambda (x)
                               (cond
                                ((eq (car x) 'c) (GLcolor (or (and (5 x) 4) 3) (2 x) (3 x) (4 x) (5 x)))
                                ((eq (car x) 's) (GLscale (2 x) (3 x) (4 x) (5 x)))
                                ((eq (car x) 'r) (GLrotate (2 x) (3 x) (4 x)))
                                ((eq (car x) 't) (GLtranslate (2 x) (3 x) (4 x)))))))
              (setq Curent-Object-Transformation e-trans)
              (setq e-trans `(,(cons 0 (cons 0 (cons 0 nil))) ,@e-trans ,(cons (- curent-list 1) (cons 0 nil))))
              (if has-drawn (setq pere new-pere))
              (setq has-drawn nil new-pere nil)
              (GA-Push pere)
                                                    ; old-trans = position de l'objet precedent
              (if GA-Trans-Stack (setq old-trans (car (caddr GA-Trans-Stack))))
                                                    ; sauvegarde dans la trans stack :
                                                    ; flow = parcour dessine
                                                    ; obj  = objet dessine
                                                    ; e-trans = position de l'objet
              (setq GA-Trans-Stack (cons flow (cons obj (cons e-trans GA-Trans-Stack))))

              (let (r-pos (GLgetXYZ (* x scale-factor) (* y scale-factor) (* z scale-factor)
                                    GA-rot-x GA-rot-y GA-rot-z
                                    GA-scale-x GA-scale-y GA-scale-z GA-scale-w))
                                                    ; position calculee
                (rplaca (nth 1 (car e-trans)) (+ (1 r-pos) (1 old-trans)))
                (rplaca (nth 2 (car e-trans)) (+ (2 r-pos) (2 old-trans)))
                (rplaca (nth 3 (car e-trans)) (+ (3 r-pos) (3 old-trans)))))

              (if Flow-Verbose (print "Transfo:" e-trans)))))

        ; Pop des transformation

        (de GA-Pop-Transformation (trans tmp-pop tmp-trans)                     GA-Pop-Transformation
          (when (and (or (listp trans) (cdr trans))
                     (not GA-Erase)
                     (neq (setq tmp-pop (GA-Pop)) t))

            (setq tmp-trans (cdr (caddr GA-Trans-Stack)))
            (rplaca (cdar (last (caddr GA-Trans-Stack))) curent-list) ; sauvegarde de la liste du pop
            (NewList (GLPopmatrix))
```

```
    (mapc (caddr GA-Trans-Stack)
        (lambda (v)
            (cond
             ((or (numbp v) (numbp (car v))) nil)
             ((eq (car v) 'r) (decr GA-rot-x (2 v)) (decr GA-rot-y (3 v)) (decr GA-rot-z (4 v)))
             ((eq (car v) 's) (setq GA-scale-x (Mdiv GA-scale-x (2 v)) GA-scale-y (Mdiv GA-scale-y (3 v))
                                    GA-scale-z (Mdiv GA-scale-z (4 v)) GA-scale-w (Mdiv GA-scale-w
(5 v)))))))
    (if Flow-Verbose (print "Poped:" (caddr GA-Trans-Stack)))
    (setq GA-Trans-Stack (cdddr GA-Trans-Stack))
    (setq pere tmp-pop has-drawn nil)))



; Récupération d'une position relative (rotation et scaling pris en compte) d'un objet
; par rapport a un autre. Le calcul s'effectue par le parcour de la pile des transformations
; jusqu'a la rencontre de l'objet destination

(de GA-Set-Delta (old-trans rot tra sca)                                    GA-Set-Delta

  (setq actuel GA-Trans-Stack
   GA-X (Mdiv (1 (car (3 GA-Trans-Stack))) scale-factor)
   GA-Y (Mdiv (2 (car (3 GA-Trans-Stack))) scale-factor)
   GA-Z (Mdiv (3 (car (3 GA-Trans-Stack))) scale-factor)
   t-x 0 t-y 0 t-z 0 r-x 0 r-y 0 r-z 0 s-x 1 s-y 1 s-z 1 s-w 1 tmp nil
   rot (cdr (assoc 'r (3 GA-Trans-Stack)))
   sca (cdr (assoc 's (3 GA-Trans-Stack)))
   tra (cdr (assoc 't (3 GA-Trans-Stack))))

                                        ; recherche de la position de l'objet precedent
  (if rot (setq r-x (- (1 rot)) r-y (- (2 rot)) r-z (- (3 rot))))
  (if tra (setq t-x (* -1000 (1 tra)) t-y (* -1000 (2 tra)) t-z (* -1000 (3 tra))))
  (if sca (setq s-w (if (4 sca) (4 sca) 1)
                s-x (/ (* 1.0 (1 sca)) s-w)
                s-y (/ (* 1.0 (2 sca)) s-w)
                s-z (/ (* 1.0 (3 sca)) s-w)))
                                        ; boucle sur ga-trans-stack
  (setq old-trans (cdddr GA-Trans-Stack))
  (while (and old-trans (not (eq old-trans pere)))
                                        ; si non.....
    (setq tra (%find (3 old-trans) 't) rot (%find (3 old-trans) 'r) sca (%find (3 old-trans) 's))

    (if sca (setq s-w (if (4 sca) (4 sca) 1)
                  s-x (* s-x (/ (* 1.0 (1 sca)) s-w))
                  s-y (* s-y (/ (* 1.0 (2 sca)) s-w))
                  s-z (* s-z (/ (* 1.0 (3 sca)) s-w))))

    (if rot (setq r-x (- r-x (1 rot))
                  r-y (- r-y (2 rot))
                  r-z (- r-z (3 rot))))

    (if tra (setq tra (GLgetXYZ (* scale-factor (1 tra)) (* scale-factor (2 tra)) (* scale-factor (3 tra)) r-x r-y r-z
s-x s-y s-z)
                  t-x (- t-x (1 tra))
                  t-y (- t-y (2 tra))
                  t-z (- t-z (3 tra))))

    (setq old-trans (cdddr old-trans)))
                                        ; fin du while...
  (setq GA-Delta `(,(Mdiv t-x scale-factor)
                   ,(Mdiv t-y  scale-factor)
                   ,(Mdiv t-z  scale-factor))
   GA-Delta-X t-x
   GA-Delta-Y t-y
   GA-Delta-Z t-z)
  (if Flow-Verbose (print "De" (car pere) "à" (car GA-Trans-Stack) "Delta" GA-Delta))))

                                        ; ****************************************** ; ;
                                        ; Generation des listes des objects dynamiques ; ;
                                        ; ****************************************** ; ;

(de GA-Create-Dynamic-Object-Lists ()                    GA-Create-Dynamic-Object-Lists
  (mapc Objects-List
   (lambda (x)
                                        ; traitement des objets susceptibles d'êtres activés à l'exécution
```

```
        (when (and (get x 'action-g)
                                        ; n'ayant pas encore de listes
                   (null (get x 'gl-list)))
                                        ; ces objets utilisent 3 listes :
                                        ; une pour le pushmatrix et la transformation de deplacement
(champ 'list)
                                        ; une pour le dessin lui-meme 'gl-list
                                        ; une pour le popmatrix
          (put x 'list (1- (NewList (GLPushmatrix))))
          (put x 'gl-list curent-list) (NewList t)
          (put x 'gl-eval nil) ; les instructions graphiques de l'objet n'ont pas été exécutées.
          (NewList (GLpopmatrix))))))


; ******************* ; ;
; routines graphiques ; ;
; ******************* ; ;

; recuperation du X angulaire..
(defmacro GetXR (r a) `(car (GLgetXYangulaire 0 0 (or ,r 0) (or ,a 0))))                    GetXR

; recuperation du Y angulaire..
(defmacro GetYR (r a) `(cadr (GLgetXYangulaire 0 0 (or ,r 0) (or ,a 0))))                   GetYR

; dessin d'un cercle
(de GLcercle (r) (GLdisk r (1+ r)))                                                        GLcercle


; pour ne pas dépasser la taille de la pile de projection
(setq num-pushed 0
      max_pushed 30) ; pour conserver une sécurité d'utilisation de push
                                        ; dans des objets graphiques

;(defmacro GLPushmatrix () '(GLpushmatrix))
(de GLPushmatrix ()                                                                        GLPushmatrix
  (if (> (incr num-pushed) max_pushed) (print "Max GL Matrix Pushed reached"))
  (GLpushmatrix))

;(defmacro GLPopmatrix () '(GLpopmatrix))
(de GLPopmatrix ()                                                                         GLPopmatrix
  (if (< (decr num-pushed) 0) (print "Watchout for Matrix underflow..."))
  (GLpopmatrix))


; Definition d'un objet graphique avec incrementation de curent-list
(de MkList (o-list instr)                                                                  MkList
  (GLwinset Meta-GL)
  (GLpushnewlist curent-list o-list)
  (if (< num-pushed max_pushed) (eval instr))
  (GLendlist)
  (incr curent-list)
  o-list)

(de ReplaceMkList (o-list instr)                                                           ReplaceMkList
  (GLwinset Meta-GL)
  (GLpushnewlist o-list (+ o-list curent-to-object))
  (if (< num-pushed max_pushed) (eval instr))
  (GLendlist)
  (if dump-file (DD-Dump-GL o-list)))

(de AddList (o-list instr added)                                                           AddList
  (GLwinset Meta-GL)
  (GLnewlist added)
  (if (< num-pushed max_pushed) (eval instr))
  (GLendlist)
  (GLnewlist o-list) (GLcalllist (+ o-list curent-to-object)) (GLcalllist added) (GLendlist)
  (if dump-file (DD-Dump-GL o-list)))


(df NewList (l)                                                                            NewList
   (GLwinset Meta-GL)
   (GLpushnewlist curent-list curent-object-list)
   (if (< num-pushed max_pushed) (mapc l 'eval))
   (GLendlist)
   (if dump-file (DD-Dump-GL curent-list))
   (incr curent-list)
```

```
   (incr curent-object-list))

(df ReplaceNewList (listargument)                                    ReplaceNewList
   (let (l-num (eval (nextl listargument)))
     (GLwinset Meta-GL)
     (GLpushnewlist l-num (+ l-num curent-to-object))
     (if (< num-pushed max_pushed) (mapc listargument 'eval))
     (GLendlist)
     (if dump-file (DD-Dump-GL l-num))))
```

Annexe I.10 Définition des ORS : ZORS.vlisp

```
                                        ; ************************************* ; ;
                                        ; Definition des fonctions liees aux menu ; ;
                                        ; ************************************* ; ;

(de Select-Yes-No-Value (obj field lw w x y)                         Select-Yes-No-Value
  (put obj field (not (get obj field)))
  (xSetValues lw "label" (strcat (get obj field))))

(de Get-String-Value (obj field lw w x y)                            Get-String-Value
  (xCreateDialogBox (strcat "valeur pour " field) "nil"
                    `(Set-String-Value xDialogValue ',obj ',field ',lw)))

(de Set-String-Value (str obj field lw w)                            Set-String-Value
  (xSetValues lw "label" str)
  (put obj field str))


                                        ; ********************** ; ;
                                        ; gestion de Objects-List ; ;
                                        ; ********************** ; ;

(de GetObjectByName (name o-list is-include)                         GetObjectByName
  (when (and name (or o-list (setq o-list Objects-List)))
   (while (and o-list (not (equal name (get (car o-list) 'name))))
      (nextl o-list))
   (if (null (or is-include o-list)) (print "Object" name "not found"))
   (car o-list)))


(de GetGlobalObjects (prop)                                          GetGlobalObjects
  (let (ret nil)
    (mapc Objects-List
     (lambda (x)
       (if (get x prop)
               (if (eq prop 'v-obj) (setq ret (cons [(get x 'name) (eq (get x 'v-obj) 'local)] ret))
                 (setq ret (cons (get x 'name) ret))))))
   ret))

                                        ; ********** ; ;
                                        ; ORS racine ; ;
                                        ; ********** ; ;

; Installation de la fenetres de specifications

(de Install-Spec-Menu (obj w)                                        Install-Spec-Menu

  (ifn (or w (setq w (Is-Menu-Installed obj)))
      (let (root (xCreateWidget '--AS-- W-AShell
                                "title" (strcat "Spécification de " (get obj 'name))
                                "iconName" (strcat "Spec " (get obj 'name))))
   (setq w (xCreateWidget '--DW-- W-Draw root "width" 340 "height" 175 "background" W-BColor))
   (ifn (boundp 'Spec-Menu-Wdgs) (setq Spec-Menu-Wdgs [w])
       (setq Spec-Menu-Wdgs (cons w Spec-Menu-Wdgs)))
   (xRealize root))
     (if (get w 'obj) (Remove-All-Methode-Menu (get w 'obj))))
```

```
  (put w 'obj (or obj (setq obj (gensym))))
  (put obj 'root obj)
  (MPV-Add obj (if (setq aux (PRC-Detect-PRC (get obj 'generation))) aux "Ind."))
  (Methode-Menu obj w 10 10 Initial-Menu nil "foreground" W-ComFg "background" W-ComBg)
  (if (get obj 'func) (FM-Create (get obj 'func))))

; déjà présent ?
(de Is-Menu-Installed (obj)                                          Is-Menu-Installed
  (ifn (boundp 'Spec-Menu-Wdgs) nil
       (let (lst Spec-Menu-Wdgs)
    (cond
     ((null lst) nil)
     ((eq obj (get (car lst) 'obj)) t)
     (t (self (cdr lst)))))))

; Lancement de l'affichage de la description d'un objet ; ;

(de Describe-Object (obj w x y)                                      Describe-Object
  (if (get obj 'descr) (aide nil nil (get obj 'descr))))

; sélection du point d'entrée du programme

(de Select-Analysed-Function (obj field lw w x y)                    Select-Analysed-Function
    (let (lst (xCreateList "Fonctions Utilisateur" all_functions 2 default-font 290))
      (xAddCallback lst W-ClBck (strcat "(Select-a-Function '" lst " '" lw " '" obj " '" field " '$O)"))))

(de Select-a-Function (wlst lw obj field value)                      Select-a-Function
  (let (p-type (if (setq aux (PRC-Detect-PRC (get obj 'generation))) aux "Ind.")) ; màj de l'interface principale
    (MPV-Delete (get obj 'func) p-type obj)
    (put obj field (implode (explode value)))
    (MPV-Add    obj p-type))
  (FM-Create (get obj 'func)) ; affichage des données du programme
  (xSetValues lw "label" value)
  (xRemove wlst))

; sélection d'une variable

(de Select-DEP-Variable (obj field lw w x y)                         Select-DEP-Variable
    (let (func (get obj 'func))
      (ifn (= (ftyp func) 0)
      (let (lst (xCreateList "liste des variables"
                                    (append '("Toutes") (%find (ival func) 0) (%find (ival func) 1))
                                    1 default-font XW-Vars YW-Vars))
        (xAddCallback lst W-ClBck (strcat "(Select-One-Var '" lst " '" obj " '" field " '" lw " '$O $l)"))))))

(de Select-One-Var (wlst obj field lw var ind)                       Select-One-Var
    (setq XW-Vars (xGetValues wlst "width"))
    (setq YW-Vars (xGetValues wlst "height"))
    (xRemove wlst)
    (xSetValues lw "label" var)
    (if (> ind 1)
   (put obj field (implode (explode var)))))

; Fin du menu Spec...

(de Quit-Spec-Menu (obj w x y)                                       Quit-Spec-Menu
  (Remove-All-Methode-Menu obj)
  (xRemove w)
  (if (eq (get (car Spec-Menu-Wdgs) 'obj) obj) (nextl Spec-Menu-Wdgs)
    (let (lst Spec-Menu-Wdgs)
      (cond
       ((null (cdr lst)) nil)
       ((eq obj (get (cadr lst) 'obj)) (rplacd lst (cddr lst)))
       (t (self (cdr lst)))))))


                                        ; ********************************* ; ;
                                        ; Selection d'un objet par son nom ; ;
                                        ; ********************************* ; ;

(de Edit-One-Object (obj w x y)                                      Edit-One-Object
    (Select-Object obj 'name nil nil 0 0 'edit))

; installation de la liste
(de Select-Object (obj field lw w x y context)                       Select-Object
```

```
  (let ((l-obj Objects-List)
   (l nil))
    (while l-obj (newl l (strcat (get (nextl l-obj) 'name))))
    ; la liste des choix est fonction du contexte indiqué par le nom du champ
    (setq l (append (cond ((eq field 'name)
                           (if context '("Annulation" "Effacer-l-objet" "Nouveau-Nom")
                             '("Annulation" "Nouveau-Nom")))
                          (t  '("Annulation")))
                    l))
    (let (l-w (xCreateList "Selection" l 2 default-font XW-Nom YW-Nom))
      (rplacd l-w `(',obj ',field ',lw ',w ,x ,y ',context))
      (xAddCallback l-w W-ClBck (strcat "(eval (append '(Select-Object-Return '$W '$O $l) (cdr '" l-w "))")))))))

; retour de la sélection
(de Select-Object-Return (wlst s-obj ind obj field lw w x y context)                    Select-Object-Return
  (setq XW-Nom (xGetValues wlst "width"))
  (setq YW-Nom (xGetValues wlst "height"))
  (xRemove wlst)
  (cond
   ((iswidget lw W-Dialog) (xSetValues lw "value" s-obj))
   ((member field '(o-in o-out i-in i-out v-obj))
    (if (> ind 0) (FM-Set-Value obj field w lw s-obj)))
   ((eq field 'name)
    (cond
     ((= ind 0) t)
     ((and context (= ind 1)) (Delete-Object obj field lw w x y))
     ((and (get obj field)
       (or (and context (= ind 2))
           (and (null context) (= ind 1))) (Set-Object-Name obj field lw w x y)))
     (t (if context (New-Object obj w x y (GetObjectByName s-obj))
     (Install-Spec-Menu (GetObjectByName s-obj) w)))))
    (t
     (put obj field (if context [s-obj context] s-obj))
     (xSetValues lw "label" (if context (strcat [s-obj context]) s-obj)))))


                                                ; ********************* ; ;
                                                ; effacement d'un objet ; ;
                                                ; ********************* ; ;

(de Delete-Object (obj field lw w x y)                                                  Delete-Object
    (setq Objects-List (delete-one Objects-List obj))
    (Select-Object nil field lw w x y))

                                                ; ********** ; ;
                                                ; nouvel ORS ; ;
                                                ; ********** ; ;

(de Create-Object (obj w x y)                                                           Create-Object
    (xCreateDialogBox "Nom du nouvel objet" nil
                      `(New-Object ',obj nil 0 0 (put (gensym) 'name xDialogValue))))

(de New-Object (obj w x y new-object)                                                   New-Object
    (cond
     (w
      (mapc (get obj 'Methode-Menu)
       (lambda (x)
         (Remove-Methode-Menu obj x)))
      (mapc (xGetValues w "children") 'xRemoveWidget)
      (Zeugma new-object w))
     (t
      (if (Menu-Present new-object 'C-Menu) t
    (ifn (member new-object Objects-List) (setq Objects-List (cons new-object Objects-List)))
    (let (root (xCreateWidget '--AS-- W-AShell "iconName" (get new-object 'name) "title" (get new-object
'name)))
      (let (w (xCreateWidget '--DW-- W-Draw root "width" 380 "height" 325 "background" W-BColor))
        (xRealize root)
        (Methode-Menu new-object w 10 10 C-Menu nil "foreground" W-ComFg "background" W-ComBg)))))))

                                                ; ************ ; ;
                                                ; Entree du nom ; ;
                                                ; ************ ; ;


(de Set-Object-Name (obj field w lw x y)                                                Set-Object-Name
    (xCreateDialogBox "Nom" (get obj field) `(Set-Name ',obj ',field ',lw xDialogValue)))
```

```
(de Set-Name (obj field lw name)                                        Set-Name
   (xSetValues lw "label" name)
   (put obj field name)
   (if (eq (get obj 'root) obj)
  (let (p-type (if (setq aux (PRC-Detect-PRC (get ,obj 'generation))) aux "Ind."))
    (MPV-Delete (get obj 'func) p-type obj)
    (MPV-Add obj p-type)))
   (cond
    ((eq field 'name)
     (ifn (member obj Objects-List) (setq Objects-List (cons obj Objects-List)))
     (xSetValues lw "label" name))
    ((eq field 'gl-data)
     (Fill-Methode-Menu-Values obj 'GL-Menu))))

                                        ; ****************************************** ; ;
                                        ; fonctions d'affectation des donnees des ORS ; ;
                                        ; ****************************************** ; ;

; Dismiss du menu

(de Dismiss-C-Menu (obj w x y)                                          Dismiss-C-Menu
   (Remove-Methode-Menu obj 'C-Menu)
   (xRemove w))

; Selection de l'action run-time

(de Select-Run-Action (obj field lw w x y)                              Select-Run-Action
   (Remove-Menu-If-Present obj 'GL-Run-Action)
   (Do-Selection-Menu obj field lw "Action a l'execution"
                      GL-Run-Actions nil
                      "foreground" W-ComFg "background" W-ComBg))

; Entree des transformation dynamiques graphiques GL

(de Select-Run-Transformation (obj field lw w x y)                      Select-Run-Transformation
   (let (root (xCreateWidget '--AS-- W-AShell "title" "Transformation a l'execution"))
     (let (dw (xCreateWidget '--DW-- W-Draw root "width" 300 "height" 160 "background" W-BColor))
  (ifn (get obj field) (put obj field (gensym)))
  (put (get obj field) 'lw lw)
  (put (get obj field) 'is-gl root)
  (Methode-Menu (get obj field) dw 5 5 T-Menu nil "foreground" W-ComFg "background" W-ComBg)
  (xRealize root))))

; attachements automatiques d'un ORS au comportement des programmes
(de Select-Run-Link (obj w x y)                                         Select-Run-Link
   (let (root (xCreateWidget '--AS-- W-AShell "title" "Transformation a l'execution"))
     (let (dw (xCreateWidget '--DW-- W-Draw root "width" 300 "height" 215 "background" W-BColor))
  (Methode-Menu obj dw 5 5 Link-Menu nil "foreground" W-ComFg "background" W-ComBg)
  (xRealize root))))

(de Dismiss-Link-Menu (obj w x y)                                       Dismiss-Link-Menu
  (Remove-Methode-Menu obj 'Link-Menu)
  (xRemove w))

(de Select-Variable-Link (obj field lw w x y)                           Select-Variable-Link
  (Remove-Menu-If-Present obj 'Variables-Link)
  (Do-Selection-Menu obj field lw "Contexte du lien"
                     Variables-Link nil
                     "foreground" W-ComFg "background" W-ComBg))

(de Enter-Variable-Test (obj field lw w x y)                            Enter-Variable-Test
  (Enter-Dialog-Values obj field lw "Test sur variable" "Test"))

; Selection et modification des donnees graphiques liées à un ORS

(de Edit-GL-Data (obj field lw w x y)                                   Edit-GL-Data
   (let (lst (xCreateList "Points" (Make-GL-Points-List (get obj field)) 1 default-font XW-Points YW-Points))
     (rplacd lst `(',obj ',field ',lst ',w ',lw))
     (xAddCallback lst W-ClBck (strcat "(eval (append '(Edit-GL-Callback $l) (cdr "' lst ")))"))))

(de Make-GL-Points-List (lst)                                           Make-GL-Points-List
   (let ((l lst) (r '("      Quiter     " "Nouvelles-Donnees")))
     (if (null l) r (self (cdr l) (append r [(strcat (car l))])))))

(de xCreateWithCallback (wdg lab clb isbmp . props)                     xCreateWithCallback
```

```
(xAddCallback (eval `(xCreateWidget '--WC-- W-Command ,wdg (if ,isbmp "bitmap" "label") ,lab ,@props))
W-ClBck clb))


(de Edit-GL-Callback (ind obj field rw w lw)                                    Edit-GL-Callback
   (setq XW-Points (xGetValues rw "width"))
   (setq YW-Points (xGetValues rw "height"))
   (xRemove rw)
   (cond
    ((= ind 0) (xSetValues lw "label" (strcat (get obj field))))
    (t
     (decr ind)
     (let ((root (xCreateWidget '--AS-- W-AShell "title" "Edit")) (r-cmd))
   (setq r-cmd  (strcat "(Confirm-GL-Edit '" root " '" lw " '" w " '" obj " '" field))
   (let (bx (xCreateWidget '--BX-- W-Box root))
    (let (d1 (xCreateWidget '--AD-- W-Dialog bx
                                    "label" "      Instruction GL      "
                                    "value" (if (= ind 0) "nil" (strcat (car (nth ind (get obj field)))))))
;      (xCreateWithCallback bx      "Liaisons avec les analyses" "(Display-Analytic-Help 'numerical)")
      (xCreateWithCallback bx "OK" (strcat r-cmd  " '" d1 " " ind ")"))
      (xCreateWithCallback bx "Autre point" (strcat r-cmd ")"))
      (xCreateWithCallback bx "Quiter" (strcat "(Quit-GL-Edit '" root ")"))
      (xRealize root)))))))

(de Confirm-GL-Edit (rw lw w obj field dia ind)                                 Confirm-GL-Edit
   (if dia
   (ifn (get obj field) (put obj field [(xGetDialogValue dia)])
    (rplacb (car (nth ind (get obj field))) (xGetDialogValue dia))))
   (xRemoveWidget rw)
   (Edit-GL-Data obj field lw w))

(de Quit-GL-Edit (rw) (xRemoveWidget rw))                                       Quit-GL-Edit

                                        ; ****************************************** ; ;
                                        ; Fonctions specifiques à la composition ORS ; ;
                                        ; ****************************************** ; ;

                                        ; ************** ; ;
                                        ; transformations ; ;
                                        ; ************** ; ;

(de Select-Transformations (obj field lw w x y)                                 Select-Transformations
   (ifn (get obj field) (put obj field (gensym)))
   (put (get obj field) 'lw lw)
   (Methode-Menu (get obj field) w 0 (+ y 80) T-Menu nil "foreground" W-ComFg "background" W-ComBg))

(de Dismiss-T-Menu (obj w x y)                                                  Dismiss-T-Menu
   (Remove-Methode-Menu obj 'T-Menu)
   (if (get obj 'is-gl)
   (progn
    (xSetValues (get obj 'lw) "label" "fixed")
    (xRemove (get obj 'is-gl))
    (remprop obj 'is-gl)
    (remprop obj 'lw))))
(de Enter-Translation (obj field lw x y)                                        Enter-Translation
   (Enter-Dialog-Values obj field lw "Translation" "X" "Y" "Z"))

(de Enter-Rotation (obj field lw x y)                                           Enter-Rotation
   (Enter-Dialog-Values obj field lw "Rotation" "X" "Y" "Z"))

(de Enter-Scaling (obj field lw x y)                                            Enter-Scaling
   (Enter-Dialog-Values obj field lw "Echelle" "X" "Y" "Z" "Fraction"))

(de Enter-Color (obj field lw x y)                                              Enter-Color
   (Enter-Dialog-Values obj field lw "Couleur" "R" "G" "B"))

(defmacro MkLabel (l)                                                           MkLabel
  `(strcat "                " ,l "                "))

(de Enter-Dialog-Values (obj field lw title . vs)                               Enter-Dialog-Values
   (let (root (xCreateWidget '--AS-- W-AShell "title" title))
     (let ((bx (xCreateWidget '--AB-- W-Box root))
       (isone (null (cdr vs)))
       (values (get obj field)))
   (while vs
    (xCreateWidget '--AD-- W-Dialog bx "label" (MkLabel (nextl vs))
```

```
                        "value" (strcat (if isone values (nextl values)))))
  (xCreateWithCallback bx (strcat bitmap-directory "ok.bmp")
                          (strcat "(Confirme-Dialog-Values '" bx " '" obj " '" field " '" lw")")
                          t)
  (xCreateWithCallback bx (strcat bitmap-directory "dismiss.bmp")
                          (strcat "(Confirm-Dialog-Values '" bx ")")
                          t))
   (xRealize root)))

(de Confirm-Dialog-Values (bxw obj field lw)                    Confirm-Dialog-Values
 (when obj
  (let (dws (xGetValues bxw "children"))
   (cond
    ((= (length dws) 3) ; une seule valeur
 (put obj field (s-Value (xGetDialogValue (car dws)))))
    (t ; plusieurs valeurs
 (put obj field (cons (s-Value (xGetDialogValue (nextl dws))) nil))
 (while (cddr dws)
  (put obj field (append (get obj field)
                         (cons (s-Value (xGetDialogValue (nextl dws))) nil))))))))
 (xRemoveWidget (xGetValues bxw "parent"))
 (xSetValues lw "label" (strcat (get obj field))))

                                        ; ********** ; ;
                                        ; objets multiples ; ;
                                        ; ********** ; ;

(de Select-Multiple-Objects (obj field lw w x y)                Select-Multiple-Objects
  (ifn (get obj field) (put obj field (gensym)))
  (ifn (get obj 'm-liste)
  (let (lst (xCreateList (strcat "Objets Générés " (get obj 'name))
                         (Create-Generation-List (get obj field)) 3 default-font XW-Gene YW-Gene))
   (xAddCallback lst W-ClBck (strcat "(Clbk-Generation-Objets '" lst " '" obj " '" field " '" lw " $I)")))))

;
; Creation de la liste d'ORS
;
(de Create-Generation-List (g-obj)                              Create-Generation-List
  (Make-Generation-List '("Fin" " " " "
                          "Nouveau triplet" " " " "
                          "Condition d'activation" "ORS but" "Transformations graphiques")
                        (get g-obj 'parcour)
                        (get g-obj 'objet)
                        (get g-obj 'transformations)))

(de Make-Generation-List (l prcs objs trans)                    Make-Generation-List
  (cond
   ((null prcs) l)
   (t
    (Make-Generation-List `(,@l ,(strcat (car prcs)) ,(car objs) ,(if (cdar trans) "Spécifiées" nil))
                          (cdr prcs) (cdr objs) (cdr trans)))))


(de Clbk-Generation-Objets (wlst obj field lw i)                Clbk-Generation-Objets
  (setq XW-Gene (xGetValues wlst "width"))
  (setq YW-Gene (xGetValues wlst "height"))
  (cond
   ((< i 3)     (xRemove wlst) (xSetValues lw "label" "Fixed"))
   ((< i 6)     (xRemove wlst) (Cond-Object-Edit obj field lw))
   ((< i 9) nil)
   (t           (xRemove wlst) (Cond-Object-Edit obj field lw (1+ (/ (- i 9) 3))))))


(de Cond-Object-Edit (obj field lw index)                       Cond-Object-Edit
  (let ((g-obj (get obj field)) (tmp))
   (let (root (xCreateWidget '--AS-- W-AShell "title" "Nouvel Objet Genere"))
  (let (bx (xCreateWidget '--AB-- W-Draw root "background" W-BColor "width" 300 "height" 315))
   (let ((tps Gen-Types) (n 0))
    (cond
     (tps
      (put g-obj n (if index (car (nth index (get g-obj (caar tps))))))
      (setq tmp (xCreateWidget '--AC-- W-Command bx "font" default-font "y" (+ 5 (* n 25))
                               "label" (strcat (cdar tps) (get g-obj n))
                               "foreground" W-ComFg "background" W-ComBg))
      (xAddCallback tmp W-ClBck
                    (strcat "(Cond-Edit-Switch '" g-obj " " n " '" tmp " '" bx " 100)"))
```

```
            (self (cdr tps) (1+ n)))
           (t
            (xCreateWithCallback bx "OK" (strcat "(Confirm-Object-Edit "' root " "' obj " "' field " "' lw " " index ")") n
l
                                 "foreground" W-ComFg "background" W-ComBg "font" default-font "x" 0
"y" (+ 5 (* n 25)))
            (xCreateWithCallback bx "Dismiss" (strcat "(Dismiss-Object-Edit "' root" "' obj " "' field " "' lw ")") nil
                                 "foreground" W-ComFg "background" W-ComBg "font" default-font "x" 25
"y" (+ 5 (* n 25)))
            (xCreateWithCallback bx "Effacer le triplet"
                                 (strcat "(Confirm-Object-Edit "' root " "' obj " "' field " "' lw " " index " t)") nil
                                 "foreground" W-ComFg "background" W-ComBg "font" default-font "y" (+
30 (* n 25))))))
          (xRealize root)))))

(de Cond-Edit-Switch (obj n lw w x)                                    Cond-Edit-Switch
   (cond
    ((= n 0) (Do-Selection-Menu obj n lw "Condition" Conditions-Type 'List-Cond-Test-PRC
                                "foreground" W-ComFg "background" W-ComBg))
    ((= n 1) (Select-Object obj n lw w x 55))
    ((= n 2) (Select-Transformations obj n lw w x 55))))

; Liste des conditions liées aux tests des parcours
(de List-Cond-Test-PRC (obj field lw w)                                List-Cond-Test-PRC
  (let (t-val (get obj field))
    (cond
     ; a t'on à faire à un parcours ?
     ((setq aux (PRC-Get-From-Nom t-val))
      ; oui : on affiche ses testes
      (eval `(Do-Selection-Menu obj field lw ,(PRC-Titre aux) ,(PRC-Nom aux)
                                ; si un de ses test a besoin de données supplémentaires
                                ; on récurse sur List-Cond-Test-PRC
                                ,(let (tsts (get aux 'tests))
                                   (cond ((null tsts) nil)
                                         ((member (car tsts) Read-Test-List) '(quote List-Cond-Test-PRC))
                                         (t (self (cdr tsts)))))
                                "foreground" W-ComFg "background" W-ComBg)))
     ; non, doit-on saisir une donnée pour le test sélectionné ?
     ((member t-val Read-Test-List)
      ; oui...
      (xCreateDialogBox (PRC-Read-Label t-val) nil
                        `(Set-Cond-Test nil ',obj ',field ',lw xDialogValue))))))


; mise à jours du test sélectionné
(de Set-Cond-Test (lst obj field lw value)                             Set-Cond-Test
   (ifn lst t
   (setq XW-Cond (xGetValues lst "width"))
   (setq YW-Cond (xGetValues lst "height"))
   (xRemove lst))
   (put obj field (cons (get obj field) value))
   (xSetValues lw "label" (strcat (get obj field))))

(de Dismiss-Object-Edit (rw obj field lw)                              Dismiss-Object-Edit
   (xRemoveWidget rw)
   (Remove-Methode-Menu (get (get obj field) 2) 'T-Menu)
   (Select-Multiple-Objects obj field lw))

(de Confirm-Object-Edit (rw obj field lw index is-remove)              Confirm-Object-Edit
   (let ((g-obj (get obj field))
    (tps Gen-Types)
    (n 0))
     (cond
      ((null tps) t)
      (is-remove                (if (numbp index) (put g-obj (caar tps) (delete-nth index (get g-obj (caar tps))
)))
      ((null (get g-obj (caar tps))) (put g-obj (caar tps) [(get g-obj n)]))
      ((numbp index)              (rplaca (nth index (get g-obj (caar tps))) (get g-obj n)))
      (t
   (put g-obj (caar tps) `(,@(get g-obj (caar tps)) ,(ifn (get g-obj n) nil (get g-obj n))))))
     (if tps (self g-obj (cdr tps) (1+ n))))
   (Dismiss-Object-Edit rw obj field lw))


                                        ; ********************** ; ;
                                        ; Dessin du Graphe des ORS ; ;
```

```
                                        ; ********************** ; ;

(de Draw-ORS-Graph (obj w x y)                                                    Draw-ORS-Graph
  (setq --careful-- careful)
  (careful nil)
  (setq GA-Initial-Object obj
   GA-Initial-Window w)
  (let (vp)
    (cond
     (GA-G-Tree
      (setq vp (xGetValues GA-G-Tree "parent")) (xRemoveWidget GA-G-Tree))
     (t
      (let (root (xCreateWidget '--AS-- W-AShell "title" "Graphe des objets" "iconName" "Graphe"))
   (setq vp (xCreateWidget '--VP-- W-Viewport root
                                        "allowVert"  "True" "allowHoriz" "True"
                                        "width" 300 "height" 300)))

      (setq GA-G-Tree (xCreateWidget '--AT-- W-Tree vp)
       GA-G-Parent (xCreateWidget '--AL-- W-Command GA-G-Tree "label" (get obj 'name) "font" Zinfo-font)
       GA-G-Done nil
       ga-comp (get obj 'a-comp))
      (put obj 't-w [GA-G-Parent])
      (xAddCallback GA-G-Parent "callback" (strcat "(New-Object '" GA-Initial-Object  " nil 0 0 '" obj ")"))
      (xRealize (xGetValues vp "parent"))
      (Do-Selection-Menu obj 'action-next nil "Dessin du graphe des objets" GA-GMenu 'GA-Do-Next-Action
                         "foreground" W-ComFg "background" W-ComBg))))
  (careful --careful--))

;
; lancement suivant le type
;
(de GA-Do-Next-Action (obj field lw w x y)  (setq ga-cond 'ALL) ((get obj field) obj nil t))GA-Do-Next-Action

(de GA-Graph-FULL (obj)    (setq ga-display-trans t)   (GA-Graph obj GA-G-Parent))       GA-Graph-FULL
(de GA-Graph-NO-TRANS (obj) (setq ga-display-trans nil) (GA-Graph obj GA-G-Parent))GA-Graph-NO-TRANS

(de GA-Graph (obj parent-w)                                                       GA-Graph
   (let ((g-obj (get obj 'generation))
    (cnds (ifn (eq ga-cond 'ALL) `(,(GA-Cond-ATOM) ,(GA-Cond-CDR) ,(GA-Cond-CAR) incond))))
     (when g-obj
      (setq GA-Graph-Done [(get obj 'name)])
      (GA-Mk-Graph g-obj cnds parent-w))))

(de GA-Mk-Graph (g-obj cnd p-w)                                                   GA-Mk-Graph
  (if g-obj
     (let ((g-objs (get g-obj 'objet))
      (prcs   (get g-obj 'parcour))
      (trans  (get g-obj 'transformations)))
   (when g-objs
     (let (str (strcat "Activation: " (car prcs) "\nORS: " (car g-objs)))
      (setq pa (xCreateWidget '--AL-- W-Command GA-G-Tree "treeParent" p-w "font" Zinfo-font
                                        "label" (if (and ga-display-trans (car trans))
                                                    (strcat str (MkStrTrans (car trans)))
                                                str)))
      (xAddCallback pa "callback" (strcat "(New-Object '" GA-Initial-Object  " nil 0 0 '"
                                                    (GetObjectByName (car g-objs)) ")")))
     (unless (member (car g-objs) GA-Graph-Done)
      (setq GA-Graph-Done (cons (car g-objs) GA-Graph-Done))
      (GA-Mk-Graph (get (GetObjectByName (car g-objs)) 'generation) cnd pa))
     (self (cdr g-objs) (cdr prcs) (cdr trans))))))

(de MkStrTrans (tobj)                                                             MkStrTrans
   (let (str "")
     (if (get tobj 'translate) (setq str (strcat "\nTrans.\n" (get tobj 'translate))))
     (if (get tobj 'rotate) (setq str (strcat str "\nRot.\n" (get tobj 'rotate))))
     (if (get tobj 'scale) (setq str (strcat str "\nEchelle\n" (get tobj 'scale))))
     (if (get tobj 'color) (setq str (strcat str "\nCouleur\n" (get tobj 'color))))
     str))
```

Annexe I.11 Informations sur les fonctions : Zfunction-info.vlisp

```
                ;
================================================================== ; ;
```

```
                ; Première partie : fonctions d'interface de l'information sur les fonctions ; ;
                ;
================================================================================ ; ;


                                                ; ********************************************************* ; ;
                                                ; Gestion du menu d'info sur les fonctions (prédicat FM-); ;
                                                ; ********************************************************* ; ;

        ; Ce menu est lancé à partir de la sélection (avec le bouton 2 de la souris) du
        ; graphe général d'information de l'interface Zeugma.

        (defun FM-Create (func)                                                    FM-Create
          (unless (GetFuncData func 'menu-w)      ; le menu est deja la
            (unless (GetPrcData 'FDC func 'FDC) (analyse-FDC func))
            (unless (GetFuncData func 'vars)    (analyse-Dynamique func))
            (unless (GetPrcData 'FDD func 'FDD) (analyse-FDD func))
            (let (root (xCreateWidget '--AS-- W-AShell "title" (strcat "Programme " func) "iconName" func))
              (let (d-w (xCreateWidget '--AF-- W-Draw root "background" W-BColor "width" 365 "height" 255))
            (PutFuncData func 'menu-w d-w)
            (put d-w 'func func)
            (put d-w 'file (getival func 20))
            (FM-Check-Sup d-w)
            (Methode-Menu d-w d-w 10 10 Func-Menu nil "foreground" W-ComFg "background" W-ComBg)
            (xRealize root)))))


        (defun FM-Quit (obj w x y)                                                 FM-Quit
          (when (GetFuncData (get obj 'func) 'menu-w)
            (PutFuncData (get obj 'func) 'menu-w nil)
            (rplacd obj nil)
            (xRemove obj)))


        ; Verification de la presence d'activite supplementaires (sur les fonctions comme sur les variables)

        (defun FM-Check-Sup (w)                                                    FM-Check-Sup
          (let ((func (get w 'func))
            (t-vals '((o-in . "FUNC") (o-out . "FUNC") (i-in . "INSTR")
                      (i-out . "INSTR") (v-obj . "VAR"))))
             ; parcours d'effacement des triplets (fonction parcours objet) enregistres dans w
             (mapc t-vals
              (lambda (x)
                (mapc (get w (car x))
                      (lambda (y)
                        (MPV-Delete func (cdr x) (GetObjectByName y))))
                (remprop w (car x))))
             ; parcours de recherche d'un attachement dans l'arbre du flux de controle
             (setq func-done ())
             (let (fcs `(,func))
               (cond
                ((null fcs) nil)
                ((member (car fcs) func-done) (self (cdr fcs)))
                (t
            (setq func-done (cons (car fcs) func-done))
            (mapc t-vals
                 (lambda (prcs)
                       ; parcours des (cadr prcs) pour la fonction
                       (mapc (GetFuncData (car fcs) (car prcs))
                            (lambda (y)
                                 ; deja presente ?
                                 (let (obj (if (listp y) (car y) y))
                                   (unless (member obj (get w (car prcs)))
                                             (put w (car prcs) (cons obj (get w (car prcs))))
                                             (ifn (eq (get (GetObjectByName obj) 'func) func) (put (GetObjectByName
obj) 'func func))
                                             (MPV-Add (GetObjectByName obj) (cdr prcs) t)))))))
            (self (getival (car fcs) 4))
            (self (cdr fcs)))))))


        ; Mise en place (ou effacement) d'un attachement a un objet

        ; obj = la fonction / variable en jeu...
        ; field = le champ
        ; w = widget racine du menu : contiend la fonction pour les variables
```

```
; lw = widget label de la valeur
; val = la nouvelle valeur

(defun FM-Set-Value (obj field w lw val)                                    FM-Set-Value
  (cond
   ((eq field 'v-obj) ; traitement special pour les variables : gestion du context : v-obj = ([g-obj . func] ...)

    (cond
     ; recherche de l'existance ...
     ((setq aux (let (l (GetVarData obj field))
                  (cond
                  ((null l) nil)
                  ((and (eq (caar l) val)
                        (or (null (cadar l))
                            (eq (cadar l) (get w 'func))))
                   (car l))
                  (t (self (cdr l))))))
      ; demande du changement de contexte
      (Yes-No-Menu "Changement de contexte (Oui) / Effacement (Non)"
                   `(FM-Switch-Context ',aux ',obj ',field ',w ',lw ',val)
                   `(FM-Delete-Value ',obj ',field ',w ',lw ',val)))
     (t
      ; inexistant demande contextuel
      (Yes-No-Menu (strcat "Contextuel à " (get w 'func))
                   `(FM-New-Context ',obj ',field ',w ',lw ',val ',(get w 'func))
                   `(FM-New-Context ',obj ',field ',w ',lw ',val nil)))))

   (t
    (cond
     ((not (member val (get obj field))) (put obj field (cons val (get obj field))))
     ((eq (car (get obj field)) val)     (put obj field (cdr (get obj field))))
     (t                                  (delete-one val (get obj field))))
    (xSetValues lw "label" (strcat (get obj field)))
    (FM-Check-Sup (get obj 'menu-w))
    (Fill-Methode-Menu-Values (get obj 'menu-w) Func-Menu))))


; gestion du contexte des variables (retour des menu oui/non)

(defun FM-Switch-Context (couple obj field w lw val)                        FM-Switch-Context
  (rplacd couple [(if (cadr couple) nil (get w 'func))])
  (xSetValues lw "label" (strcat (get obj field))))

(defun FM-Delete-Value (obj field w lw val func)                            FM-Delete-Value
                                                ; effacement dans la variable
  (if (eq val (caar (GetVarData obj field))) (put obj field (cdr (get obj field)))
    (let (l (GetVarData obj field))
      (ifn l nil
      (if (and (eq (caar (cdr l)) val)
               (or (null (cadar (cdr l)))
                   (eq (cadar (cdr l)) (get w 'func)))) (rplacd l (cddr l)))
      (self (cdr l)))))
                                                ; verification de la presence de l'objet dans une autre variable
  (setq func (get (get w 'menu-w) 'func))
  (let (v (GetFuncData func 'vars))
    (cond
     ((null v)                                  ; pas trouve : effacement de la v-obj liste de la fonction
      (PutFuncData func field (delete-one val (GetFuncData func field))))
     ((assq val (GetVarData (cadar v) field)) nil) ; trouve
     (t (self (cdr v)))))
  (FM-Setted obj field w lw))

(defun FM-New-Context (obj field w lw val context)                          FM-New-Context
  (PutVarData obj field `(,[val context] ,@(GetVarData obj field)))
  (let (f (get (get w 'menu-w) 'func))
    (ifn (member val (GetFuncData f 'v-obj)) (PutFuncData f 'v-obj (cons val (GetFuncData f 'v-obj)))))
  (FM-Setted obj field w lw))

; post traitement pour les variables
(defun FM-Setted (obj field w lw func val)                                  FM-Setted
  (xSetValues lw "label" (strcat (GetVarData obj field)))
  (FM-Check-Sup (get w 'menu-w))
  (Fill-Methode-Menu-Values (get w 'menu-w) Func-Menu))

                                                ; ********************************************************* ; ;
                                                ; Dessin des graphes des analyses des fonctions et variables ; ;
```

```
                                        ; (lancé de la fenêtre d'info sur le programme étudié)      ; ;
                                        ; ********************************************************* ; ;

; creation de la fenetre de l'arbre
(defun FM-Root-Widget ()                                                          FM-Root-Widget
   (let (root (xCreateWidget '--AS-- W-AShell "title" "Graphe d'analyse" "iconName" "Graphe"))
     (let (vp (xCreateWidget '--AV-- W-Viewport root
                                    "allowVert"  "True" "allowHoriz" "True"
                                    "width" 300 "height" 300))
   (setq FM-Root-Widget (xCreateWidget '--AT-- W-Tree vp "gravity" "left"))
   (xRealize root))))


; nouvel widget
(defun FM-New-Graph-Wdg (obj func par fnc is-var)                                 FM-New-Graph-Wdg
  (let (n-w (xCreateWidget '--AC-- W-Command FM-Root-Widget "treeParent" par "font" "-*-*-*-*-*-*-*-12-*-*-*-
*-*-*-*"))
    (xAddCallback n-w "callback" (strcat "(" fnc " '" func " '" obj " '" is-var ")"))
    (let (lbl func)
      (cond
       (is-var                                  ; Variable
   (setq lbl (strcat "Variable: " lbl "\nFonction: " is-var ))
   (if (GetVarData func 'v-stop)
      (setq lbl (strcat lbl "\nStepper")))
   (if (GetVarData func 'v-obj) (setq lbl (strcat lbl "\n--Liens--")))
   (mapc (GetVarData func 'v-obj)
       (lambda (x)
             (if (or (null (cadr x)) (eq (cadr x) is-var))
                (setq lbl (strcat lbl "\n" (car x)
                                  (if (cadr x) " - Local" " "))))))
      (t                                        ; Fonction
   (if (GetFuncData func 's-in)  (setq lbl (strcat lbl "\nStep in")))
   (if (GetFuncData func 's-out) (setq lbl (strcat lbl "\nStep out")))
   (if (GetFuncData func 'o-in)  (setq lbl (strcat lbl "\nO in  " (GetFuncData func 'o-in))))
   (if (GetFuncData func 'o-out) (setq lbl (strcat lbl "\nO out " (GetFuncData func 'o-out))))
   (if (GetFuncData func 'i-in)  (setq lbl (strcat lbl "\nI in  " (GetFuncData func 'i-in))))
   (if (GetFuncData func 'i-out) (setq lbl (strcat lbl "\nI out " (GetFuncData func 'i-out))))))
     (xSetValues n-w "label" lbl))
    n-w))


                                        ; ********************************************* ; ;
                                        ; construction du graph pour le flux de control ; ;
                                        ; ********************************************* ; ;

(defun FM-Function-Graph (d-w w x y)                                              FM-Function-Graph
  (analyse-FDC (get d-w 'func))
  (analyse-Dynamique (get d-w 'func))
  (analyse-FDD (get d-w 'func))
  (cond
   ((and w (boundp 'FM-Root-Widget) (iswidget FM-Root-Widget))
    (xRemove FM-Root-Widget) (setq FM-Root-Widget 'undef))
   (t
    (ifn (boundp 'FM-Root-Widget) (FM-Root-Widget))
    (xUnmap (xGetValues FM-Root-Widget "parent"))
    (mapc (xGetValues FM-Root-Widget "children") 'xRemoveWidget)
    (let ((fdc (GetPrcData 'FDC (get d-w 'func) 'FDC)) (pa))
      (when fdc
   (PutFuncData (car fdc) 'menu-w d-w)
   (setq aux (FM-New-Graph-Wdg d-w (car fdc) pa 'FM-Function-Info))
   (if (null (setq aux1 (GetFuncData (car fdc) 'tree-w)))
      (PutFuncData (car fdc) 'tree-w [aux])
     (rplacd (last aux1) [aux]))
   (if (atom (cadr fdc)) (self (cdr fdc) pa)
     (self (cadr fdc) aux)
     (self (cddr fdc) pa))))
    (xMap (xGetValues FM-Root-Widget "parent"))))))

                                        ; ********************************************* ; ;
                                        ; construction du graph pour le flux de donnees ; ;
                                        ; ********************************************* ; ;

(defun FM-Variables-Graph (d-w w x y)                                             FM-Variables-Graph
  (cond
   ((and w (boundp 'FM-Root-Widget) (iswidget FM-Root-Widget))
    (xRemove FM-Root-Widget)
```

```
   (setq FM-Root-Widget 'undef))
  (t
   (ifn (boundp 'FM-Root-Widget) (FM-Root-Widget))
   (xUnmap (xGetValues FM-Root-Widget "parent"))
   (mapc (xGetValues FM-Root-Widget "children") 'xRemoveWidget)
   (mapc (GetPrcData 'FDD (get d-w 'func) 'FDD)
    (lambda (v-tree)
     (if (cadr v-tree) (PutFuncData (cadr v-tree) 'menu-w d-w))
     (let ((childs (cddr v-tree)) (pa (FM-New-Graph-Wdg d-w (cadr v-tree) nil 'FM-Var-Info (car v-tree))))
      (cond
       ((or (null childs) (null (car childs))) t)
       ((listp (car childs)) (self (car childs) pa) (self (cdr childs) pa))
       ((listp (caddr childs))
            (setq aux (FM-New-Graph-Wdg d-w (cadr childs) pa 'FM-Var-Info (car childs)))
            (if (null (setq aux1 (GetVarData (cadr childs) 'tree-w)))
               (PutVarData (cadr childs) 'tree-w [aux])
              (rplacd (last aux1) [aux]))
            (self (cddr childs)
                 aux))
       (t
            (setq aux (FM-New-Graph-Wdg d-w (cadr childs) pa 'FM-Var-Info  (car childs)))
            (PutVarData (cadr childs) 'tree-w aux)
            (self (cddr childs) pa))))))
   (xMap (xGetValues FM-Root-Widget "parent")))))


                                        ; ********************************************************* ; ;
                                        ; Fonction d'information sur les fonction et les variables ; ;
                                        ; ********************************************************* ; ;
; information sur les fonctions

(defun FM-Function-Info (func d-w)                                                    FM-Function-Info
  (let (drw (GetFuncData func 'f-info))
    (cond
     (drw
      (mapc (xGetValues drw "children") 'xRemoveWidget)
      (Remove-Methode-Menu func 'Func-Info-Menu))
     (t
      (let (root (xCreateWidget '--AS-- W-AShell "title" func "iconName" func))
   (PutFuncData func `f-info
                  (setq drw (xCreateWidget '--DW-- "Drawxbvl" root
                                           "width" 600 "height" 500
                                           "background" W-BColor
                                           "font" default-font)))
   (xRealizeWaiting root))))
    (xSetValues drw "foreground" W-BColor)
    (xFillRectangles drw 4 0 200 100)
    (xSetValues drw "foreground" "#FFF")
    (xDrawString drw 4 20 (strcat "Variables locales  " (getival func 0)))
    (xDrawString drw 4 35 (strcat "Variables globales " (getival func 1)))
    (xDrawString drw 4 50 (strcat "Fichier source     " (getival func 20)))
    (let (c-vals (GetPrcData 'COMP func 'COMPOSITION))
      (if c-vals
      (let (drw-str "Analyses:")
        (if (> (get c-vals 'local) 0) (setq drw-str (strcat drw-str " loc " (get c-vals 'local))))
        (if (> (get c-vals 'loop) 0)  (setq drw-str (strcat drw-str " loop " (get c-vals 'loop))))
        (if (> (get c-vals 'param) 0)  (setq drw-str (strcat drw-str " param " (get c-vals 'param))))
        (if (> (get c-vals 'lambda) 0)  (setq drw-str (strcat drw-str " lambda " (get c-vals 'lambda))))
        (if (> (get c-vals 'global) 0)  (setq drw-str (strcat drw-str " glob " (get c-vals 'global))))
        (if (> (get c-vals 'modif) 0)  (setq drw-str (strcat drw-str " modif " (get c-vals 'modif))))
        (if (> (get c-vals 'total) 0)  (setq drw-str (strcat drw-str " tot " (get c-vals 'total) " ftyp ")))
        (mapc '(0 1 2 3 4 5 6 7)
                  (lambda (x)
                    (if (> (get c-vals x) 0) (setq drw-str (strcat drw-str " " x " " (get c-vals x))))))
        (xDrawString drw 4 65 drw-str))))
    (put drw 'func func)
    (put drw 'menu-w d-w)
    (mapc Func-Info-Menu
     (lambda (mItem)
       (if (listp mItem) (put func (cadr mItem) (GetFuncData func (cadr mItem))))))
    (Methode-Menu func drw 4 70 Func-Info-Menu nil "foreground" W-ComFg "background" W-ComBg)))

; Fonction d'information sur les variables ; ;

(defun FM-Var-Info (var d-w func)                                                     FM-Var-Info
   (let (drw (GetVarData var 'v-info))
```

```
    (cond
     (drw
  (mapc (xGetValues drw "children") 'xRemoveWidget)
  (Remove-Methode-Menu var 'Var-Info-Menu))
     (t
  (let (root (xCreateWidget '--AS-- W-AShell "title" (strcat var " Dans " func)
                                    "iconName" (strcat var "/" func)))
   (PutVarData var `v-info (setq drw (xCreateWidget '--DW-- "Drawxbvl" root
                                                              "width" 530 "height" 500
                                                              "background" W-BColor
                                                              "font" default-font)))
   (xRealize root))))
    ; placement de la fonction dans la widget racine du menu
    (put drw 'func func)
    (put drw 'menu-w d-w)
    (put drw 'var var)
    (xSetValues drw "foreground" "#fff")
    (xDrawString drw 4 20 (strcat "Local dans    " (getval var 'localvar)))
    (xDrawString drw 4 35 (strcat "Global dans   " (getval var 'globalvar)))
    (xDrawString drw 4 50 (strcat "Modifiee dans " (getval var 'incr)))
    (mapc Var-Info-Menu
     (lambda (mItem)
       (if (listp mItem) (put var (cadr mItem) (GetVarData var (cadr mItem))))))
    (Methode-Menu var drw 4 65 Var-Info-Menu nil "foreground" W-ComFg "background" W-ComBg)))

                                          ; ************************************* ; ;
                                          ; affichage du code source d'une fonction ; ;
                                          ; ************************************* ; ;

(defvar FM-Code-Line-Pos 0) ; variables utilisées pour la trace de l'exécution
(defvar FM-Code-Line nil)

; affichage des sources d'un programme (commande de la fenêtre d'info sur le programme)
(defun FM-Display-Program (d-w w x y)                                    FM-Display-Program
  (let (f (get d-w 'func))
    (unless (GetPrcData 'COMP f 'COMPOSITION) (analyse-COMP f))
                                          ; calcul du nombre total de lignes à afficher
    (let ((fcs (GetPrcData 'FDC f 'liste-func)) (nl 0) (nc 0))
      ; calcul de la taille à afficher (nb lignes, nb colonnes)
      (mapc fcs
       (lambda (aFunc)
         (mapc (GetPrcData 'COMP aFunc 'chaines)
                  (lambda (aLine)
                    (incr nl)
                    (if (> (setq aux (strlen (car aLine))) nc) (setq nc aux))))
         (incr nl))) ; ligne entre chaque fonction
      ; création de la fenêtre d'affichage des sources du programme
      (let (root (xCreateWidget '--AS-- W-AShell "title" (strcat "Sources de " f)
                                    "iconName" f))
   (PutFuncData f 'p-code
                  (xCreateWidget '--DR-- W-Draw
                                       (xCreateWidget '--AV-- W-Viewport root "width" 300 "height" 300
                                                              "allowHoriz" "True" "allowVert"  "True")
                                       "width" (* (+ 2 nc) FontWidth)
                                       "height" (* (1+ nl) FontHeight)))
   (xRealize root)
   (xOverride (GetFuncData f 'p-code) "<Btn1Down>"
                 (strcat "(FM-Marque '" f " " d-w " (1+ (/ (4 (xGetPosPointer '$W)) FontHeight)))"))
   (xOverride (GetFuncData f 'p-code) "<Btn2Down>"
                 (strcat "(FM-Info-Ligne '" f " " d-w " (1+ (/ (4 (xGetPosPointer '$W)) FontHeight)))")))
      ; et maintenant on affiche
      (setq nl 0)
      (mapc fcs
       (lambda (aFunc)
         (PutFuncData aFunc 'p-pos nl)
         (PutFuncData aFunc 'p-code (GetFuncData f 'p-code))
         (setq nl (1+ (FM-Print-Func-Code (GetFuncData f 'p-code) aFunc nl))))))))


; routines d'affichage du code source
(defun FM-Print-Func-Code (wid f y)                                      FM-Print-Func-Code
  (let (l (GetPrcData 'COMP f 'chaines))
    (FM-Print-Code-Line wid (incr y) (caar l) "#55f")
    (mapc (cdr l)
     (lambda (line)
       (FM-Print-Code-Line wid (incr y) (car line)
```

```
                                (or (and (cddr line) W-BColor) W-ComFg))))
  y))

(defun FM-Print-Code-Line (w n line col)                                    FM-Print-Code-Line
 (when (> n 0)
  (xSetValues w "foreground" col)
  (xDrawString w 0 (* n FontHeight) line)))

; affichage d'une ligne en cours d'exécution
(defun FM-Display-Code-Running (func lnum)                                  FM-Display-Code-Running
 (FM-Print-Code-Line (GetFuncData func 'p-code) FM-Code-Line-Pos
                     (car FM-Code-Line)
                     (or (and (cddr FM-Code-Line) W-MColor) W-ComFg))

 (setq FM-Code-Line-Pos (+ lnum (GetFuncData func 'p-pos))
  FM-Code-Line (car (nth lnum (GetPrcData 'COMP func 'chaines))))

 (FM-Print-Code-Line (GetFuncData func 'p-code) FM-Code-Line-Pos
                     (car FM-Code-Line)
                     (or (and (cddr FM-Code-Line) W-MRColor) W-RColor)))

; placement d'une activité supplémentaire sur une ligne de code
(defun FM-Marque (pg d-w lnum)                                              FM-Marque
 ; on cherche la fonction et la ligne sélectionnées
 (let ((aFunc nil) (aLigne nil))
  (let (fcs (GetPrcData 'FDC pg 'liste-func))
   (cond
    ((null fcs) nil) ; sélection en dehors d'une fonction
    ((= lnum (1+ (setq aux (GetFuncData (car fcs) 'p-pos)))) ; sélection du point d'entrée d'une fonction
 (FM-Function-Info (car fcs) d-w))
    ((and (> lnum aux)
     (or (null (cdr fcs))
         (< lnum (GetFuncData (cadr fcs) 'p-pos)))) ; on est dans une fonction
  (when (setq aLigne (car (nth (- lnum aux) (GetPrcData 'COMP (car fcs) 'chaines))))
   (setq aFunc (car fcs))))
    (t (self (cdr fcs)))))
  ; trouvés
  (cond
   ((null aLigne) nil)
   ((null (cddr aLigne))
    (let ((l-obj Objects-List) (l nil))
  (while l-obj (newl l (strcat (get (nextl l-obj) 'name))))
  (newl l "Annulation")
  (let (lstw (xCreateList "Objet Lié" l 2 default-font XW-Nom YW-Nom))
   (xAddCallback lstw W-ClBck
                 (strcat "(FM-Marque-Ligne '$W $I '$O "' aFunc " " lnum ")")))))
   (t
    (rplacd (cdr aLigne) nil)
    (FM-Print-Code-Line (GetFuncData pg 'p-code) lnum (car aLigne) W-ComFg)))))

(defun FM-Marque-Ligne (lw ind obj func lnum aLigne)                        FM-Marque-Ligne
 (print ind obj func lnum)
 (xRemove lw)
 (when (> ind 0)
  (rplacd (cdr (setq aLigne ((- lnum aux) (GetPrcData 'COMP func 'chaines))))
     (cons obj nil))
  (FM-Print-Code-Line (GetFuncData func 'p-code) lnum (car aLigne) W-MColor)))

                                        ; ********************************************* ; ;
                                        ; Fonctions liées au menu fonctions/variables ; ;
                                        ; ********************************************* ; ;

; Switch t/nil d'une valeur (utilisé pour la mise en place du mode pas à pas)
(defun Switch-Stop (obj field lw w x y)                                     Switch-Stop
 (cond
  ((get w 'var)
   (PutVarData obj field (not (GetVarData obj field)))
   (xSetValues lw "label" (strcat (GetVarData obj field))))
  (t
   (PutFuncData obj field (not (GetFuncData obj field)))
   (xSetValues lw "label" (strcat (GetFuncData obj field))))))

; Confirmation de la répercussion des valeurs pour les fonctions d'une arborescence
(defun Dispatch-Func-Data (obj w x y)                                       Dispatch-Func-Data
 (let (done)
  (let (fdc (getival (get w 'func) 4))
```

```
      (cond
       ((or (null fdc) (and (litatom (car fdc)) (= (ftyp (car fdc)) 0))) t)
       ((and (atom (car fdc)) (not (member (car fdc) done)))
    (PutFuncData (car fdc) 's-in  (GetFuncData obj 's-in))
    (PutFuncData (car fdc) 's-out (GetFuncData obj 's-out))
    (PutFuncData (car fdc) 'o-in  (GetFuncData obj 'o-in))
    (PutFuncData (car fdc) 'o-out (GetFuncData obj 'o-out))
    (PutFuncData (car fdc) 'i-in  (GetFuncData obj 'i-in))
    (PutFuncData (car fdc) 'i-out (GetFuncData obj 'i-out))
    (setq done `(,(car fdc) ,@done))
    (self (getival (car fdc) 4))
    (self (cdr fdc)))
       (t (self (cdr fdc))))))
  (FM-Check-Sup (get w 'menu-w) (get w 'func))
  (Fill-Methode-Menu-Values (GetFuncData (get w 'func)) Func-Menu)
  (FM-Function-Graph (get w 'menu-w)))

; Répercussion des données des variables : prise en compte du context
(defun Dispatch-Var-Data (obj w x y)                    Dispatch-Var-Data
  ; le graph des dépendances
  (let (deps (GetFuncData (get (get w 'menu-w) 'func) 'DEP))
    (when deps
      ; une branche
      (let (var-data (car deps))
    (cond
     ((null var-data) t)
     ((listp (car var-data))
      (self (car var-data))
      (self (cdr var-data)))
     (t
      (if (and (eq (cadr var-data) (get w 'var))
                   (eq (car var-data) (get w 'func)))
          ; application aux éléments de la branche
          (Apply-Sub-Var-Tree (get w 'var) (get w 'func) var-data)
        (self (caddr var-data))))))
      (self (cdr deps))))
  (FM-Check-Sup (get w 'menu-w) (get w 'func))
  (Fill-Methode-Menu-Values (GetVarData (get w 'func)) Func-Menu)
  (FM-Variables-Graph (get w 'menu-w)))

; répercussion des données sur une sous branche du flot de données
(defun Apply-Sub-Var-Tree (var func var-tree)           Apply-Sub-Var-Tree
  (cond
   ((null var-tree) t)
   ((listp (car var-tree))
    (Apply-Sub-Var-Tree var func (car var-tree))
    (Apply-Sub-Var-Tree var func (cdr var-tree)))
   (t
    (PutVarData (cadr var-tree) 'v-stop (GetVarData var 'v-stop))
    (PutVarData (cadr var-tree) 'v-obj (copy (GetVarData var 'v-obj)))
    ; verification du context local...
    (mapc (GetVarData (cadr var-tree) 'v-obj)
     (lambda (x)
       (if (and (cadr x) (eq (cadr x) func)) (rplaca (cdr x) (car var-tree)))))
    (Apply-Sub-Var-Tree var func (caddr var-tree)))))

; bouton OK
(defun Confirm-Func-Info-Menu (obj w x y)               Confirm-Func-Info-Menu
  (mapc Func-Info-Menu
   (lambda (mItem)
     (when (listp mItem)
       (PutFuncData obj (get obj (cadr mItem)))
       (remprop obj (cadr mItem)))))
  (Confirm-Info-Menu obj w x y))

(defun Confirm-Var-Info-Menu (obj w x y)                Confirm-Var-Info-Menu
  (mapc Var-Info-Menu
   (lambda (mItem)
     (when (listp mItem)
       (PutVarData obj (get obj (cadr mItem)))
       (remprop obj (cadr mItem)))))
  (Confirm-Info-Menu obj w x y))

(defun Confirm-Info-Menu (obj w x y)                    Confirm-Info-Menu
    (Remove-All-Methode-Menu obj)
    (xRemoveWidget (xGetValues w "parent"))
```

```
 (if (eq obj w) ; cas de la visualisation d'une instruction
(delete-car (get obj 'instr) 'gr-info)
  ; si non...
  (remprop obj 'info-menu)
  (delival (get w 'func) 'gr-info)
  (DelFuncData (get w 'func) 'f-info)
  (when (get w 'var)
   (delival (get w 'var) 'gr-info)
   (DelVarData (get w 'var) 'v-info)))
 t)

; bouton Anulation
(defun Dismiss-Func-Info-Menu (obj w x y)                    Dismiss-Func-Info-Menu
   (Yes-No-Menu "Remise à zéro des valeurs de la fonction" `(Do-Dismiss-Func-Info ',obj ',w)))

(defun Do-Dismiss-Func-Info (obj w)                          Do-Dismiss-Func-Info
  (mapc Func-Info-Menu
   (lambda (mItem)
     (when (listp mItem)
       (remprop obj (cadr mItem))
       (DelFuncData obj (cadr mItem)))))
  (Confirm-Info-Menu obj w))

(defun Dismiss-Var-Info-Menu (obj w x y)                     Dismiss-Var-Info-Menu
   (Yes-No-Menu "Remise à zéro des valeurs de la Variable" `(Do-Dismiss-Var-Info ',obj ',w)))

(defun Do-Dismiss-Var-Info (obj w)                           Do-Dismiss-Var-Info
  (mapc Var-Info-Menu
   (lambda (mItem)
     (when (listp mItem)
       (remprop obj (cadr mItem))
       (DelFuncData obj (cadr mItem)))))
  (Confirm-Info-Menu obj w))

            ;
========================================================================================= ; ;
            ; Deuxième partie : Présentation des résultats des analyses numériques sur les fonctions ; ;
            ;
========================================================================================= ; ;

                              ; ****************************** ; ;
                              ; Dessin du resultat des analyses ; ;
                              ; ****************************** ; ;

(defmacro StringLength (str) `(* (+ 1 (strlen ,str)) Zinfo-font-size-x))         StringLength

(defun Draw-Analyses-Results (obj w x y)                    Draw-Analyses-Results
  (if (and (boundp 'Draw-Analyses-Results)
     (iswidget Draw-Analyses-Results)) (xRemove Draw-Analyses-Results))
  (let (y-siz (+ 30 (* Zinfo-font-size-y (length (GetPrcData 'FDC (get obj 'func) 'liste-func)))))
   (print (get obj 'func) y-siz)
   (setq Draw-Analyses-Results
    (xCreateWidget '--AD-- W-Draw
              (xCreateWidget '--AV-- W-Viewport
                         (setq aux (xCreateWidget '--AS-- W-AShell
                                          "title" (strcat "Analyses numériques
sur " (get obj 'func))
                                          "iconName" "An. Num."))
                         "allowHoriz" "True" "allowVert" "True"
                         "width" 300 "height" (if (> y-siz 500 500 y-siz)))
              "font" Zinfo-font
              "width" 800 "height" 500)))
  (xRealizeWaiting aux)
  (let ((x-init 0) (x-max 0) (y-pos 0) (x-pos))
   (xSetValues Draw-Analyses-Results "foreground" 0)
                         ; initialisation
   (let (f-list (GetPrcData 'FDC (get obj 'func) 'liste-func))
    (cond
     ((null f-list)
  (xDrawLines Draw-Analyses-Results (setq x-pos (incr x-max 5)) 5 x-max (- y-pos 16))
  (incr y-pos (+ Zinfo-font-size-y 5))
  (xDrawLines Draw-Analyses-Results x-max (- y-pos 16) 300 (- y-pos 16))
  (mapc Composition-Variables
     (lambda (x)
          (let (str-x (strcat x))
           (xDrawLines Draw-Analyses-Results x-pos 5 x-pos (- y-pos 16))
```

```
                    (xDrawString Draw-Analyses-Results (+ x-pos 3) y-pos str-x)
                    (incr x-pos (StringLength str-x))))))
      (t
    (xDrawString Draw-Analyses-Results 0 (incr y-pos Zinfo-font-size-y) (car f-list))
    (if (> (setq aux (StringLength (strcat (car f-list)))) x-max)
       (setq x-max aux))
    (self (cdr f-list)))))
                                          ; dessin des valeurs
    (setq y-pos 0)
    (let ((f-list (GetPrcData 'FDC (get obj 'func) 'liste-func))
     (f-comp (GetPrcData 'COMP (car (GetPrcData 'FDC (get obj 'func) 'liste-func)) 'COMPOSITION)))
      (when f-list
    (setq x-pos (+ x-max 3))
    (incr y-pos Zinfo-font-size-y)
    (when f-comp
     (mapc Composition-Variables
              (lambda (x)
                (if (get f-comp x)
                   (xDrawString Draw-Analyses-Results x-pos y-pos (strcat (get f-comp x))))
                (incr x-pos (StringLength (strcat x))))))
    (self (cdr f-list) (GetPrcData 'COMP (cadr f-list) 'COMPOSITION))))))



          ; ========================================================= ; ;
          ; Troisième partie : Information sur les objets graphiques ; ;
          ; ========================================================= ; ;


    (defun GI-Put-Func-Info (drw gr-obj gl-num gl-object is-dep-func)                    GI-Put-Func-Info
       (xSetValues drw "foreground" "#300")
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Fonction " gr-obj (if is-dep-func (strcat " Liés a " is-
dep
    func) " ")))
       (xSetValues drw "foreground" "#000")
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Variables locales  " (getival gr-obj 0)))
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Variables globales " (getival gr-obj 1)))
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Fichier source     " (getival gr-obj 20)))
       (let (c-vals (GetPrcData 'COMP gr-obj 'COMPOSITION))
        (if c-vals
        (let (drw-str "Analyses:")
          (mapc Composition-Variables
                   (lambda (x)
                     (if (> (get c-vals x) 0) (setq drw-str (strcat drw-str " " x " " (get c-vals x))))))
          (mapc '(0 1 2 3 4 5 6 7)
                   (lambda (x)
                     (if (> (get c-vals x) 0) (setq drw-str (strcat drw-str " " x " " (get c-vals x))))))
          (xDrawString drw 4 (incr y Zinfo-font-size-y) drw-str))))
       (GI-Put-Drawing-Information gl-object gl-num))


    (defun GI-Put-Var-Info (drw gr-obj gl-num gl-object dep-func)                        GI-Put-Var-Info
       (xSetValues drw "foreground" "#300")
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Variable " gr-obj " de la fonction " dep-func))
       (xSetValues drw "foreground" "#000")
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Local dans    " (getival gr-obj 'localvar)))
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Global dans   " (getival gr-obj 'globalvar)))
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Modifiee dans " (getival gr-obj 'incr)))
       (GI-Put-Drawing-Information gl-object gl-num))

    (defun GI-Put-Instr-Info (drw gr-obj num-obj gl-object func)                         GI-Put-Instr-Info
       (xCreateWidget '--AL-- W-Label drw "label" (strcat (car gr-obj)) "x" 4 "y" y "font" Zinfo-font)
       (xDrawString drw 4 (incr y 40) (strcat "Instruction de la fonction: " func))
       (setq tmp gr-obj)
       (GI-Put-Drawing-Information gl-object num-obj))


    (defun GI-Put-Drawing-Information (gl-object gl-num)                                 GI-Put-Drawing-Information
       (while (car gl-object)
         (cond
          ((eq gl-num (GL-Get-Tag (car gl-object)))
       (xSetValues drw "foreground" "#404")
       (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Graphic Dessine depuis " (GL-Get-Object (car gl-
object)) " :")
```

```
    (xSetValues drw "foreground" "#000")
    (mapc (get (GetObjectByName (GL-Get-Object (car gl-object))) 'gl-data)
        (lambda (x)
            (xDrawString drw 15 (incr y Zinfo-font-size-y) (strcat x))))
    (Put-Transformation-Info (GL-Get-Transfo (car gl-object)))))
      (nextl gl-object)))


(defun Put-Transformation-Info (gl-transfo)                                    Put-Transformation-Info
    (xSetValues drw "foreground" "#303")
    (ifn (car (last (car gl-transfo))) (xDrawString drw 4 (incr y Zinfo-font-size-y) "Pas de Transformations")
    (xDrawString drw 4 (incr y Zinfo-font-size-y) "Transformations : ")
    (xSetValues drw "foreground" "#000")
    (let ((t-objs (reverse gl-transfo)) (s-drawn))
      (cond
       ((null t-objs) nil)
       (t
        (xSetValues drw "foreground" "#000")
        (xDrawString drw 4 (incr y Zinfo-font-size-y) (strcat "Objet: " (cadr t-objs) " Flot : "
                                                              (caddr t-objs)))
        (xSetValues drw "foreground" "#022")
        (mapc (car t-objs)
              (lambda (x)
               (cond
                ((not (listp x)) t)
                ((numbp (car x))
                 (if s-drawn (xDrawString drw 4 (incr y Zinfo-font-size-y) s-drawn))
                 (setq s-drawn (strcat "Position (" (Mdiv (1 x) scale-factor) " "
                                                    (Mdiv (2 x) scale-factor) " " (Mdiv (3 x) scale-factor) ") )))
                ((setq aux (assq (car x) '((t . "Translation ") (r . "Rotation    ")
                                                                (s . "Scaling     ") (c . "Couleur     "))))
                 (setq s-drawn (strcat s-drawn (cdr aux) (cdr x) " ")))))))

      (self (cdddr t-objs)))))))

(defun GL-Info (win obj x y z)                                                 GL-Info
  (setq  current-name-stack (if (eq win DD-Root-Window) dump-name-stack GA-Name-Stack))
  (while obj
    (GL-Make-Info (car obj))
    (nextl obj)))


(defmacro GetFuncFromInstr (instr) `(cadar (last (car (get-car-value ,instr 'entree)))))    GetFuncFromInstr

(defun SeekFuncFromStack (num)                                                 SeekFuncFromStack
  (let (stk (cdr (member num current-name-stack)))
    (and stk (or (and (listp (car stk)) (GetFuncFromInstr (car stk)))
                 (self (cddr stk))))))

(defun GL-Make-Info (obj)                                                      GL-Make-Info
  (let ((gr-obj (cadr (GetInNameStack obj current-name-stack))) (y 5) (gl-obj))
    (when (and gr-obj
          (null (or (and (listp gr-obj) (get-car-value gr-obj 'gr-info))
                    (and (atom  gr-obj) (car (getival gr-obj 'gr-info)))))
          (setq gl-obj (GL-Get-All-GL gr-obj obj)))
                                                    ; si non on la cree
      (let (title (if (listp gr-obj) (strcat (car gr-obj)) gr-obj))
        (let (root (xCreateWidget '--AS-- W-AShell "title" title "iconName" title))
          (setq drw (xCreateWidget '--DW-- "Drawxbvl" root "width" 530 "height" 700
                                           "background" W-BColor "font" Zinfo-font))
          (if (atom gr-obj) (addival gr-obj `(gr-info ,drw))
                (replace-car gr-obj 'gr-info drw))
          (xRealize root)))
      (cond
       ((and (atom gr-obj)                  ; l'objet designe une fonction ou une variable
             (> (ftyp gr-obj) 5))
        (Methode-Menu gr-obj drw 4 y Gr-Info-Menu nil "foreground" W-ComFg "background" W-ComBg)
        (incr y (* Zinfo-font-size-y 2))

        (cond
         ((null (is-var-dep (car gl-obj)))        (GI-Put-Func-Info drw gr-obj obj gl-obj))
         ((eq (GL-Get-TDEP (car gl-obj)) 'fdd-FUNC) (GI-Put-Func-Info drw gr-obj obj gl-obj (GL-Get-DDEP
(car gl-obj))))
         (t                                 (GI-Put-Var-Info drw gr-obj obj  gl-obj (GL-Get-DDEP (car gl-obj)))))
        (t                                  ; l'objet designe une instruction
         (Methode-Menu drw drw 4 y Gr-Info-Menu nil "foreground" W-ComFg "background" W-ComBg)
```

```
      (incr y (* Zinfo-font-size-y 2))
      (put drw 'instr gr-obj)
      (GI-Put-Instr-Info drw gr-obj obj gl-obj (or (and (listp gr-obj) (GetFuncFromInstr gr-obj))
                                                   (SeekFuncFromStack obj))))))))
```

Annexe I.12 Editeur de représentation : Zmeta-editor.vlisp

```
; Ce fichier defini un editeur de graphiques generes suivant la structure de construction.


; Constructeur d'automate :
;
; Par proposition de schema de base :
;
;   - Parcour de Flot :
;  - attachement a l'initialisation
;  - attachement au noeud de parcour
;  - attachement conditionel
;
(de ME-Init-Window ()                                                    ME-Init-Window
  (when GA-Tree
    (ifn (boundp 'Builder-GL) (Make-GL-Builder))
    (ifn (boundp 'ME-Windows)
        (let (root (xCreateWidget '--AS-- W-AShell "title" "Inspection de structures" "iconName" "Structures"))
          (let ((form (xCreateWidget '--AF-- W-Form root "width" 410 "height" 410 "background" W-BColor))
                (tmp)(tmp1))
               (setq ME-Windows (gensym))
               (put ME-Windows 'match 'partiel)
;              (setq tmp (xCreateCom form "Select" nil nil   "(ME-New-Pattern)" Meta-Font))
;              (setq tmp (xCreateCom form "Load" tmp nil     "(ME-Load-Pattern)" Meta-Font))
;              (setq tmp (xCreateCom form "Save" tmp nil     "(ME-Save-Pattern)" Meta-Font))
;              (setq tmp (xCreateCom form "Settings" tmp nil "(ME-Pattern-Settings)" Meta-Font))
;              (setq tmp (xCreateCom form "Find" tmp nil     "(ME-Match)" Meta-Font))
;              (setq tmp (xCreateCom form "Compute" tmp nil  "(ME-Compute-Match)" Meta-Font))
;              (setq tmp (xCreateCom form "Undo" tmp nil     "(ME-Compute-Match t)" Meta-Font))
;              (setq tmp1 (setq tmp (xCreateCom form "Fin" tmp nil "(ME-Quit)" Meta-Font)))
;              (xCreateCom form "Help" tmp nil "(ME-Help)" Meta-Font)
;              (setq tmp (xCreateWidget '--AV-- W-Viewport form "width" 300 "heigth" 300
;                                       "allowHoriz" "True" "allowVert" "True"
;                                       "vertDistance" (xGetVert tmp)
;                                       "left" "ChainLeft" "top" 0
;                                       "right" 2))
;              (put ME-Windows 'creat (xCreateWidget '--AD-- "brTree" tmp "width" 2000 "height" 550))
               (setq tmp (xCreateWidget '--AV-- W-Viewport form "width" 400 "heigth" 400
                                        "allowHoriz" "True" "allowVert" "True"
                                        "horizDistance" (xGetHoriz tmp)
                                        "vertDistance" (xGetVert tmp1)
                                        "left" "ChainLeft" "top" 0))
               (put ME-Windows 'tree (xCreateWidget '--AT-- "brTree" tmp "width" 2000 "height" 550
                                                    "hSpace" 3 "vSpace" 3))
               (xRealize root)
               (ME-Graphique)
;              (ME-Find-Tree-Pattern (cadr (xGetValues (get ME-Windows 'tree) "children")))
               (MPV-Reset-Label))))))



                                        ; ********************************* ; ;
                                        ; Dessin de la structure resultante ; ;
                                        ; ********************************* ; ;


(de ME-Graphique (force)                                                 ME-Graphique
  (when GA-Tree
      (put ME-Windows 'ga-tree GA-Tree)
      (ifn force
      (progn
        (remprop ME-Windows 'me-tree)
        (remprop ME-Windows 'find)
        (remprop ME-Windows 'compute)))
```

```
            (xUnmap (xGetValues (get ME-Windows 'tree) "parent"))
            (mapc (cdr (xGetValues (get ME-Windows 'tree) "children"))
             (lambda (x)
               (xRemoveWidget x)
               (ifn force (rplacd x nil))))
            (setq -list-transformation- nil)
            (let (g-t GA-Tree)
           (when g-t
              (ME-Tree (car g-t) (cadr (getcar g-t)))))
            (xMap (xGetValues (get ME-Windows 'tree) "parent"))))

        ; lancement de la construction de l'arbre
        (de ME-Tree (tree r-obj)                                                    ME-Tree
          (if tree
             (let (t-root (xCreateWidget 'node W-Command (get ME-Windows 'tree) "label" r-obj "background"
"#05f"))
           (xAddCallback t-root "callback" "(MPV-Message \"Elément racine de l'arbre\")")
           (setq t-histo nil)
           (ME-Construit-Tree tree t-root nil))))

        ; construction d'un nouveau widget dans l'arbre
        (de ME-Construit-Wdg (elem transfo pa)                                      ME-Construit-Wdg
        ;  (print elem transfo pa)
          (let (w-o (xCreateWidget '--FO-- "awCommand" (get ME-Windows 'tree)
                                   "treeParent" pa
                                   "bitmap" (cond ((and transfo elem) Node-Obj-Bitmap)
                                                  (transfo Node-Bitmap)
                                                  (t Obj-Bitmap))))
            (xAddCallback w-o "callback"
                          (cond ((and transfo elem) (strcat "(progn (MPV-Message \"" (cadr transfo)
                                                            "\") (ME-GR-Info " (car elem) "))"))
                                (transfo (strcat "(MPV-Message \"" (cadr transfo) "\")"))
                                (t (strcat "(ME-GR-Info " (car elem) ")"))))
            (xAugment w-o W-Btn2 (strcat "(ME-Select '" w-o ")"))
            (xAugment w-o W-Btn3 (strcat "(ME-Compute '" w-o ")"))
            w-o))

        ; s-wdg = supérieur dans l'arbre
        ; p-wdg = supérieur du supérieur
        (de ME-Construit-Tree (tree s-wdg)                                          ME-Construit-Tree
          (when tree
          (ifn (member (car tree) t-histo) (newl t-histo (car tree))
             (print "Doublon" tree)
             (break a))
          (cond
           ((null tree) t)
           ((listp (car tree))
            (ME-Construit-Tree (car tree) s-wdg)
            (ME-Construit-Tree (cdr tree) s-wdg))
           ((and (listp (caddr tree))
            (listp (car (caddr tree)))
            (getcar (cddr tree)))
            (ME-Construit-Tree (caddr tree)
                               (ME-Construit-Wdg nil (getcar (cddr tree))
                                                 (ME-Construit-Wdg tree (getcar tree) s-wdg)))
            (ME-Construit-Tree (cdr (cddr tree)) s-wdg))
           ((listp (caddr tree))
            (let (pa (ME-Construit-Wdg tree (getcar tree) s-wdg))
              (ME-Construit-Tree (caddr tree) pa)
              (let (n-tree (cdr (cddr tree)))
            (cond
             ((null n-tree) nil)
             ((atom (car n-tree)) (ME-Construit-Tree n-tree s-wdg))
             (t
              (ME-Construit-Tree (car n-tree) pa)
              (self (cdr n-tree)))))))
           (t
            (ME-Construit-Wdg tree (getcar tree) s-wdg)
            (ME-Construit-Tree (cddr tree) s-wdg)))))

        (de ME-C-Branche (tree s-wdg p-wdg)                                         ME-C-Branche
          (cond
           ((null tree) t)
           ((atom (car tree))
            (print "New P Elem" (car tree) (cadr tree))
            (ME-Construit-Tree tree p-wdg nil))
```

```
   ((and (listp (caar tree))
    (getcar tree))
    (print "Null Elem" (getcar tree) (car tree))
    (ME-Construit-Tree (caar tree) (ME-Construit-Wdg nil (getcar tree) s-wdg) s-wdg))
   (t
    (print "Branche+1")
    (ME-Construit-Tree (car tree)
    (ME-C-Branche (cdr tree))))))


(de Display-Transformation (l) (MPV-Message (strcat (car l))))                    Display-Transformation

(de ME-New-Node (ele -parent- -superieur- objet have-w)                           ME-New-Node
  (let (w (cond
     ((stringp ele)
      (setq aux (if have-w have-w
                        (xCreateWidget "node" "awCommand" -parent-
                                       "treeParent" -superieur- )))
      (cond
       (-list-transformation-
        (put aux 'node -list-transformation-)
        (xSetValues aux "bitmap" Node-Bitmap)
        (xAddCallback aux "callback" (strcat "(MPV-Message (strcat (car '" -list-transformation- ")))")))
       (t
        (xSetValues aux "label" "R")
        (xAddCallback aux "callback" "(MPV-Message \"Element racine de l'arbre\")")))
      aux)
     (t (let (n-w (if have-w have-w
                        (xCreateWidget '--Tree-Node-- "awCommand" -parent-

                                       "treeParent" -superieur- )))
          (xSetValues n-w "font" "Fixed" "bitmap" Obj-Bitmap)
          (put n-w 'tag ele)
          (xAddCallback n-w "callback" (strcat "(ME-GR-Info '" ele ")"))
          (if (and force (get n-w 'couleur))
              (progn
                (xSetValues n-w "background" (get n-w 'couleur))
                (if -superieur- (xSetValues -superieur- "background" (get n-w 'couleur)))))
          n-w))))
   (xAugment w W-Btn2 (strcat "(ME-Select '" w ")"))
   (xAugment w W-Btn3 (strcat "(ME-Compute '" w ")"))))


                                                ;
*****************************************************************************************; ;
                                                ; Edition du code source a partir du graphique..... (Bouton
Mouse 5 : centre + droit) ; ;
                                                ;
*****************************************************************************************; ;

; Objet remplacent les compaction...

(de ME-Draw-Empty ()                                                              ME-Draw-Empty
  (if (boundp 'empty-list) empty-list
    (setq empty-list (GLnewlist))
    (GLpushAttrib "GL_CURRENT_BIT")
    (GLcolor 3 255 255 255)
    (GLcubeSolid 5)
    (GLcolor 3 0 255 255)
    (GLcubeWire 5)
    (GLpopAttrib)
    (GLendlist)))

; ME-Compact : compact a partir d'un objet selectionne

(de ME-Compact (win obj gl-data)                                                  ME-Compact
  (ifn (boundp 'empty-list) (ME-Draw-Empty))
  (let (code (cadr (member (car obj) GA-Name-Stack)))
    (if (listp code)                            ; ne fonctionne que pour la compression d'expressions
    (let ((tag (GL-Get-List (setq gl-data (GL-Get-GL code (car obj))))) (lisp-code) (f-list) (l-list))
      (setq f-list (caar (last (GL-Get-Transfo-Values gl-data)))
            l-list (cadar (last (GL-Get-Transfo-Values gl-data))))
      (ifn (or f-list l-list) (setq f-list tag l-list tag))
      (cond
       ((null code) nil)
       ((equal (car code) "**EMPTY*BLOCK**")
```

```
            (GLnewlist tag)
            (GLcalllist (+ tag curent-to-object))
            (GLendlist)
            (rplaca code (get-car-value code 'original))
            (setq lisp-code (car code))
            (ME-Build-Expression (car code) t t (car obj)))
           (t
            (GLnewlist tag)
            (GLloadname (car obj))
            (GLcalllist empty-list)
            (GLendlist)
            (ME-Build-Expression (car code) nil t (car obj))
            (ifn (get-car-value code 'original)
                 (add-in-car code 'original (car code)))
            (setq lisp-code (car code))
            (rplaca code "**EMPTY*BLOCK**")))
          (ME-Add-In-Builder (cons (car obj) (cons f-list (cons l-list (cons lisp-code)))))))))

; ME-Build-Expression : parcour de l'expression,
; selon le mode (is-empty) : effacement ou restauration du graphique original
; construction du pattern
; construction du graphique correspondant dans la fenetre ME-Meta

(de ME-Build-Expression (expr is-empty is-first list)                    ME-Build-Expression
  (if (listp expr)
      (progn
    (let (gl-list (GL-Get-List (GL-Get-GL expr list)))
      (let (t-list (car (last (GL-Get-Transfo-Values (GL-Get-GL expr list)))))
        (if (and (car t-list) (< (car t-list) f-list)) (setq f-list (car t-list)))
        (if (> (cadr t-list) l-list) (setq l-list (cadr t-list))))
      (if gl-list
          (progn
                (if (< gl-list f-list) (setq f-list gl-list))
                (if (> gl-list l-list) (setq l-list gl-list))
                ))
      (cond
       ((null gl-list) t)
       (is-first t)
       (is-empty
        (GLnewlist gl-list)
        (GLcalllist (+ gl-list curent-to-object))
        (GLendlist))
       (t
        (GLnewlist gl-list)
        (GLendlist))))
    (if (listp expr) (progn (ME-Build-Expression (car expr) is-empty) (ME-Build-Expression (cdr expr) is-
empty))))))


; ME-Is-In-Builder :
; verification de la non-existance de l'objet dans Builder
; ou dans une partie d'un objet deja dans builder
; object = (n-build [n-first-obj n-last-obj])

(de ME-Is-In-Builder (object)                                            ME-Is-In-Builder
   (if (and (not (member (car object) GA-Build-Stack))
       (or (null object)
                (not (ME-Is-Included (cdr object)))))) nil
     t))

(de ME-Is-Included (obj tmp)                                             ME-Is-Included
  (let (tmp GA-Build-Stack)
    (cond
     ((null tmp) nil)
     ((and (ge (car obj) (cadr tmp))
      (le (cadr obj) (caddr tmp))) t)
     (t
      (self (nth 4 tmp))))))

; Ajout d'un objet dans la fenetre du Builder

(de ME-Add-In-Builder (new-object tmp)                                   ME-Add-In-Builder
  (if (ME-Is-In-Builder new-object) nil
    ; initialisation
    (GLwinset Builder-GL)
    (rplacd (last new-object) GA-Build-Stack)
```

```
   (setq GA-Build-Stack new-object)
   (ifn (boundp 'ME-Curent-Position) (setq ME-Curent-Position 0))
   (setq tmp (cadr new-object))
   ; construction de la transformation
   (GLnewlist (1+ ME-Curent-Last-List))
   (GLtranslate ME-Curent-Position 0 0)
   (GLendlist)
   ; redessin dans la fenetre du builder
   (GLnewlist ME-Curent-Last-List)
   (GLloadname ME-Curent-Last-List)
   (GLpushmatrix)
   (GLcalllist (1+ ME-Curent-Last-List))
   ; dessin du cube de l'objet
   (GLcolor 3 255 255 255)
   (GLcubeWire 10)
   ; rappel des listes
   (while (< tmp (1+ (caddr new-object)))
     (GLcalllist (+ tmp curent-to-object))     ; appel du reel objet graphique... (celui copie dans Builder)
     (incr tmp))
   (GLpopmatrix)
   (GLpopmatrix)
   (GLendlist)
   ; maj des variables...
   (incr ME-Curent-Last-List 2)
   (incr ME-Curent-Position 20)
   (setq tmp ME-First-List)
   ; reconstruction du root-list
   (GLnewlist ME-Root-List)
   (while (< tmp (1+ ME-Curent-Last-List))
     (GLcalllist tmp)
     (incr tmp))
   (GLendlist)
   (GLrootlist ME-Root-List)))


(de ME-Quit ()                                                                    ME-Quit
   (ifn (boundp 'ME-Windows) nil
   (xRemove (get ME-Windows 'creat))
   (setq ME-Windows 'undef)))

(de ME-Help ()                                                                    ME-Help
   (terpri)
   (print "Bouton 1 | Bouton 2    | Bouton 3")
   (print "Creation | Destruction | Type")
   t)

                                                ; ******************* ; ;
                                                ; Gestion des patterns ; ;
                                                ; ******************* ; ;

(defmacro GetPatternByName (nom) `(GetObjectByName ,nom ME-Pattern-List))      GetPatternByName

                                                ; ********************** ; ;
                                                ; lecture dans un fichier ; ;
                                                ; ********************** ; ;

; syntaxe :
; (define-Z-pattern nom
;     (ordre   <numero d'ordre>) : ordre de recherche
;     (match   <partiel|exact>)  : correspondance partielle ou exacte
;     (couleur <couleur>)        : couleur d'affichage
;     (pattern <pattern>)        : le pattern
;     (graphique <instructions-graphiques>)) : les instructions graphiques en remplacement
;
;
; <couleur> = (r g b) : 0 <= r <= 15, 0 <= g <= 15, 0 <= b <= 15
; <pattern> = (feuille ... feuille noeud ( sous-pattern ) feuille ... feuille)
; <sous-pattern> = <pattern>
; <feuille> = un atome
; <noeud>   = un atome ou un nom de pattern existant

(deff  define-Z-pattern (data)                                                    define-Z-pattern
  (ifn (assoc 'pattern data) (MPV-Message (strcat "Definission d'un pattern vide " (car data)))
       (let (obj (GetPatternByName (car data)))
    (ifn obj (setq ME-Pattern-List (cons (setq obj (gensym)) ME-Pattern-List)))
```

```
    (put obj 'name (nextl data))
    (mapc data (lambda (x) (put obj (car x) (cadr x))))
    (ME-Parse-New-Pattern (get obj 'pattern)))))


(de ME-Parse-New-Pattern (pattern)                                    ME-Parse-New-Pattern
  (if (atom pattern) nil
    (let (a-name (car pattern))
      (cond
       ((and a-name (atom a-name))
    (rplaca pattern (gensym))
    (put (car pattern) 'type (cond ((GetPatternByName a-name)
                                                (put (car pattern) 'pat-compo a-name)
                                                'compo)
                                           ((listp (cadr pattern)) 'node)
                                           (t 'feuille))))
       (a-name
    (ME-Parse-New-Pattern (car pattern))))
      (ME-Parse-New-Pattern (cdr pattern)))))



; ******** ; ;
; ecriture ; ;
; ******** ; ;

(de ME-Write-Pattern (f-name)                                         ME-Write-Pattern
  (output f-name)
  (mapc ME-Pattern-List
    (lambda (x)
      (print "; definission du pattern " (get x 'name))
      (print "(define-Z-pattern" (get x 'name))
      (princ "  (pattern (") (ME-Print-Pattern (get x 'pattern)) (print "))")
      (print "  (order " (get x 'ordre) ")")
      (print "  (couleur " (get x 'couleur) ")")
      (print "  (graphique " (get x 'graphique) "))")
      (terpri))))

(de ME-Print-Pattern (pattern)                                        ME-Print-Pattern
  (cond
    ((null pattern) nil)
    ((listp (cadr pattern))
     (princ "node (") (ME-Print-Pattern (cadr pattern)) (princ ")")
     (ME-Print-Pattern (cddr pattern)))
    (t
     (princ (or (get (car pattern) 'pat-compo) "feuille"))
     (ME-Print-Pattern (cdr pattern)))))


(defmacro print-cond (v f n) `(if (get ,v ,f) (print ,n (get ,v ,f))))   print-cond

(de ME-Save-Pattern ()                                                ME-Save-Pattern
    (xGetFileName () 'Write-Pattern-File))

(de Write-Pattern-File (f-name)                                       Write-Pattern-File
    (output f-name)
    (mapc ME-Pattern-List
      (lambda (x)
        (print "PATTERN " (get x 'name))
        (print-cond x 'pattern "DATA")
        (print-cond x 'graphique "GL")
        (print-cond x 'order "ORDER"))))

                                                ; ********************** ; ;
                                                ; Traitement des patterns ; ;
                                                ; ********************** ; ;


; selection dans les patterns existants

(de ME-Select-Pattern (action)                                        ME-Select-Pattern
    (if action (put 'ME-Select-Pat 'action action) (remprop 'ME-Select-Pat 'action))
    (let (lst (ME-Make-Pat-List action))
      (let (w-lst (xCreateList "Selectionnez un pattern" lst 2))
    (xAddCallback w-lst "callback" (strcat "(ME-Select-Pat $l '" lst " '" w-lst ")")))))
```

```
(de ME-Select-Pat (ind lst w-lst)                                    ME-Select-Pat
   (xRemove w-lst)
   (if (and (= ind 0) (get 'ME-Select-Pat 'action)) (eval (get 'ME-Select-Pat 'action))
    (ME-Display-Pattern (GetObjectByName ((if (get 'ME-Select-Pat 'action) (- ind 1) ind) lst) ME-Pattern-
List))))


(de ME-Make-Pat-List (is-create)                                     ME-Make-Pat-List
   (let ((o-lst ME-Pattern-List) (ret nil))
    (cond
     ((null o-lst) (if (null is-create) ret `(Nouveau ,@ret)))
     (t
   (self (cdr o-lst) (cons (get (car o-lst) 'name) ret))))))

; creation d'un nouveau

(de ME-New-Pattern ()                                                ME-New-Pattern
   (ME-Select-Pattern '(xCreateDialogBox "Nom du pattern" nil '(ME-New-Pat-Name xDialogValue)) t))

(de ME-New-Pat-Name (name)                                           ME-New-Pat-Name
   (let (new-pat (GetObjectByName name ME-Pattern-List))
    (if new-pat (ME-Display-Pattern new-pat)
   (setq ME-Pattern-List (cons (setq new-pat (MkNewElem (get ME-Windows 'creat))) ME-Pattern-List))
   (put new-pat 'name name)
   (put new-pat 'root new-pat)
   (put new-pat 'pattern [new-pat])
   (put new-pat 'match 'exact)
   (put new-pat 'display t)
   (ME-Set-Pattern-Color new-pat new-pat))))

; ************************************************ ; ;
; affichage / Moditication des donnees d'un pattern ; ;
; ************************************************ ; ;

(de ME-Pattern-Settings ()                                           ME-Pattern-Settings
   (if (get ME-Windows 'settings)
   (progn (xRemove (get ME-Windows 'settings))
        (remprop ME-Windows 'settings)
        (mapc ME-Pattern-List (lambda (x) (remprop x 'tree))))
    (if ME-Pattern-List
    (let (root (xCreateWidget '--AS-- W-AShell "title" "Pattern Setting" "iconName" "Set"))
     (put ME-Windows 'settings root)
     (let (vp (xCreateWidget '--AV-- W-Viewport root "width" 275 "height" 400
                                    "allowVert" "True"))
      (let (bx (xCreateWidget '--AB-- W-Box vp "width" 275 "height" (* 175 (length ME-Pattern-List))
                                    "background" "#ea1"))
           (mapc ME-Pattern-List
              (lambda (x) (ME-Make-One-Pattern-Window x bx)))
           (xRealize root)))))))


(de ME-Make-One-Pattern-Window (p-root r-wid)                        ME-Make-One-Pattern-Window
   (ifn (and p-root (get p-root 'pattern) (null (get p-root 'tree))) nil
   (let ((form (xCreateWidget '--AF-- W-Form r-wid "width" 250 "height" 175 "background" "#ae4" ))
        (tmp)
        (tmp1 4))
     (xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command form "label" (strcat (get p-root 'name))))
                   "callback" (strcat "(ME-Set-Pattern-Value 'Nom 'name '" tmp " '" p-root ")"))
     (xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command form
                                            "horizDistance" (setq tmp1 (+ (xGetValues tmp
"width") 9))
                                            "label" " "
                                            "background" (ME-Make-Color (get p-root 'couleur))))
                   "callback" (strcat "(ME-Set-Pattern-Color '" tmp " '" p-root ")"))
     (xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command form
                                            "horizDistance" (incr tmp1 (+ (xGetValues tmp "width")
5))
                                            "label" (strcat (get p-root 'match))))
                   "callback" (strcat "(ME-Set-Pattern-Type '" tmp " '" p-root ")"))
     (xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command form
                                            "horizDistance" (incr tmp1 (+ (xGetValues tmp "width")
5))
                                            "label" (strcat (get p-root 'ordre))))
                   "callback" (strcat "(ME-Set-Pattern-Value 'Ordre 'ordre '" tmp " '" p-root ")"))
     (xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command form
                                            "horizDistance" (incr tmp1 (+ (xGetValues tmp "width")
```

```
5))
                                                    "label" (strcat (get p-root 'graphique))))
                                "callback" (strcat "(ME-Set-Pattern-Value \"Fonction GL\" 'graphique '" tmp " '" p-root
")"))
        (setq tmp (xCreateWidget '--AV-- W-Viewport form "width" (incr tmp1 (xGetValues tmp "width")) "heigth"
150
                                    "allowHoriz" "True" "allowVert" "True"
                                    "vertDistance" 30
                                    "left" "ChainLeft" "top" 0
                                    "right" 2))
        (put p-root 'tree (xCreateWidget '--AD-- "brTree" tmp "width" tmp1 "height" 150))
        (ME-Display-Pattern-Tree p-root (get p-root 'tree))
        form)))

    ; affichage de l'arbre d'un pattern

    (de ME-Display-Pattern-Tree (pat t-root)                                    ME-Display-Pattern-Tree
      (let ((p (get pat 'pattern)) (t-par))
        (cond
         ((atom p) t)
         ((listp (car p)) t)
         ((listp (cadr p))
      (self (cadr p) (CreateNewWidget t-root t-par (get (car p) 'type)))
      (self (cddr p) t-par))
         (t
      (CreateNewWidget t-root t-par (get (car p) 'type))
      (self (cdr p) t-par)))))


    (de ME-Display-Pattern (pat root)                                           ME-Display-Pattern
      (ifn (get pat 'display)
      (let (t-root (if root root (get ME-Windows 'creat)))
        (put pat 'display t)
        (let ((n-pat (get pat 'pattern)) (t-par (MkNewElem t-root)))
          (cond
           ((null n-pat) t)
           ((atom n-pat) (MkNewElem t-root t-par n-pat))
           ((listp (car n-pat))
            (self (cdr n-pat) t-par)
            (setq t-par (MkNewElem t-root t-par))
            (self (car n-pat) t-par))
           (t
            (self (car n-pat) t-par) (self (cdr n-pat) t-par)))))))

    (de ME-Set-Pattern-Value (what field wid pat)                               ME-Set-Pattern-Value
      (xCreateDialogBox (strcat what " : " (get pat field)) (get pat field) `(ME-Set-Pat-Value ',wid ',pat ',field xDial
    gValue)))

    (de ME-Set-Pat-Value (wid pat field value)                                  ME-Set-Pat-Value
      (put pat field (implode (explode value)))
      (xSetValues wid "label" value))


    (de ME-Set-Pattern-Type (wid pat)                                           ME-Set-Pattern-Type
      (remprop pat 'filter)
      (Do-Selection-Menu pat 'match wid "Type de Match" Match-Type nil "foreground" W-ComFg "back-
ground" W-ComBg))

    ; Traitement de la couleur ; ;
    (de dec-hex (n p)                                                           dec-hex
      (cond
       ((< n 10) n)
       (p 'a)
       (t ((- n 9) '(a b c d e f)))))

    ; conversion num -> str de la couleur
    (de ME-Make-Color (n-col is-do)                                             ME-Make-Color
      (ifn n-col "#fff"
      (strcat "#" (dec-hex (1 n-col) is-do) (dec-hex (2 n-col) is-do) (dec-hex (3 n-col) is-do))))

    ; changement de la couleur de base
    (de ME-Set-Pattern-Color (wid pat)                                          ME-Set-Pattern-Color
      (let (root (xCreateWidget '--AS-- W-AShell "title" "Couleurs"))
        (let ((box (xCreateWidget '--AB-- W-Box root "width" 150 "height" 32)) (tmp 0))
      (repeat N-Base-Colors
                (xAddCallback (xCreateWidget '--AC-- W-Command box "label" " "
```

```
                                        "background" (ME-Make-Color ((incr tmp) Base-Colors)))
                                "callback" (strcat "(ME-Set-Pat-Color '" root " '" wid " '" pat " " tmp ")"))))
    (xRealize root)))

(de ME-Set-Pat-Color (r-list wid pat col)                                    ME-Set-Pat-Color
   (put pat 'couleur (col Base-Colors))
   (if wid (xSetValues wid "background" (ME-Make-Color (col Base-Colors))))
   (xRemove r-list))


                                        ; ********************************************** ; ;
                                        ; recherche de patterns dans les arbres graphiques ; ;
                                        ; ********************************************** ; ;

(de ME-get-childrens (t-w)                                                   ME-get-childrens
  (xGetValues t-w "treeChildren"))

(de ME-Find-Tree-Pattern (root min-depth get-childrens)                      ME-Find-Tree-Pattern
  (ifn get-childrens (setq get-childrens 'ME-get-childrens))
  (setq c-list nil max-children nil
   pattern-list nil s-patterns nil)
  (setq max-depth (car (ME-Build-Pattern-List (car (xGetValues (get ME-Windows 'tree) "children")))))
  (ME-Build-Patterns c-list max-depth max-children))

(de ME-Build-Pattern-List (elem depth childrens)                             ME-Build-Pattern-List
  (ifn (setq childrens (get-childrens elem)) '(0 . 0)
      (let ((l-c (length childrens))
       (ret nil))
    (while childrens
      (setq aux (ME-Build-Pattern-List (nextl childrens)))
      (if (> (1+ (car aux)) depth) (setq depth (1+ (car aux))))
      (newl c-list aux))
    (if (> l-c max-children) (setq max-children l-c))
    (setq ret (cons depth l-c))
    (if (and (ge depth min-dept)
                (not (member ret s-patterns))) (newl s-patterns ret))
    ret)))

(de ME-Build-And-Find-Patterns (elem depth childrens)                        ME-Build-And-Find-Patterns
  (ifn (setq childrens (get-childrens elem)) 0
      (while childrens
    (setq aux (ME-Build-Pattern-List (setq aux1 (nextl childrens))))
    (if (> (1+ aux) depth)
       (ME-Check-Pattern (setq depth (1+ aux)) aux1)))
      depth))

(de ME-Check-Pattern (depth root)                                            ME-Check-Pattern
  (let ((found nil)(e-patterns (cadr (member depth pattern-list))))
    (while (and e-patterns (null found))
      (ifn e-patterns (progn (newl pattern-list [root]) (newl pattern-list depth))))))



(de ME-Build-Patterns (lst mdepth mchild)                                    ME-Build-Patterns
  (while s-patterns
    (let ((couple (car s-patterns))
     (found-couple (member (car s-patterns) c-list))
     (others))
      (ifn found-couple (nextl s-patterns)
      (let (autres-couple (member couple (cdr found-couple)))
        (ifn autres-couple
                (when (> n 1) (newl pattern-list found-couple))
                                        ; recherche de correspondance entre les couples
                (if (ME-Similar-Pattern found-couple autres-couple) (incr n))
                (self (member couple (cdr autres-couple)))))
      (setq others found-couple)
      (while (and others
                  (setq others (member couple (cdr others)))
                  (ME-Was-Found others)))
      (if others (self couple others nil))))
    (nextl s-patterns)))

(de ME-Was-Found (pattern)                                                   ME-Was-Found
  (let (f-pat pattern-list)
    (cond
     ((or (null f-pat)
```

```
    (not (and (= (car pattern) (caar f-pat))
              (= (cdr pattern) (cdar f-pat))))) nil)
    ((ME-Similar-Pattern pattern (car f-pat)) t)
    (t (self (cdr f-pat))))))

(de ME-Similar-Pattern (pattern1 pattern2)                    ME-Similar-Pattern
  (ME-Compare-Pattern))

(de ME-Compare-Pattern ()                                     ME-Compare-Pattern
  (ifn (and (= (caar pattern1) (caar pattern2))
       (= (cdar pattern1) (cdar pattern2))) nil
       (let ((n-p (cdar pattern1))
        (is-similar t))
    (cond
     ((null is-similar) nil)
     ((< n-p 1) t)
     (t
      (nextl pattern1)
      (nextl pattern2)
      (setq is-similar (ME-Compare-Pattern))
      (self (1- n-p) is-similar))))))

(de ME-Rebuild-Pattern (pattern)                              ME-Rebuild-Pattern
  (ME-Do-Rebuild-Pattern))

(de ME-Do-Rebuild-Pattern ()                                  ME-Do-Rebuild-Pattern
  (if (= (cdar pattern) 0) 'feuille
    (let (new-pat nil)
      (repeat (cdar pattern)
        (nextl pattern)
        (newl new-pat (ME-Do-Rebuild-Pattern)))
      new-pat)))



                                        ; ****************************************** ; ;
                                        ; Actions a partir de la structure resultante ; ;
                                        ; ****************************************** ; ;

; informations ; ;
(de ME-GR-Info (tag)                                          ME-GR-Info
    (GL-Info Meta-GL [tag])
    (car (cadr (GetInNameStack tag GA-Name-Stack))))

; selection / deselection d'un sous arbre ; ;
(de ME-Select (w)                                             ME-Select
  (ME-Do-Select w (get w 'selected)))

(de ME-Do-Select (w is-do)                                    ME-Do-Select
  (ifn w t
       (xSetValues w "background" (if is-do "#fff" "#000") "foreground" (if is-do "#000" "#fff"))
       (put w 'selected (not is-do))
       (mapc (xGetValues w "treeChildren")
        (lambda (x) (ME-Do-Select x is-do)))))

; compaction / decompaction d'un sous arbre ; ;
(de ME-Compute (w)                                            ME-Compute
  (xUnmap (xGetValues (get ME-Windows 'tree) "parent"))
  (ifn (get w 'selected) (ME-Do-Select w))
  (if (get w 'reduced)   (ME-Undo-Compute w)
    (setq ME-Computed-Lists nil)
    ; en premier lieu on compacte les descendants
    (put w 't-child (xGetValues w "treeChildren"))
    (mapc (get w 't-child) 'ME-Do-Compute)
    ; puis on compacte l'objet graphique sélectionné
    (when (get w 'tag) (GLnewlist (Find-Object-List-From-Tag (get w 'tag))) (GLendlist))
    (xSetValues w "background" "#f99"))
  (put w 'reduced (not (get w 'reduced)))
  (xMap (xGetValues (get ME-Windows 'tree) "parent")))

(de ME-Do-Compute (w)                                         ME-Do-Compute
  (put w 't-child (xGetValues w "treeChildren"))
  (print w (get w 't-child))
  (mapc (get w 't-child) 'ME-Do-Compute)
  (when (get w 'tag)
   (newl ME-Computed-Lists (Find-Object-List-From-Tag (get w 'tag)))
```

```
  (GLnewlist (Find-Object-List-From-Tag (get w 'tag))) (GLendlist))
 (xRemoveWidget w))

(de ME-Undo-Compute (w -list-transformation- force)                    ME-Undo-Compute
 (when (get w 'tag)
  (GLnewlist (setq tmp (Find-Object-List-From-Tag (get w 'tag))))
  (GLcalllist (+ tmp curent-to-object))
  (GLendlist))
 (mapc (get w 't-child)
  (lambda (aW)
   (setq -list-transformation- (if (get aW 'node) (strcat (get aW 'node))))
   (ME-New-Node (if (get aW 'tag) (get aW 'tag) " ") nil w nil
                (xCreateWidget aW "awCommand" (get ME-Windows 'tree) "treeParent" w
                               "background" "#fff" "foreground" "#000"))
   (ME-Undo-Compute aW)))
 (remprop w 't-child))

                                        ; ********************************* ; ;
                                        ; creation d'une structure a la main ; ;
                                        ; ********************************* ; ;

(de ME-Add-In-Pattern (pat-root new parent)                            ME-Add-In-Pattern
   (ifn (and (get pat-root 'pattern) parent) (if pat-root (put pat-root 'pattern [new]))
   (let (o-pattern (get pat-root 'pattern))
    (cond
     ((atom o-pattern) t)
     ((listp (car o-pattern)) (self (car o-pattern)) (self (cdr o-pattern)))
     ((eq (car o-pattern) parent)
      (if (listp (cadr o-pattern)) (rplacd (last (cadr o-pattern)) (cons new))
        (rplacd o-pattern (cons (cons new) (cdr o-pattern)))))
     (t
      (self (cdr o-pattern)))))))

(de ME-Sub-In-Pattern (pat-root elem)                                  ME-Sub-In-Pattern
   (ifn (get pat-root 'pattern) nil
   (remprop pat-root 'filter)
   (let ((o-pattern (get pat-root 'pattern)) (aux))
    (cond
     ((atom o-pattern) t)
     ((and (eq (cadr o-pattern) elem) (null (cddr o-pattern))) (rplacd o-pattern nil))
     ((eq (car o-pattern) elem)
      (if (atom (cadr o-pattern)) (rplac o-pattern (cadr o-pattern) (cddr o-pattern))
        (rplacd (last (cadr o-pattern)) (cddr o-pattern))
        (rplac  o-pattern (car (cadr o-pattern)) (cdr (cadr o-pattern)))))
     (t (self (car o-pattern))
        (self (cdr o-pattern)))))))

(defmacro CreateNewWidget (root t-par type nom)                        CreateNewWidget
 `(xCreateWidget (ifn ,nom 'pattern ,nom) W-Command ,root
                 "bitmap" (cond ((eq ,type 'node) Node-Bitmap)
                                ((eq ,type 'compo) Tree-Bitmap)
                                (t Obj-Bitmap))
                 "treeParent" ,t-par))

(de MkNewElem (par t-par type)                                         MkNewElem
   (let ((n-f (CreateNewWidget par t-par type))
    (o-pat (ifn t-par t-par (get t-par 'root))))
    (when t-par
     (put t-par 'type 'node)
     (xSetValues t-par "bitmap" Node-Bitmap))
    (put n-f 'root o-pat)
    (put n-f 'type 'feuille)
    (remprop o-pat 'filter)
    (ME-Add-In-Pattern o-pat n-f t-par)
    (xSetValues n-f "background" (ME-Make-Color (get o-pat 'couleur)))
    (xAddCallback n-f W-ClBck (strcat "(ME-Cre-Node '" n-f ")"))
    (xAugment n-f W-Btn2   (strcat "(ME-Kill-Node '" n-f ")"))
    (xAugment n-f W-Btn3   (strcat "(ME-Node-Type '" n-f ")"))
    n-f))

(de ME-Node-Type (node)                                                ME-Node-Type
 (print node)
 (remprop (get node 'root) node 'filter)
 (cond
  ((eq (get node 'type) 'feuille)
   (put node 'type 'compo)
```

```
   (xSetValues node "bitmap" Tree-Bitmap))
  ((get node 'pat-compo)
   (xSetValues node "background" (ME-Make-Color (get (get node 'root) 'couleur)))
   (xSetValues node "bitmap" Obj-Bitmap)
   (remprop node 'pat-compo)
   (put node 'type 'feuille))
  (t
   (let (lst (ME-Make-Pat-List))
     (let (w-lst (xCreateList "Selectionnez un pattern" lst 2))
   (xAddCallback w-lst "callback" (strcat "(ME-Set-Pat-Compo "' node " $l "' lst " "' w-lst ")")))))))

(de ME-Set-Pat-Compo (pat ind lst w-lst)                                    ME-Set-Pat-Compo
   (xRemove w-lst)
   (put pat 'pat-compo (GetObjectByName (car (nth (1+ ind) lst)) ME-Pattern-List))
   (xSetValues pat "background" (ME-Make-Color (get (get pat 'pat-compo) 'couleur))))


(de ME-Cre-Node (node)                                                      ME-Cre-Node
   (remprop ME-Windows 'find)
   (xSetValues (xGetValues node "treeParent") "bitmap" Node-Bitmap)
   (MkNewElem (get ME-Windows 'creat) node))

(de ME-Kill-Node (node)                                                     ME-Kill-Node
   (if (eq (xGetValues node "treeParent") 'root) "Impossible de detruir la racine"
     (remprop ME-Windows 'find)
     (let (t-parent (xGetValues node "treeParent"))
   (let ((n-child (xGetValues node "treeChildren")) (n-brother (xGetValues node "treeBrother")))
     (cond
      ((null n-child) t)
      (t
       (xSetValues (car n-child) "treeParent" t-parent)
       (xSetValues (car n-child) "treeBrother" n-brother)
       (self (cdr n-child) (car n-child)))))
   (ifn (xGetValues t-parent "treeChildren") (xSetValues t-parent "bitmap" Obj-Bitmap)))
     (ME-Sub-In-Pattern (get node 'root) node)
     (xRemoveWidget node)))


                                        ; ********************************************** ; ;
                                        ; Creation automatique d'une nouvelle structure ; ;
                                        ; (a partire de la recherche)                   ; ;
                                        ; ********************************************** ; ;

(de ME-Add-Found-Pattern (occurences depth pattern)                         ME-Add-Found-Pattern
  (let (nom (implode (append '(p a t t e r n -)
                             (append (explode depth)
                                     (append '(-)
                                             (append (explode occurences) '(-)))))))
    (print nom)
    (let (new-pat (CreateNewWidget (get ME-Windows 'creat) nil nil nom))
      (newl ME-Pattern-List new-pat)
      (put new-pat 'name new-pat)
      (put new-pat 'root new-pat)
      (put new-pat 'pattern [new-pat])
      (put new-pat 'match 'exact)
      (put new-pat 'display t)
      (put new-pat 'couleur (cons 15 (cons (rem occurences 16) (cons (/ occurences 16) nil))))
      (xSetValues new-pat "background" (ME-Make-Color (get new-pat 'couleur)))
      (put new-pat 'ordre 1)
      (ME-Fill-New-Found-Pattern pattern new-pat))))

(de ME-Fill-New-Found-Pattern (pattern t-par)                               ME-Fill-New-Found-Pattern
  (if pattern
      (let ((p-root (get t-par 'root))
        (new-p  (MkNewElem (get ME-Windows 'creat) t-par)))
    (cond
     ((listp (car pattern))
      (ME-Fill-New-Found-Pattern (car pattern) new-p))
     (t
      (ME-Fill-New-Found-Pattern (cdr pattern) t-par))))))


                                        ; ************************************************** ; ;
                                        ; Correspondance entre une structure et le graphique ; ;
                                        ; ************************************************** ; ;
```

```
(de ME-Make-Patterns ()                                                    ME-Make-Patterns
   (let ((ordre 1) (ret) (lst) (tmp))
     (setq lst ME-Pattern-List tmp nil)
     (while lst
  (if (and (= (get (car lst) 'ordre) ordre)
                (setq tmp (ME-Get-Filter (car lst))))
      (progn
        (rplacd (getcar tmp) nil)
        (putcar tmp (car lst))
        (putcar tmp 'pattern)
        (putcar tmp (ME-Make-Color (get (car lst) 'couleur)))
        (putcar tmp 'couleur)
        (ifn ret (setq ret (cons tmp)) (rplacd (last ret) (cons tmp)))))
  (nextl lst))
     (if tmp (self (1+ ordre) ret))
     ret))

(de ME-Get-Filter (pat)                                                    ME-Get-Filter
   (ifn pat nil
   (if (get pat 'filter) (get pat 'filter)
     (let (filter (car (ME-Make-Filter (get pat 'pattern) (eq (get pat 'match) 'partiel))))
       (put pat 'filter filter)
       filter))))

(de ME-Make-Filter (pattern is-partiel)                                    ME-Make-Filter
   (let (pat pattern)
     (cond
       ((atom pat) nil)
       ((and (listp (cadr pat)) is-partiel) (cons (append (self (cadr pat)) '(R)) (self (cddr pat))))
       ((listp (cadr pat)) (cons (self (cadr pat)) (self (cddr pat))))
       (t
  (cons (if (eq (get (car pat) 'type) 'compo) (cons 'P (get (car pat) 'pat-compo)) (cons 'A (gensym)))
        (self (cdr pat)))))))

(de ME-Make-Filter-From-Tree (tree)                                        ME-Make-Filter-From-Tree
   (let (t-e tree)
     (cond
       ((null t-e) nil)
       ((atom t-e) `(A . ,(gensym)))
       (t
  (cons (self (car tree)) (self (cdr tree)))))))

(de ME-Make-Tree ()                                                        ME-Make-Tree
   (if (and (eq (get ME-Windows 'ga-tree) GA-Tree) (get ME-Windows 'me-tree)) (get ME-Windows 'me-tree)
     (let (nodes (xGetValues (get ME-Windows 'tree) "children"))
  (let (me-tree (ME-Build-Tree (cadr nodes) (xGetValues (cadr nodes) "treeChildren")))
    (put ME-Windows 'me-tree me-tree)
    me-tree))))

(de ME-Build-Tree (node node-c)                                            ME-Build-Tree
   (cond
    ((null node) nil)
    ((null node-c) node)
    (t (mapcar node-c (lambda (x) (ME-Build-Tree x (xGetValues x "treeChildren")))))))

; action = find ou compute

(de ME-Match ()                                                            ME-Match
   (ifn (> (xGetValues (get ME-Windows 'creat) "numChildren") 1) t
                                          ; raz des precedents resultats
   (if (get ME-Windows 'compute) (ME-Compute-Match t))
   (mapc (cdr (xGetValues (get ME-Windows 'tree) "children"))
        (lambda (x)
              (if (get x 'selected) (put x 'pattern t)
                (xSetValues x "background" "#fff")
                (remprop x 'pattern)
                (remprop x 'match)
                (remprop x 'matched))))
                                          ; construction des filtre
   (let ((filtres (ME-Make-Patterns)) (aux))
     (ifn  filtres nil
                                          ; parcour de l'arbre
              (let (tree (setq aux (ME-Make-Tree)))
                (ifn tree t
                                          ; parcours des filtres
```

```
                                (ifn (eq tree aux)
                                    (mapc filtres
                                            (lambda (x)
                                              (if (setq aux (Meta-Match x tree)) (ME-Compute-Match-Find aux x)))))
                                (princ ".")
                                (cond
                                 ((atom (car tree)) (self (cdr tree)))
                                 (t (self (car tree))
                                    (self (cdr tree))))))))))

        (de ME-Compute-Match-Find (m-result filtre)                                    ME-Compute-Match-Find
           (print m-result filtre)
           (let ((pat (get-car-value filtre 'pattern)) (elem) (t-par))
             (put ME-Windows 'find t)
             (let ((ftr filtre) (p-level 0))
            (cond
             ((atom ftr) nil)
             ((or (eq (car ftr) 'R) (eq (car ftr) 'P)) nil)
             ((atom (car ftr))
              (setq elem (car (%find m-result (cdr ftr))))
              (setq t-par elem)
              (repeat (1+ p-level)
                        (xSetValues t-par "background"  (get-car-value filtre 'couleur))
                        (setq t-par (xGetValues t-par "treeParent")))
              (put elem 'pattern pat)
              (put elem 'match m-result))
             ((listp (car ftr))
              (self (car ftr) (1+ p-level))
              (self (cdr ftr) 0))))))

        ; Realisation Graphique du match
        ; si is-undo est a t, le graphique est restaure

        (de ME-Compute-Match (is-undo)                                                 ME-Compute-Match
           (ifn (get ME-Windows 'find) (ME-Match))
           (if is-undo (ME-Graphique t))
           (put ME-Windows 'compute (not is-undo))
           (ifn is-undo (xUnmap (xGetValues (get ME-Windows 'tree) "parent")))
           (let (tree (ME-Make-Tree))
             (if (or (atom tree) (null tree)) nil
           (if (and (atom (car tree))
                    (setq aux (get (car tree) (if is-undo 'matched 'match)))) (ME-Compute-Element aux is-undo))
           (self (car tree))
           (self (cdr tree))))
            (ifn is-undo (xMap (xGetValues (get ME-Windows 'tree) "parent"))))


        (de ME-Compute-Element (list-of-elem is-undo r-stack)                          ME-Compute-Element
          (let ((c-tag (get (cadar list-of-elem) 'tag))
           (r-elem (ifn is-undo (xGetValues (cadar list-of-elem) "treeParent")))
           (c-elem (cadar list-of-elem))
           (l-elem list-of-elem) (tmp))
            (cond
             ((null l-elem)
              (unless is-undo
                 (let (p-elem (get c-elem 'pattern))
                        (GLnewlist (setq tmp (Find-Object-List-From-Tag c-tag)))
        ;               (GLcalllist (+ tmp curent-to-object))
                        (eval (get p-elem 'graphique))
                        (GLendlist)
                        (mapc r-stack
                             (lambda (x)
                                    (when (and (iswidget x) (null (xGetValues x "treeChildren")))
                                         (if (eq x r-elem) (setq r-elem (xGetValues x "treeParent")))
                                         (xRemoveWidget x))))

                        (ifn (member r-elem r-stack) (xCreateWidget '--NW-- "awLabel" (xGetValues r-elem "parent")
                                                                    "treeParent" r-elem "bitmap" Tree-Bitmap
                                                                    "background" (ME-Make-Color (get p-elem
'couleur) t))
                             (xSetValues r-elem "bitmap" Tree-Bitmap "background" (ME-Make-Color (get p-elem
'couleur) t))))))
              (t
               (GLnewlist (setq tmp (Find-Object-List-From-Tag (get c-elem 'tag))))
               (cond
                (is-undo
```

```
        (put c-elem 'match (get c-elem 'matched))
        (remprop c-elem 'matched)
        (GLcalllist (+ tmp curent-to-object)))
            (t
        (put c-elem 'matched (get c-elem 'match))
        (remprop c-elem 'match)
        (when (iswidget c-elem)
            (if (< (xGetValues (setq tmp (xGetValues c-elem "treeParent")) "y") (xGetValues r-elem "y")) (setq r-
elem tmp)
                    (ifn (member tmp r-stack) (setq r-stack (cons tmp r-stack))))
              (xRemoveWidget c-elem))))
          (GLendlist)
          (self (if (< c-tag (setq tmp (get (cadar l-elem) 'tag))) c-tag tmp)
           r-elem
           (if (cdr l-elem) (cadar (cdr l-elem)) c-elem)
           (cdr l-elem))))))
```

Annexe I.13 Animation des représentations : Zin-out.vlisp

```
;
; Fonctions de points d'entree et de sortie en actions supplementaires
;


(de Init-Execution-Variables ()                                    Init-Execution-Variables
  (setq func-is-first-entree t exe-in nil exe-out nil
    exe-nfunc 0                             ;
    exe-func nil                            ; fonction actuellement en cours
    exe-p-func nil          ; fonction précédemment en cours
    exe-args nil            ; arguments de la fonction
    exe-p-args nil
    exe-sortie nil          ; valeur en sortie de la fonction
    exe-nth 0                       ; rang d'appel
        exe-n-in 0                  ; nombre d'entree (func + var)
    exe-n-out 0                     ; nombre de sorties (func seulement)
    exe-elem-nth 0                  ; taille de la pile d'accès à l'élément
    exe-nth-elem-nth 0      ; nombre d'accès à la même position dans la
                                    ; pile d'accès à l'élément
    exe-is-var nil                  ; arret sur variable ?
    exe-value nil))                 ; valeur des variables


                                    ; ****************************** ; ;
                                    ; f-entree et f-sortie pour Meta ; ;
                                    ; ****************************** ; ;

(de find-parent ()                                                 find-parent
  (let (x 7)
    (while (and (setq aux (frame x t))
                (not (and (listp aux)
                          (listp (fval (car aux)))
                          (listp (ival (car aux)))
                          (cdr (ival (car aux))))))
      (incr x))
    (car aux)))


; f-entree et f-sortie ne sont pas prises en compte en mode non-careful...

(de f-entree (obj type func-instr is-rec)                          f-entree
  (ifn is-rec (careful nil))
  (GLwinset Meta-GL)
  (setq f-entree t exe-in t)
  (incr exe-n-in)
  (let ((graphic-obj) (chk-sum))
    (cond
                                    ; variables
     ((eq type 'var)
      (if (GetVarData obj 'v-stop) (Z-Step-In obj "Arret sur la Variable"))
      (cond
```

```
        ((Var-In-Context obj exe-func)
        (if (GetVarData obj 'v-obj) (Z-Step-Label obj "Variable"))
        (PutVarData obj 'v-access (+ (GetVarData obj 'v-access) 1))
        (setq exe-elem-nth (GetVarData obj 'v-access)
                exe-var obj
                exe-value (eval obj))
        (if (setq aux (GetVarData obj 'v-access-n))
           (put aux exe-elem-nth (setq exe-n-elem-nth (+ 1 (get aux exe-elem-nth))))
          (put (setq aux (gensym)) exe-elem-nth (setq exe-n-elem-nth 1))
          (PutVarData obj 'v-access-n aux))
        (Update-Var-Context obj exe-func exe-value)
        (setq exe-var-init (cadr (member 'init (GetVarData obj exe-func)))
                exe-var-min (cadr (member 'min (GetVarData obj exe-func)))
                exe-var-max (cadr (member 'max (GetVarData obj exe-func))))
        (let ((v-objs (GetVarData obj 'v-obj))
                (v-obj (GetObjectByName (caar (GetVarData obj 'v-obj))))
                (has-computed nil))
          (cond
           ((null v-objs)
            (if (and exe-dynamic-dump (null is-rec))
                 (DD-Dump-Progress 'in 'var obj exe-func (eval obj) has-computed)))
           ((and (or (null (cadar v-objs)) (eq (cadar v-objs) exe-func))
                 (or (null (setq aux (get v-obj 'v-test))) (eval aux)))
            (IO-Eval obj v-obj exe-func 'gl-dyn-in (cadar v-objs))
            (self (cdr v-objs) (GetObjectByName (car (cadr v-objs))) t))
           (t (self (cdr v-objs) (GetObjectByName (car (cadr v-objs))) has-computed)))))
         ((and exe-dynamic-dump (null is-rec))
       (DD-Dump-Progress 'in 'var obj exe-func (eval obj)))))

                                                ; arret sur l'entree d'une fonction
     ((eq type 'func)
                                                ; reinitialisation des valeurs des variables en cas de premier
acces a la premiere fonction
                                                ; du programme
        (setq exe-func obj exe-p-func (find-parent) exe-p-args (and exe-p-func (car (fval exe-p-func)))
         exe-n-recurse (+ 1 (GetFuncData obj 'n-recurse)))
        (PutFuncData obj 'n-recurse exe-n-recurse)
        ; affichage de l'entrée de la fonction dans la représentation arborescente du programme
        (if (GetFuncData obj 'tree-w)
        (IO-Update-TreeW (GetFuncData obj 'tree-w) W-MColor
                         ((if (> exe-n-recurse 15) 15 exe-n-recurse) W-Colors)))
        ; sauvegarde des coordonnées de l'origine de l'appel
        (when (and func-is-first-entree (eq obj (get (GetFuncData obj 'root-obj) 'func)))
       (cond
        ((not (boundp 'Gl-Dyn-Lists)) (setq Gl-Dyn-Lists nil))
        ((numbp (car Gl-Dyn-Lists))
         (mapc Gl-Dyn-Lists 'GLdellist)
         (setq curent-object-list (- (car (last Gl-Dyn-Lists))))
         (setq Gl-Dyn-Lists nil)))
       (setq p-instr nil)
       (mapc (GetFuncData (GetFuncData obj 'root-obj) 'f-used) (lambda (x) (PutFuncData x 'f-depth 0)))
       (mapc localvar 'Clear-Var-Context)
       (mapc globalvar 'Clear-Var-Context))
        (PutFuncData obj 'f-depth (+ (GetFuncData obj 'f-depth) 1))
        (setq com-instr obj exe-args (car (fval obj)))
                                                ; maj des variables parametre (si besoin)
        (mapc (car (fval obj))
         (lambda (aParam) (if (GetVarData aParam 'v-obj) (f-entree aParam 'var nil nil t))))
                                                ; execution des entrees
        (if exe-dynamic-dump (DD-Dump-Progress 'in 'func obj nil nil (GetFuncData obj 'o-in)))
        (when (GetFuncData obj 'o-in)
        (Set-Composition-Numeration com-instr)
        (Z-Step-Label obj "Entree dans")
        (setq exe-nth      (incr exe-nfunc)
                exe-elem-nth (GetFuncData obj 'f-depth)
                exe-is-var nil exe-instr obj  com-instr obj)
        (if (setq aux (GetFuncData obj 'f-depth-n))
           (put aux exe-elem-nth (setq exe-n-elem-nth (+ 1 (get aux exe-elem-nth))))
          (put (setq aux (gensym)) exe-elem-nth (setq exe-n-elem-nth 1))
          (PutFuncData obj 'f-depth-n aux))
        (mapc (GetFuncData obj 'o-in)
                (lambda (o)
                  (IO-Eval obj (GetObjectByName o) nil 'gl-dyn-in))))
        (PutFuncData exe-func 'ga-values GA-Values)
        (if (GetFuncData obj 's-in) (Z-Step-In obj "Entree dans")))
                                                ; instructions
```

```
    (t
     (setq exe-func func-instr)
     (when (and (GetFuncData func-instr 'p-code) ; les sources du programme sont affichées
                (setq aux (get-car-value obj 'l_nb))) ; changement de ligne
   (FM-Display-Code-Running func-instr aux))
     (when (GetFuncData func-instr 'i-in)
   (Z-Step-Label func-instr "Progression dans")
   (if exe-dynamic-dump (DD-Dump-Progress 'in 'instr obj func-instr nil t))
   (if (listp obj)
       (progn
        (replace-car obj 'i-access (+ (get-car-value obj 'i-access) 1))
        (setq exe-elem-nth (get-car-value obj 'i-access))))
   (if (setq aux (get-car-value obj 'i-access-n))
         (put aux exe-elem-nth (setq exe-n-elem-nth (+ 1 (get aux exe-elem-nth))))
       (put (setq aux (gensym)) exe-elem-nth (setq exe-n-elem-nth 1))
       (replace-car obj 'i-access-n aux))
   (setq exe-is-var nil exe-instr obj)
   (mapc (GetFuncData func-instr 'i-in)
         (lambda (x)
                 (IO-Eval obj (GetObjectByName x) nil 'gl-dyn-in)))
   (setq exe-old-instr obj)))))
  (setq f-entree nil exe-in nil)
  (ifn is-rec (careful t)))

(de f-sortie (obj type func-instr)                                    f-sortie
  (careful nil)
  (GLwinset Meta-GL)
  (setq f-sortie t exe-out t exe-in nil exe-is-var nil)
  (incr exe-n-out)
  (let (graphic-obj)
    (cond
                                          ; arret a la sortie d'une instruction
     ((eq type 'instr)
      (when (GetFuncData func-instr 'i-out)
   (Z-Step-Label (car obj) "Sortie de l'instruction")
   (setq exe-elem-nth (get-car-value obj 'i-access)
         exe-n-elem-nth (get (get-car-value obj 'i-access-n) exe-elem-nth)
         exe-instr obj
         exe-func func-instr)
   (if exe-dynamic-dump (DD-Dump-Progress 'out 'instr obj exe-func nil t))
   (mapc (GetFuncData func-instr 'i-out)
         (lambda (x)
                 (IO-Eval obj (GetObjectByName x) nil 'gl-dyn-out)))
   (setq exe-old-instr obj)))
                                          ; arret a la sortie d'une fonction
     ((eq type 'func)
      (setq com-instr obj func-instr obj exe-func obj
       exe-p-func (find-parent)
       exe-n-recurse (1- (GetFuncData obj 'n-recurse))
       exe-args (car (fval obj))
       exe-p-args (and exe-p-func (car (fval exe-p-func)))
       exe-sortie sortie)
      (PutFuncData obj 'n-recurse exe-n-recurse)
      (if (and (GetFuncData obj 'tree-w)
           (= 0 exe-n-recurse))
       (IO-Update-TreeW (GetFuncData obj 'tree-w) W-Black W-White))

      ; restauration des coordonnées de l'appel
      (when (and exe-p-func (setq aux (GetFuncData exe-p-func 'ga-values)))
   (setq p-values aux)
   (Set-GA-Values t))
      ; point d'arrêt sur la valeur de sortie de la fonction ?
      (if (and (Var-In-Context 'sortie func-instr) (GetVarData 'sortie 'v-obj))
       (f-entree 'sortie 'var func-instr))
      (ifn (GetFuncData obj 'o-out) (if exe-dynamic-dump (DD-Dump-Progress 'out 'func obj))
       (if exe-dynamic-dump (DD-Dump-Progress 'out 'func obj nil nil t))
       (Z-Step-Label obj "Sortie de")
       (Set-Composition-Numeration com-instr)
       (PutFuncData obj 'f-depth (- (setq exe-elem-nth (GetFuncData obj 'f-depth)) 1))
       (setq exe-n-elem-nth (get (GetFuncData obj 'f-depth-n) exe-elem-nth)
                 exe-nth      (incr exe-nfunc)
                 exe-instr obj  com-instr obj)
       (mapc (GetFuncData obj 'o-out)
                 (lambda (x)
                   (IO-Eval obj (GetObjectByName x) nil 'gl-dyn-out)))
       (Set-GA-Values t))
```

```
    (if (GetFuncData obj 's-out) (Z-Step-In obj "Sortie de")))))
  (setq f-sortie nil exe-out nil)
  (if (get MPV-Root-Window 'io-text) (Z-Step-Label " " "Programme fini"))
  (careful t))


                                        ; ******************** ; ;
                                        ; Action a l'execution ; ;
                                        ; ******************** ; ;

; maj couleur arborescence
(de IO-Update-TreeW (lst fg bg)                                          IO-Update-TreeW
  (mapc lst
   (lambda (aW)
     (if (iswidget aW) (xSetValues aW "foreground" fg "background" bg)))))

; misc

(de IO-Set-GA-Values (gl-obj)                                            IO-Set-GA-Values
  (setq GA-Values (car (GL-Get-Transfo-Values gl-obj)))
  (Set-GA-Values))

(defmacro Set-GA-Values (is-p-values)                                    Set-GA-Values
  (if is-p-values
      '(setq p-x (Mdiv (1 p-values) scale-factor)
        p-y (Mdiv (2 p-values) scale-factor)
        p-z (Mdiv (3 p-values) scale-factor))
    '(setq GA-X (Mdiv (1 GA-Values) scale-factor)
      GA-Y (Mdiv (2 GA-Values) scale-factor)
      GA-Z (Mdiv (3 GA-Values) scale-factor))))

; evaluation de l'action a partir d'une fonction

(de IO-Eval (f-obj g-obj func-on-var gl-mode var-context)                 IO-Eval
  (let ((r-act (get g-obj 'action-g))
   (gl-objects))

    (if Flow-Verbose (print "IO-Eval" r-act f-obj (get g-obj 'name)))
    (when (or (eq r-act 'eff-recons) (eq r-act 'recons))
      (setq ga-cond 'EXE)
      (PRC-Init-Global-Variables f-obj)
      (PRC-Set-Var-Values))
                                        ; dessin de l'objet
                                        ; s'il n'existe pas
    (unless (or (GL-Get-GL-From-ORS f-obj (get g-obj 'name) func-on-var)
                (get g-obj 'gl-eval)
                (member r-act '(recons mv-uniq-recons eff-recons)))
      (eval `(ReplaceNewList ,(get g-obj 'gl-list) ,@(get g-obj 'gl-data)))
      (put g-obj 'gl-eval t))

    (IO-Eval-Action (or (GL-Get-GL-From-ORS f-obj (get g-obj 'name) func-on-var)
                        (car (GI-Get-a-GL f-obj)))
                    (get g-obj 'list))

    (if (get g-obj gl-mode) (IO-Eval-gl-mode))
                                        ; sauvegarde des donnees de position et de parenté
    (setq p-x GA-X p-y GA-Y p-z GA-Z p-instr com-instr)))

; évaluation des actions liés aux ORS

(de IO-Get-List (obj-name f-obj)                                         IO-Get-List
  (let (gl-elem (GL-Get-GL-From-ORS f-obj obj-name func-on-var))
    (GL-Get-LList gl-elem)))

(de IO-Eval-Action (gl-obj gl-num)                                       IO-Eval-Action
  (if gl-obj (IO-Set-GA-Values gl-obj))

  (ifn p-instr (setq p-x GA-X p-y GA-Y p-z GA-Z))
  (cond
   ((eq r-act 'transfo)   (IO-Apply-Transformation))

   ((eq r-act 'recons)
    (PRC-Generate-ORS g-obj f-obj nil 'a)) ; adding new graphic (a)

   (r-act
    (ifn (eq r-act 'eff-recons) (IO-Set-New-Position))
```

```
    (ifn (eq r-act 'mv-uniq)    (PRC-Generate-ORS g-obj f-obj nil 'r))))) ; replacing old value (r)


; application des contraintes si nécessaire
(de Set-Trans-With-Constraint (n . vals)                                    Set-Trans-With-Constraint
  (rplaca (nth (1+ n) exe-translation)
    (eval (ifn (setq aux (get g-obj 'mv-constraint)) (car (nth n vals))
                `(,(car (nth n aux)) p-x p-y p-z GA-X GA-Y GA-Z . ,vals)))))

; deplacement de l'objet graphique
(de IO-Set-New-Position ()                                                  IO-Set-New-Position
  (let ((cpt) (dx) (dy) (dz) (n) (ix) (iy) (iz))
    (setq cpt  (ifn (boundp 'path-steps) (setq path-steps 3) path-steps)
     dx (- GA-X p-x) dy (- GA-Y p-y) dz (- GA-Z p-z))
    (if (= 0 cpt) (setq cpt 1))
    (setq ix (/ dx cpt)
     iy (/ dy cpt)
     iz (/ dz cpt)
     n cpt)
    (if (and (= ix 0) (= iy 0) (= iz 0)) (setq cpt 1 ix dx iy dy iz dz))

    (if (and (= ix 0) (= iy 0) (= iz 0)) nil
      (while (> n -1)
    (GLnewlist (- gl-num curent-to-object)) (GLpushmatrix)
    ; si jamais qqun a envi de mettre du graphique dans la gestion des contraintes...
    (Set-Trans-With-Constraint 1 (- GA-X (* n ix)) (- GA-Y (* n iy)) (- GA-Z (* n iz)))
    (Set-Trans-With-Constraint 2 (- GA-X (* n ix)) (- GA-Y (* n iy)) (- GA-Z (* n iz)))
    (Set-Trans-With-Constraint 3 (- GA-X (* n ix)) (- GA-Y (* n iy)) (- GA-Z (* n iz)))
    (eval exe-translation)
    (GLcalllist gl-num)
    (GLpopmatrix)
    (GLendlist)
    (if exe-dynamic-dump (DD-Dump-GL (- gl-num curent-to-object)))
    (if (get MPV-Root-Window 'io-follow)
       (IO-Do-Follow (- GA-X (* n ix)) (- GA-Y (* n iy)) (- GA-Z (* n iz))
                     GA-X GA-Y GA-Z))
    (decr n)))))


; application des transformations
(de IO-Apply-Transformation (f-what f-to has-color has-moved)               IO-Apply-Transformation
  (let ((to (cdr (GL-Get-Transfo-Values gl-obj)))
    (what (get g-obj 'gl-transfo)))
     (mapc '((color c) (translate t) (rotate r) (scale s))
      (lambda (x)
        (when (get what (car x))
          (setq f-what (MkTransfo (cadr x) (get what (car x)) (if (eq (cadr x) 's) 1 0))
                f-to   (cdr (assoc (cadr x) to)))
          (when f-what
                (ifn (eq (cadr x) 'c) (setq has-moved t))
                (cond
                 ((null f-to) (rplacd (GL-Get-Transfo-Values gl-obj) (cons f-what to)))
                 (t
                  (mapc '(1 2 3) (lambda (x) (rplaca (nth x f-to) ((1+ x) f-what))))
                  (if (and (eq (cadr x) 's) (5 f-what))
                     (if (nth 4 f-to) (rplaca (nth 4 f-to) (5 f-what))
                           (rplacd (nth 3 f-to) (cons (5 f-what) nil)))))))))))
    (when to
      (ReplaceNewList (caar (last to))
                (GLpushmatrix)
                (mapc to
                          (lambda (x)
                            (cond
                             ((eq (car x) 'c) (GLpushAttrib "GL_CURRENT_BIT") (GLcolor 3 (2 x) (3 x) (4 x)))
                             ((eq (car x) 's) (GLscale (2 x) (3 x) (4 x) (5 x)))
                             ((eq (car x) 'r) (GLrotate (2 x) (3 x) (4 x)))
                             ((eq (car x) 't) (GLtranslate (2 x) (3 x) (4 x)))))))

      (ReplaceNewList (cadar (last to))
                (GLpopmatrix)
                (when has-color (GLpopAttrib))))
                                        ; modification de la sauvegarde de la position reelle de l'objet, le
calcul est fait
                                        ; par GA-Set-Delta (Mgraphiques)...
    (when has-moved
      (setq pere nil)
```

```
        (setq GA-Trans-Stack (GL-Get-Transfo gl-obj))
        (GA-Set-Delta)
        (setq GA-Trans-Stack nil)
        (setq to (car (GL-Get-Transfo-Values gl-obj)))
        (rplaca (nth 1 to) (- GA-Delta-X))
        (rplaca (nth 2 to) (- GA-Delta-Y))
        (rplaca (nth 3 to) (- GA-Delta-Z))
        (if (get MPV-Root-Window 'io-follow)
       (IO-Do-Follow  (Mdiv (1 to) scale-factor) (Mdiv (2 to) scale-factor) (Mdiv (3 to) scale-factor)
                      GA-X GA-Y GA-Z))
        (setq GA-Values to) (Set-GA-Values))))

; ajout d'une nouvelle construction
(de IO-Add-GL ()                                                    IO-Add-GL
  (let (a-list (get g-obj 'add-list))
    (ifn (setq aux (member gl-num a-list))
    (ifn a-list (setq aux (setq a-list (cons gl-num nil)))
        (rplacd (last a-list (setq aux (cons gl-num nil))))))))


; évaluation des actions liés aux instructions graphiques supplémentaires

(de IO-Eval-gl-mode ()                                              IO-Eval-gl-mode
                                    ; Recherche des objets graphiques liés à l'objet observé dans
tous les parcours
    (let ((Prcs (cdr Z-Parcours))
     (***gl-object*** (GetPrcData (PRC-Nom (car Z-Parcours)) f-obj 'gl-object)))
      (cond
       ((null Prcs) nil)                ; fini...
       ((null ***gl-object***)
      (self (cdr Prcs)
          (GetPrcData (PRC-Nom (car Prcs)) f-obj 'gl-object)))

       ((or (null func-on-var)                  ; on ne parle pas de variable
       (null var-context)               ; elle est de context global
       (and (eq (GL-Get-TDEP (car ***gl-object***)) 'fdd-VAR)
                (eq func-on-var (GL-Get-DDEP (car ***gl-object***))))
       (and (eq (GL-Get-TDEP (car ***gl-object***)) 'fdd-FUNC)
                (eq f-obj (GL-Get-DDEP (car ***gl-object***)))))

      (IO-Set-GA-Values (car ***gl-object***))
      (eval `(ReplaceNewList ,(- (GL-Get-List (car ***gl-object***)) 1)
                             ,@(get g-obj gl-mode)))

      (self Prcs (cdr ***gl-object***)))
        (t
      (self Prcs (cdr ***gl-object***)))))))
```

Annexe I.14 Création de l'historique : Zdump.vlisp

```
                                    ; ********************************************* ; ;
                                    ; Dump dynamique de l'historique de l'execution ; ;
                                    ; ********************************************* ; ;


                                    ; ********************* ; ;
                                    ; Création du dump ; ;
                                    ; ********************* ; ;

(df DD-Dump (what)                                                  DD-Dump
   (status print 2)
   (output exe-dump-file)
   (eval (cons 'print what))
   (output)
   (status print 0))

(de DD-Dump-New-Step ()                                             DD-Dump-New-Step
  (if (not (= dump-step 0)) (close dump-file))
  (incr dump-step)
  (setq dump-file (eval `(open ,(strcat exe-dump-file-gl "-" dump-step) "w"))))
```

```
(de DD-Dump-GL (lst)                                                    DD-Dump-GL
  (MESAwritelist dump-file lst t))

; initialisation (à la définition d'un ORS racine)
(de DD-Dump-Root-Object (obj)                                           DD-Dump-Root-Object
  (DD-Dump "(DD-def-root-object '" (get obj 'name) " "
     (if (setq aux (get obj 'source-file)) (strcat "\"" aux "\""))
     ")"))

; initialisation d'une génération
(de DD-Dump-Generation (obj)                                            DD-Dump-Generation
  (print 'generation (get obj 'name) exe-dump-file)
  (DD-Dump "(DD-def-generation '" (get obj 'name) " '" (get obj 'func)
     (if (setq aux (getival (get obj 'func) 20)) (strcat " \"" (car aux) "\""))
     ")"))

; indication de la fin des générations initiales
(de DD-Dump-End-Generation ()                                           DD-Dump-End-Generation
  (DD-Dump "(DD-end-def-generation " (get Meta-GL 'root-list) ")")
  (setq dump-step 0 dump-file nil))


; dump de la progression de l'exécution
(de DD-Dump-Progress (i-o type obj func var-value has-gl)               DD-Dump-Progress
  (if has-gl (DD-Dump-New-Step))
  (DD-Dump "(DD-program-progress '" i-o " '" type " '" obj " '"
     func " '"
     (if (eq type 'var) var-value
       (if (eq type 'func)
                (if (eq i-o 'in) (mapcar (car (fval obj)) 'eval)
                  (eval sortie))))
     " '" has-gl " " (if has-gl dump-step) ")"))


                                          ; ************************** ; ;
                                          ; Lecture d'un dump dynamique ; ;
                                          ; ************************** ; ;

; load installe la liste des dumps
(de DD-Load (obj w wc)                                                  DD-Load
  (setq exe-dump-include nil
   dump-files-to-load nil)
  (let ((f-num 0) (f-list nil))
    (setq dump-root-obj nil dump-root-func nil
     dumped-objs nil n-dumped-objs 0)
    (cond
     ((probef (setq exe-dump-file (strcat default-dump-directory "Zeugma-dump-" f-num)))
      (letf (EOF () (input) (End-Header-Read))
       (escape End-Header-Read
                  (input exe-dump-file)
                  (while t (eval (read)))))
      (self (1+ f-num) (cons (strcat "Fonction " dump-root-func" Racine " dump-root-obj) f-list)))
     (t
      (setq f-list (ifn f-list [(if (get MPV-Root-Window 'd-dump) "Anulation de la création" "Création d'un
dump")]
                       (cons (if (get MPV-Root-Window 'd-dump) "Anulation de la création" "Création d'un
dump")
                             (reverse f-list))))
      (let (l-w (xCreateList "Dump"  f-list 1 default-font 280 100))
     (xAddCallback l-w W-ClBck (strcat "(DD-Init-Load-Dump '$W $l '" wc ")"))))))

; retour de la sélection dans la liste des dumps : création ou lecture....
(de DD-Init-Load-Dump (w d-num wc)                                      DD-Init-Load-Dump
  (if w (xRemove w))
  (cond
   ((= d-num 0)
    (xSetValues wc "bitmap" (strcat bitmap-directory (if (get MPV-Root-Window 'd-dump) "exe-dump" "dum-
ping")))
    (put MPV-Root-Window 'd-dump (setq exe-dynamic-dump (null (get MPV-Root-Window 'd-dump))))
    (let (x -1)
     (while (probef (setq exe-dump-file (strcat default-dump-directory "Zeugma-dump-" (incr x))))))
    (setq exe-dump-file-gl (strcat exe-dump-file "-gl")))
   (t
    (setq exe-dump-file (strcat default-dump-directory "Zeugma-dump-" (1- d-num))
     exe-dump-file-gl (strcat exe-dump-file "-gl"))
```

```
    (setq exe-dump-include t)
    (DD-Reset-Dump-Vars)
    (DD-Interface)
    (GLwinset (setq Meta-GL DD-Root-Window))
    (xSetValues (get DD-Root-Window 'file) "label"
                (strcat "Fonction(s) " dump-root-func " Racine(s) " dump-root-obj))
    (GLdrive DD-Root-Window)
    (DD-Load-Dump)
    (setq dump-stack-length (length dump-run-stack))
    (DD-Display-Result)
    (DD-Init-dump-stack)
    (Init-Execution-Variables)
    (MPV-Reset-Label))))

; inclusion effective d'une définition
(de DD-Load-Dump (file-to-load is-data-file)                                    DD-Load-Dump
  (MPV-Message (strcat "Chargement du dump" exe-dump-file))
  (letf (EOF () (End-Read-Dump (input)))
    (input exe-dump-file)
    (escape End-Read-Dump (while t (eval (read)))))
  (when dump-files-to-load
    (mapc dump-files-to-load 'Read-File)))

; intialisation des variables utilisées pour le dump
(de DD-Reset-Dump-Vars ()                                                       DD-Reset-Dump-Vars
  (setq dump-root-obj nil       ; objet racine
    dump-root-func nil      ; fonction principale
    dump-initial-stack nil  ; Pile d'initialisation
    dump-generation-stack nil
    dump-files-to-load nil
    dump-run-stack nil      ; Pile d'exécution
    dump-stack-fw nil ; a faire... (à la mode de Turing)
    dump-stack-bk nil ; fait...
    dump-stack-length 0
    dump-files-to-load nil))

; définitions de l'entête des dumps (exe-dump-include) indique si la lecture
; est un chargement ou la construction de la liste des dumps existants)
(de DD-def-root-object (ors-name file)                                          DD-def-root-object
  (if dump-root-obj (rplacd (last dump-root-obj) (cons ors-name nil))
    (setq dump-root-obj [ors-name]))
  (when (and (null (GetObjectByName ors-name))
         (not (member file dump-files-to-load file)))
    (newl dump-files-to-load file)))

; annonce de l'exécution de la génération d'une représentation analogique
; à partir du code des programmes
(de DD-def-generation (ors-name func file)                                      DD-def-generation
  (cond
    ((null dump-root-func) (setq dump-root-func [func]))
    ((not (member func dump-root-func)) (rplacd (last dump-root-func) (cons func nil))))
  (when (null exe-dump-include)
    (when (and (= (ftyp func) 0)
           (not (member file dump-files-to-load)))
      (newl dump-files-to-load file))
    (if (boundp 'dump-generation-stack)
    (rplacd (last dump-generation-stack) (cons ors-name (cons func nil)))
      (setq dump-generation-stack [ors-name func]))))

; fin des initialisations
(de DD-end-def-generation (root-list)                                           DD-end-def-generation
  (if exe-dump-include (setq dump-root-list root-list)
    (EOF)))

;
(setq W-Print-Loop '("-" "\\" "|" "/")) ; |
(rplacd (last W-Print-Loop) W-Print-Loop)
;
; ajout d'un élément dans la pile décrivant l'exécution du programme
;
(de DD-program-progress (i-o type obj func-on-var var-value has-gl gl-step)    DD-program-progress
  (MPV-Message (strcat "Chargement de l'historique  " (nextl W-Print-Loop)))
  ; construction de la pile
  (let (elem [i-o type obj var-value func-on-var has-gl gl-step])
    (cond
```

```
        ((null dump-run-stack)
         (setq tmp-run-stack (setq dump-run-stack (cons elem nil)))
         (setq gl-run-stack nil))
        (t
         (rplacd tmp-run-stack (cons elem nil))
         (nextl tmp-run-stack))))
    ; chargement des opération Open GL
    (if has-gl
        (let (fdes (eval `(open ,(strcat exe-dump-file-gl "-" gl-step) "r")))
      (let ((aListe (MESAreadlist fdes)) (dd-listes nil))
        (cond
         ((null aListe)
          (newl gl-run-stack dd-listes)
          (newl gl-run-stack gl-step))
         (t
          (MESAcopylist aListe (setq aux (GLgenlist)))
          (self (MESAreadlist fdes) (cons aListe (cons aux dd-listes))))))
      (close fdes))))


                                        ; ************************************************ ; ;
                                        ; Creation de l'interface de visualisation des dumps ; ;
                                        ; ************************************************ ; ;

; largeur de l'interface
(setq DD-Width 450)

; bouton de commande pour l'interface
(defmacro DD-NewBtn (bmp cmd left)                                          DD-NewBtn
  `(xAddCallback (setq tmp (xCreateWidget '--AC-- W-Command fw
                                          "bitmap" (strcat bitmap-directory ,bmp)
                                          "fromVert" (get DD-Root-Window 'file)
                                          "fromHoriz" ,left "top" 0 "bottom" 0))
                 "callback" (strcat "(DD-Do '" ,cmd ")")))

; définition de l'interface
(de DD-Interface (tmp)                                                      DD-Interface
  (unless DD-Root-Window
    (setq DD-Root-Window (Initialize-Meta-GL))
    (let (root (xCreateWidget '--AS-- W-AShell "title" "Observation d'un dump dynamique" "iconName"
"Dump"))
      (let (fw (xCreateWidget '--AF-- W-Form root "background" W-BColor))
    (put DD-Root-Window 's-incr 1)
    (put DD-Root-Window 'file (xCreateWidget '--AL-- W-Label fw
                                             "label" "no file loaded"
                                             "width" DD-Width  "top" 0 "bottom" 0))

    (DD-NewBtn 'dmp-rev 'rev)
    (DD-NewBtn 'dmp-rev-play 'rev-play tmp)
    (DD-NewBtn 'dmp-pause 'pause tmp)
    (put DD-Root-Window 'pause tmp)
    (put DD-Root-Window 'stop
        (setq tmp (xCreateWidget '--AC-- W-Command fw
                                 "bitmap" (strcat bitmap-directory 'dmp-stop)
                                 "fromVert" (get DD-Root-Window 'file)
                                 "fromHoriz" tmp  "top" 0 "bottom" 0)))
    (DD-NewBtn 'dmp-play 'play tmp)
    (DD-NewBtn 'dmp-forward 'forward tmp)
    (put DD-Root-Window 'label-func
        (xCreateWidget '--AL-- W-Label fw "label" "                    " "width" DD-Width
                       "fromVert" tmp "top" 0 "bottom" 0))
    (put DD-Root-Window 'label-instr
        (xCreateWidget '--AL-- W-Label fw "label" "                    " "width" DD-Width
                       "fromVert" (get DD-Root-Window 'label-func) "top" 0 "bottom" 0))
    (let (vp (xCreateWidget '--VP-- W-Viewport fw "width" DD-Width "height" 100
                            "fromVert" (get DD-Root-Window 'label-instr)
                            "allowHoriz" "True"  "top" 0
                            "allowVert"  "True"))
      (put DD-Root-Window 'tree-data
          (xCreateWidget '--AT-- W-Tree vp))
      (setq tmp (xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data) "label" "Variable"))
      (xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data) "label" "Valeur" "treeParent" tmp)))
     (xRealizeWaiting root))))


; lancement des générations des représentations analogiques initiales
```

```
(de DD-Display-Result ()                                                    DD-Display-Result
  (when (boundp 'dump-generation-stack)
    (let (tmp-gen dump-generation-stack)
    (while tmp-gen
      (put (GetObjectByName (car tmp-gen)) 'func (cadr tmp-gen))
      (MPV-Build-Analogy (GetObjectByName (car tmp-gen)))
      (setq tmp-gen (cddr tmp-gen))))
    (MPV-End-Analogy-Generation)))

                                          ; ************************************ ; ;
                                          ; Traitement des commandes de l'interface ; ;
                                          ; ************************************ ; ;

(de DD-Do (what)                                                            DD-Do
  (let ((s-incr 0) (is-cont nil))
    (cond
     ((eq what 'pause)   (setq s-incr (get DD-Root-Window 's-incr) is-cont t))
     ((eq what 'play)    (setq s-incr 1 is-cont nil))
     ((eq what 'forward) (setq s-incr 1 is-cont t))
     ((eq what 'rev-play)(setq s-incr -1 is-cont nil))
     ((eq what 'rev)     (setq s-incr -1 is-cont t)))
    (when (or (and (< 0 s-incr) (DD-May-Forward))
          (and (> 0 s-incr) (DD-May-Rewind)))
      (if (> s-incr 0) (DD-Forward) (DD-Rewind))
      (while is-cont
    (cond
     ((xCheckEvent (get DD-Root-Window 'stop) "ButtonPress") (setq is-cont nil))
     ((xCheckEvent (get DD-Root-Window 'pause) "ButtonPress") (setq is-cont nil)
      (put DD-Root-Window 's-incr s-incr))
     (t
      (if (or (and (< 0 s-incr) (not (DD-May-Forward)))
                  (and (> 0 s-incr) (not (DD-May-Rewind)))) (setq is-cont nil)
        (if (> s-incr 0) (DD-Forward) (DD-Rewind)))))))))


(defmacro DD-Has-GL (elem) `(6 ,elem))                                      DD-Has-GL
(defmacro DD-Get-GL-Step (elem) `(nth 7 ,elem))                             DD-Get-GL-Step

; vérification de la possibilité d'aller en avant ou en arrière
(defmacro DD-May-Forward ()  '(DD-Test-Next (cdr dump-stack-fw)))           DD-May-Forward
(defmacro DD-May-Rewind ()   '(DD-Test-Next dump-stack-bk))                 DD-May-Rewind

(de DD-Test-Next (aStack)                                                   DD-Test-Next
  (and aStack
       (or (DD-Has-GL (car aStack))
      (DD-Test-Next (cdr aStack)))))

; mise à jours de valeurs liées à un élément du programme
(de DD-Set-Context (elem isfwd)                                             DD-Set-Context
  (selectq (2 elem)
    (var
     (eval `(setq ,(3 elem) ',(4 elem))))
    (func
     (if (eq (car elem) 'out) (setq sortie (4 elem))
       (let ((Params (car (fval (3 elem))))
         (Values (4 elem)))
     (when Params
       (eval `(setq ,(car Params) ',(car Values)))
       (self (cdr Params) (cdr Values))))))
    (t nil)))


; retour arrière
(de DD-Rewind ()                                                            DD-Rewind
                                          ; comme on pointe sur un graphique, on retourne d'un pas en ar-
rière
  (while (and dump-stack-bk
          (not (DD-Has-GL (setq aux (newl dump-stack-fw (nextl dump-stack-bk))))))
    (DD-Reinit-Values aux))               ; les valeurs des variables en jeu reviennent ainsi à leur valeur
précédente

  (if (and (eq (caar dump-stack-fw) 'out)
       (eq (cadar dump-stack-fw) 'func))
      (remprop DD-Root-Window 'disp-func)) ; on force le réaffichage des valeurs
  (DD-Update-Values (car dump-stack-fw) t))
```

```
; avancement
(de DD-Forward ()                                                    DD-Forward
                                        ; recherche du prochain élément non graphique
  (newl dump-stack-bk (nextl dump-stack-fw))

  (while (not (DD-Has-GL (car dump-stack-fw))) ; recherche du prochain graphique
    (if (and (eq (caar dump-stack-fw) 'out)
        (eq (cadar dump-stack-fw) 'func))
   (remprop DD-Root-Window 'disp-func)) ; on force le réaffichage de toutes les données
    (DD-Update-Values (car dump-stack-fw))
    (newl dump-stack-bk (nextl dump-stack-fw)))

  (DD-Update-Values (car dump-stack-fw)))

; initialisation des piles....
(de DD-Init-dump-stack ()                                            DD-Init-dump-stack
  (setq dump-stack-fw dump-run-stack
   dump-stack-bk nil
   is-in-dump t)

  (while (and dump-stack-fw (not (DD-Has-GL (car dump-stack-fw))))
    (DD-Update-Values (newl dump-stack-bk (nextl dump-stack-fw))))

  (DD-Update-Values (car dump-stack-fw)))

; récupération des valeurs précédent d'une mise à jour
(de DD-Reinit-Values (i-o type obj func-on-var)                      DD-Reinit-Values
  (cond
   ((eq type 'instr) t) ; rien à faire
   ((eq type 'func)
    (when (eq  i-o 'out) ; recherche de l'entrée correspondante
      (mapc (append (getival obj 1) (getival obj 0))
       (lambda (aVar)
         (when aVar
                (DD-Display-Values (DD-Reinit-Values nil 'var aVar func)))))))

   (t ; les variables
    (let (stk dump-stack-bk)
      (while (and stk
                  (not (and (eq obj (3 (car stk))) ; la variable
                            (eq func (5 (car stk))))) ; la fonction
                  (not (and (eq 'in (caar stk)) ; ou entrée dans la fonction
                            (eq 'func (2 (car stk)))
                            (eq func (3 (car stk))))))
   (nextl stk))
     (if (eq 'var (2 (car stk))) (eval `(setq ,obj ',(4 (car stk))))
   (let ((Params (car (fval func))) (Values (4 (car stk))))
     (cond
      ((null Params) nil) ; pas trouvé ?
      ((eq (car Params) obj) (eval `(setq ,obj ',(car Values))))
      (t (self (cdr Params) (cdr Values))))))
      (car stk)))))



; mise à jours des valeurs affichées dans l'interface (Update) à partir d'une
; indication de progression dans l'exécution
(de DD-Update-Values (elem is-back)                                  DD-Update-Values
  (DD-Set-Context elem)
  (DD-Display-Values elem)
  (if (DD-Has-GL elem) (DD-Eval-GL-Step elem is-back)))

(de DD-Display-Values (elem)                                         DD-Display-Values
  (cond
   ((eq (cadr elem) 'func) ; cas de l'entrée ou la sortie des fonctions
    (cond
     ((eq (car elem) 'in)  ; entrée dans une fonction : on remplace l'affichage existant par un nouveau
      (xSetValues (get DD-Root-Window 'label-func) "label" (strcat  "Entrée dans " (3 elem)))
      (mapc (get DD-Root-Window 'data-labels) 'xRemoveWidget)
      (remprop DD-Root-Window 'data-labels)
      (put DD-Root-Window 'disp-func (3 elem))
      (let ((Params (car (fval (3 elem)))) (Values (4 elem)))
   (when Params
     (put DD-Root-Window 'data-labels
         `(,(setq tmp (xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data)  "label" (car Params)))
                ,(xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data)
```

```
                                        "label" (strcat (car Values)) "treeParent" tmp)
                    ,@(get DD-Root-Window 'data-labels)))
     (put tmp 'var (car Params))
     (self (cdr Params) (cdr Values)))))
     (t ; sortie de la fonction
      (unless (eq (3 elem) (get DD-Root-Window 'disp-func))       ; sort-on de la fonction affichée ?
   (DD-Update-Fonction (3 elem)))
      (xSetValues (get DD-Root-Window 'label-func) "label" (strcat "Sortie de " (3 elem)))
      (DD-Update-Variable 'Sortie (4 elem) (3 elem)))))

   ; c'est une variable
   ((eq (cadr elem) 'var)
    (DD-Update-Variable (3 elem) (4 elem) (5 elem)))

   ; c'est une instruction
   ((eq (cadr elem) 'instr)
    (xSetValues (get DD-Root-Window 'label-instr) "label" (caddr elem)))))


; Misc....
(de DD-Add-if-Absent (stk-obj aStack)                                    DD-Add-if-Absent
  (let (stk aStack)
    (cond
     ((null stk)                    (cons stk-obj aStack))
     ((eq (3 stk-obj) (3 (car stk)))  aStack)
     (t (self (cdr stk))))))


; remise à l'ordre du jour des variables d'une fonction et recherche dans
; la pile de son dernier appel
(de DD-Update-Fonction (func)                                            DD-Update-Fonction
                                                ; on efface les variables affichées
  (mapc (get DD-Root-Window 'data-labels) 'xRemoveWidget)
  (remprop DD-Root-Window 'data-labels)
  (put DD-Root-Window 'disp-func func)
  ; on recherche le dernier accès en entrée dans la fonction
  (let ((stk (if (and (eq (caar dump-stack-bk) 'out)
                      (eq (2 (car dump-stack-bk)) 'func))
                 (cdr dump-stack-bk)
               dump-stack-bk))
    (stack nil) (depth 0))
    (cond
     ((null stk) nil)  ; pas trouvé : pb
     ; pile des appels inclus
     ((and (eq (1 (car stk)) 'out)
       (eq (2 (car stk)) 'func))
      (self (cdr stk) stack (1+ depth)))
     ((and (eq (1 (car stk)) 'in)
       (eq (2 (car stk)) 'func)
       (> depth 0))
      (self (cdr stk) stack (1- depth)))
     ((> depth 0) (self (cdr stk) stack depth))
     ; on est au niveau 0 des appels...
     ; enfin trouvé....
     ((and (eq (1 (car stk)) 'in)
       (eq (2 (car stk)) 'func)
       (eq (3 (car stk)) func))
      (DD-Update-Values (car stk)) ; maj de la fonction
      (mapc stack 'DD-Update-Values)) ; maj des valeurs
     ; variable de la fonction, on garde
     ((eq (2 (car stk)) 'var)
      (self (cdr stk) (DD-Add-if-Absent (car stk) stack) depth))
     ; si non on continue...
     (t
      (self (cdr stk) stack depth)))))


(de DD-Update-Variable (name value func)                                 DD-Update-Variable
  ; la fonction de la variable ne correspond pas à la fonction affichée :
  ; on change et lance la recherche de la bonne fonction
  (unless (eq func (get DD-Root-Window 'disp-func))
    (DD-Update-Fonction func))

  (let (WidVars (get DD-Root-Window 'data-labels))
    (cond
     ((null WidVars)                            ; non
```

```
        (put DD-Root-Window 'data-labels
         `(,(setq tmp (xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data)  "label" name))
          ,(xCreateWidget '--AL-- W-Label (get DD-Root-Window 'tree-data)
                          "label" (strcat value) "treeParent" tmp)
          ,@(get DD-Root-Window 'data-labels)))
        (put tmp 'var name))
       ((eq (get (car WidVars) 'var) name)
        (xSetValues (cadr WidVars) "label" (strcat value)))
       (t
        (self (cddr WidVars))))))

; mise à jour de la représentation graphique
(de DD-Eval-GL-Step (elem is-back)                                    DD-Eval-GL-Step
  (cond
   ((null is-back) (DD-Eval-GL-Listes (cadr (member (DD-Get-GL-Step elem) gl-run-stack))))
   (t
    (DD-Rewind-GL (cddr (setq aux (member (DD-Get-GL-Step elem) gl-run-stack)))
                  (cadr aux)))))

(de DD-Eval-GL-Listes (listes)                                        DD-Eval-GL-Listes
  (while listes
    (print (cadr listes) "->" (car listes))
    (MESAcopylist (cadr listes) (car listes))
    (nextl listes)
    (nextl listes)))

(de DD-Rewind-GL (g-stack listes)                                     DD-Rewind-GL
  (let (p-listes (DD-Seek-Prev-GL g-stack (car listes)))
    (when p-listes
      (while listes
    (if (setq aux (member (car listes) p-listes))
        (MESAcopylist (cadr p-listes) (car listes))
      (DD-Rewind-GL g-stack [(car listes)]))
    (setq listes (cddr listes))))))

(de DD-Seek-Prev-GL (g-stack aListe)                                  DD-Seek-Prev-GL
  (while (and g-stack (not (member aListe (cadr g-stack))))
     (nextl p-gl-stack)
     (nextl p-gl-stack))
   (cadr g-stack)))
```

Annexe I.15 Gestion de l'historique : Zsuivit.vlisp

```
                                        ; ****************************** ; ;
                                        ; Suivit graphique de l'animation ; ;
                                        ; ****************************** ; ;

(de Initialize-Follow-GL (obj w x y)                                  Initialize-Follow-GL
  (cond
   ((get obj 'io-follow)
    (xRemove Follow-GL)
    (remprop obj 'io-follow))
   (t
    (put obj 'io-follow t)
    (ifn (boundp 'Meta-GL) (Initialize-Meta-GL))
    (unless (and (boundp 'Follow-GL) (iswidget Follow-GL))
      (setq Follow-GL (xCreateWidget '--AS-- W-AShell "title" "Suivit graphique de l'exécution"
                                     "iconName" "Suivit graphique"))
      (let (form (xCreateWidget '--AF-- W-Form Follow-GL "background" W-BColor))
    (put Follow-GL 1 (GLwinopen "--GL--" form Meta-GL "width" 200 "height" 200))
    (put Follow-GL 2 (GLwinopen "--GL--" form Meta-GL "width" 200 "height" 200 "horizDistance" 210))
    (put Follow-GL 3 (GLwinopen "--GL--" form Meta-GL "width" 200 "height" 200 "vertDistance"  210))
    (put Follow-GL 4 (GLwinopen "--GL--" form Meta-GL "width" 200 "height" 200 "horizDistance" 210 "vert-
Distance" 210))

    (xRealize Follow-GL)
    (setq Meta-gl-windows `(,(get Follow-GL 1) ,(get Follow-GL 2) ,(get Follow-GL 3) ,(get Follow-GL 3)
,@Meta-gl-windows
    )
    (GLwinset (get Follow-GL 1))
    (GLimtranslate 0 (/ (xGetValues (get Follow-GL 1) "height") 2)
                   (/ (xGetValues (get Follow-GL 1) "width") 2))
```

```
  (GLrootlist (get Meta-GL 'root-list))
  (GLwinset (get Follow-GL 2))
  (GLimtranslate (/ (xGetValues (get Follow-GL 2) "width") 2)
                 0 (- (/ (xGetValues (get Follow-GL 2) "height") 2)))
  (GLrootlist (get Meta-GL 'root-list))
  (GLwinset (get Follow-GL 3))
  (GLimtranslate (- (/ (xGetValues (get Follow-GL 2) "width") 2))
                 (/ (xGetValues (get Follow-GL 2) "height") 2) 0)
  (GLrootlist (get Meta-GL 'root-list))
  (GLwinset (get Follow-GL 4))
  (GLrootlist (get Meta-GL 'root-list)))))))


(de Update-Follow-Root-Lists ()                                    Update-Follow-Root-Lists
  (if (boundp 'Follow-GL)
      (mapc '(1 2 3 4)
       (lambda (x)
         (GLwinset (get Follow-GL x))
         (GLrootlist (get Meta-GL 'root-list))))))


(defmacro is-sup (v0 v1) `(or (and (> ,v0 ,v1) 1) -1))                                    is-sup

(de IO-Do-Follow (from-x from-y from-z to-x to-y to-z switch)                    IO-Do-Follow
;  (print from-x from-y from-z "," to-x to-y to-z)
  (ifn (boundp 'Follow-GL) (Initialize-Follow-GL))
  ; proposition de 3 vues :
  ; vue de face
  (GLwinset (get Follow-GL 1))
  (GLlookat (- from-x 200) from-y from-z from-x from-y from-z 0 1 0)
  ; vue de haut
  (GLwinset (get Follow-GL 2))
  (GLlookat from-x (- from-y 200) from-z from-x from-y from-z 1 0 0)
  ; vue de cote
  (GLwinset (get Follow-GL 3))
  (GLlookat from-x from-y (- from-z 200) from-x from-y from-z 0 1 0)
  ; vue en subjectif
  (GLwinset (get Follow-GL 4))
  (let ((dxy (and (= (- to-x from-x) 0) (= (- to-y from-y) 0)))
   (dz  (= (- to-z from-z) 0))
   (ix (if (> from-x to-x) 100 -100))
   (iy 0)
   (iz (if (> from-z to-z) 100 -100)))
    (unless (and dxy dz)
      (GLlookat (+ from-x ix) (+ from-y iy) (+ from-z iz) (+ to-x ix) (+ to-y iy) (+ to-z iz)
                0 (if dxy 0 1) (if dxy 1 0))))
  (GLwinset Meta-GL))

                                                ; ***************************** ; ;
                                                ; Suivit Textuel de l'exécution ; ;
                                                ; ***************************** ; ;


(de Create-Stepper-Widget ()                                    Create-Stepper-Widget
  (unless (boundp 'Steper-Widgets)
    (let (root (xCreateWidget '--AS-- W-AShell "title" "Suivit de l'execution" "iconName" "Suivit"))
      (let (bx (xCreateWidget '--BX-- "awBox" root))
        (setq Steper-Widgets nil)
        (newl Steper-Widgets (xCreateWidget '--AC-- "awCommand" bx "label" "Boucle d'inspection"))
        (newl Steper-Widgets (xCreateWidget '--AC-- "awCommand" bx "label" "Continue"))
        (newl Steper-Widgets (xCreateWidget '--AC-- "awLabel"   bx "label" "                    "))
        (newl Steper-Widgets (xCreateWidget '--AC-- "awLabel"   bx "label" "                    "))
        (newl Steper-Widgets root)))))

(de Z-Step-Label (obj label)                                    Z-Step-Label
  (when (get MPV-Root-Window 'io-text)
    (xSetValues (3 Steper-Widgets) "label" (strcat label))
    (xSetValues (2 Steper-Widgets) "label" (strcat obj))))

(de Z-Step-In (obj label)                                    Z-Step-In
  (unless (get MPV-Root-Window 'io-text) (put MPV-Root-Window 'io-text t))
  (unless (xIsRealized (1 Steper-Widgets)) (xRealizeWaiting (1 Steper-Widgets)))
  (Z-Step-Label obj label)
  (let (brk)
    (cond
     (brk t)
```

```
    ((xCheckEvent (4 Steper-Widgets) "ButtonPress") (self t))
    ((xCheckEvent (5 Steper-Widgets) "ButtonPress") (break) (self nil))
    (t (self nil)))))
```

Annexe I.16 Integration des analogies (1) : Zfile.vlisp

```
                                        ; ************************************ ; ;
                                        ; Chargement via un fichier de données ; ;
                                        ; ************************************ ; ;


; chargement d'un fichier lisp
(de Load-Lisp-File (obj w)                                                    Load-Lisp-File
   (xGetFileName default-lisp-directory 'Read-Lisp-File))

(de Read-Lisp-File (file crfl)                                                Read-Lisp-File
  (MPV-Message (strcat file))
  (ifn crfl (careful t))
  (eval `(include ,file))
  (ifn crfl (careful nil))
  (MPV-Reset-Label)
  t)

(defmacro s-Value (x) `(implode (explode ,x)))                                    s-Value
(defmacro S-Cdr-Value (x n) `(implode (explode (strcdr ,x ,n))))                  S-Cdr-Value

                                        ; *********** ; ;
                                        ; OBJECT META ; ;
                                        ; *********** ; ;

                                        ; ******* ; ;
                                        ; Lecture ; ;
                                        ; ******* ; ;

(de Set-Load-Directory (obj w)                                                Set-Load-Directory
   (xCreateDialogBox "Directory de lecture par defaut" default-data-directory
                '(setq default-data-directory xDialogValue)))

(de Get-File-Name (obj w)                                                     Get-File-Name
  (setq old-obj obj)
  (xGetFileName default-data-directory 'Read-File))

(de Reload-Source (obj w)                                                     Reload-Source
  (setq old-obj obj)
  (if (get obj 'source-file) (Read-File (get obj 'source-file))))

(de Read-File (file is-from-file is-lisp-file)                                Read-File
  (setq files-to-read nil files-to-include nil)
  (MPV-Message (strcat "Begining to Read..." file))
  (setq root-object nil included-files nil)
  (letf (EOF () (print "Prematuraly Reached EOF"))
   (input file)
   (setq aux (read))
   (input))
  (setq Z-Root-File file)
  (remprop MPV-Root-Window 'new-root)
  (cond
   ((null aux) (print "Null file ?" file))
   ((listp aux) (eval ['include file]))
   (t
    (letf (EOF () (End-Read-File
                (input)
                (mapc root-object
                        (lambda (r-obj)
                          (put (GetObjectByName r-obj) 'source-file file)
                          (put (GetObjectByName r-obj) 'meta-files files-to-read)
                          (put (GetObjectByName r-obj) 'included-files included-files)
                          (put (GetObjectByName r-obj) 'ROOT t)))
                (put MPV-Root-Window 'new-root root-object)
```

```
                (MPV-Reset-Label)
                (mapc files-to-read
                        (lambda (x)
                          (MPV-Message (strcat "Reading Included File" x))
                          (if files-to-read (setq files-to-read nil))
                          (Read-File x t)))))
    (input file)
    (escape End-Read-File (Parse-Objects-File aux)))))
  (MPV-Init-Root (reverse (get MPV-Root-Window 'new-root)))
  (MPV-Reset-Label))

(de Parse-Objects-File (f-word obj)                                    Parse-Objects-File
  (cond
   ((null f-word) t)
   ((eq f-word '#) (setq f-read (read)) (while (not (eq f-read '#)) (setq f-read (read))))
   ((eq f-word 'ROOT-OBJECT)   (newl root-object (read)))
   ((eq f-word 'OBJECT)
    (let (name (read))
      (if (setq obj (GetObjectByName name nil t))
      (if (and root-object (not (member (get obj 'root-object) root-object)))
         (print "WARNING : objet " name " utilisé par " root-object " et par " (get obj 'root-object)))
   (setq Objects-List (cons (setq obj (gensym)) Objects-List)))
      (rplacd obj nil)                          ; clear des valeurs precedentes ...
      (if root-object (put obj 'root-object (last root-object)))
      (put obj 'name name)))
   ((eq f-word 'READ-FILE)      (setq files-to-read (cons (read) files-to-read)))
   ((eq f-word 'GL-FUNCTION) (setq ME-GL-Function (read)))
   ((eq f-word 'INCLUDE)
    (let (f-file (read))
      (MPV-Message (strcat f-file))
      (setq included-files (cons f-file included-files))
      (letf (EOF () (End-Read-Include))
       (escape End-Read-Include (eval `(include ,f-file))))))
   (obj
    (cond
     ((eq f-word 'O-IN)      (put obj 'o-in (read)))
     ((eq f-word 'O-OUT)     (put obj 'o-out (read)))
     ((eq f-word 'I-IN)      (put obj 'i-in (read)))
     ((eq f-word 'I-OUT)     (put obj 'i-out (read)))
     ((eq f-word 'V-OBJECT)  (put obj 'v-obj (read)))
     ((eq f-word '#) (setq f-read (read)) (while (not (eq f-read '#)) (setq f-read (read))))
     ((eq f-word 'READ-FILE)      (setq files-to-read (cons (read) files-to-read)))
     ((eq f-word 'INCLUDE)
      (let (f-file (read))
    (MPV-Message (strcat f-file))
    (setq included-files (cons f-file included-files))
    (letf (EOF () (End-Read-Include))
         (escape End-Read-Include (eval `(include ,f-file))))))
     ((eq f-word 'ROOT-OBJECT)   (newl root-object (read)))
     ((eq f-word 'TYPE)          (put obj 'type  (read)))
     ((eq f-word 'DESCRIPTION)   (put obj 'descr (read)))
     ((eq f-word 'INFLUENCE)     (put obj 'variable (read)))
     ((eq f-word 'COMPOSITION)   (put MPV-Root-Window 'composition (read)))
     ((eq f-word 'UNIQUE)        (put obj 'unique (read)))
     ((eq f-word 'ACTION-G)      (put obj 'action-g (read)))
     ((eq f-word 'MV-CONSTRAINT) (put obj 'mv-constraint (read)))
     ((eq f-word 'FUNCTION-ROOT) (Read-Function obj (read)))
     ((eq f-word 'GL-RECONSTRUCTION) (put obj 'gl-recons (read)))
     ((eq f-word 'GL-DATA)       (Read-Until-And-Put obj 'gl-data 'END-GL-DATA (read)))
     ((eq f-word 'GL-DYNAMIQUE-IN)(Read-Until-And-Put obj 'gl-dyn-in  'END-GL-DATA (read)))
     ((eq f-word 'GL-DYNAMIQUE-OUT)(Read-Until-And-Put obj 'gl-dyn-out  'END-GL-DATA (read)))
     ((eq f-word 'GENERATION)     (Read-Generation (get obj 'generation)))
     ((eq f-word 'GL-TRANSFORMATION)
      (ifn (get obj 'gl-trans) (put obj 'gl-trans (gensym)))
      (let (f-w1 (read))
    (while (not (eq (setq f-w1 (Read-Trans (get obj 'gl-trans) f-w1)) 'END-TRANSFORMATIONS)))))
      (t
       (print "Mot inconnu" f-word "dans l'objet" (if obj (get obj 'name) "Aucun"))))))
  (Parse-Objects-File (read) obj))

(defmacro Add-Activity (func act val)                                  Add-Activity
  `(PutFuncData ,func ,act (cons val (GetFuncData ,func ,act))))

(de Read-Function (obj func mot-lut)                                   Read-Function
    (put obj 'func func)
    (let (c-func)
```

```
    (setq mot-lut (read))
    (cond
     ((eq mot-lut '#) (setq mot-lut (read)) (while (not (eq mot-lut '#)) (setq mot-lut (read))))
     ((eq mot-lut 'END-FUNCTION) t)
     ((or (eq mot-lut 'FUNCTION) (eq mot-lut 'VARIABLE)) (self (read)))
     ((eq mot-lut 'FUNC-FILE)
  (let (f-file (read))
   (put obj 'func-file f-file)
   (if (= (ftyp c-func) 0)
      (letf (EOF () (End-Read-Include (terpri) (careful)))
            (careful t)
            (escape End-Read-Include (eval `(include ,f-file))))))
  (self c-func))
     ((eq mot-lut 'S-IN)      (Add-Activity c-func 's-in t)  (self c-func))
     ((eq mot-lut 'S-OUT)     (Add-Activity c-func 's-out t) (self c-func))
     ((eq mot-lut 'O-IN)      (Add-Activity c-func 'o-in (read))  (self c-func))
     ((eq mot-lut 'O-OUT)     (Add-Activity c-func 'o-out (read)) (self c-func))
     ((eq mot-lut 'I-IN)      (Add-Activity c-func 'i-in (read))  (self c-func))
     ((eq mot-lut 'I-OUT)     (Add-Activity c-func 'i-out (read)) (self c-func))
     ((eq mot-lut 'V-STOP)    (Add-Activity c-func 'v-stop t)     (self c-func))
     ((eq mot-lut 'V-OBJECT)  (Add-Activity c-func 'v-obj (read)) (self c-func))
     ((null mot-lut) (self c-func))
     (t (print "Mot inconnu dans FUNCTION" mot-lut) (self c-func)))))


(de Read-Generation (c-obj)                                                    Read-Generation
   (ifn c-obj (put obj 'generation (setq c-obj (gensym))))
   (Read-Until-And-Put c-obj 'objet      'END-OBJECTS (read))
   (Read-Until-And-Put c-obj 'parcour    'END-PARCOURS (read))
   (Read-Transformations c-obj 'transformations nil))


(de Read-Until-And-Put (obj field cds f-w)                                     Read-Until-And-Put
   (let ((w f-w) (b))
    (cond
     ((eq w '#) (setq w (read)) (while (not (eq w '#)) (setq w (read))))
     ((eq w cds)  (put obj field b))
     (t   (self (read) (append b [w])))))))

(de Read-Transformations (obj field vals)                                      Read-Transformations
   (let ((f-w (read)) (n 0))
    (cond
     ((eq f-w '#) (setq f-w (read)) (while (not (eq f-w '#)) (setq f-w (read))))
     ((eq f-w 'TRANSFORMATION)      (setq vals (append vals (cons '() '())))) (self (read) (1+ n)))
     ((eq f-w 'END-TRANSFORMATIONS) (put obj field vals))
     (t
  (setq f-w (Read-Trans (Make-Next-Transfo n (nth n vals)) f-w))
  (self f-w n)))))

(de Make-Next-Transfo (n n-obj)                                                Make-Next-Transfo
   (cond
    ((and n-obj (car n-obj)) (setq n-obj (car n-obj)))
    (n-obj                   (setq n-obj (car (rplaca n-obj (gensym)))))
    (t (setq n-obj (gensym)) (setq vals (append vals (if (null vals) [n-obj] n-obj)))))
   n-obj)


(defmacro l-put (v ind value)                                                  l-put
  `(if (setq aux (member ,ind (cdr ,v))) (rplacd aux ,value)
    (rplacd ,v (append (cdr ,v) [,ind ,value]))))

(de Read-Trans (obj f-w)                                                       Read-Trans
  (cond
   ((or (eq f-w 'TRANSLATION) (eq f-w 'T)) (l-put obj 'translate (read)) (Read-Trans obj (read)))
   ((or (eq f-w 'ROTATION) (eq f-w 'R))    (l-put obj 'rotate (read))   (Read-Trans obj (read)))
   ((or (eq f-w 'SCALE) (eq f-w 'S))       (l-put obj 'scale (read)) (Read-Trans obj (read)))
   ((or (eq f-w 'COLOR) (eq f-w 'C))       (l-put obj 'color (read)) (Read-Trans obj (read)))
   (t f-w)))


                                    ; ******** ; ;
                                    ; Ecriture ; ;
                                    ; ******** ; ;

; sauvegarde des donnees
```

```
(de Save-All-Objects (obj w x y)                                    Save-All-Objects
  (setq root-object (get obj 'root-object))
  (xGetFileName () 'Write-To-File))

(de Write-To-File (file)                                            Write-To-File
   (cond
    ((null file) t)
    ((probef file)
     (Yes-No-Menu "Effacement (oui) ou rajout (non) ?"
                  `(Del-And-Write ',file)
                  `(Write-File    ',file)))
    (t (Write-File file))))

(de Del-And-Write (file)                                            Del-And-Write
  (sh (strcat "rm " file))
  (Write-File file))

(de Write-File (file)                                               Write-File
   (output file)
   (print "ROOT-OBJECT" root-object)
   (mapc Objects-List
    (lambda (obj)
      (if (eq (get obj 'type) 'gl-root) (Print-GL-root-Object obj)
        (Print-C-GL-Object obj))))
   (output))

(de Print-GL-root-Object (obj)                                      Print-GL-root-Object
   (terpri)
   (print "OBJECT " (get obj 'name))
   (print "TYPE     gl-root")
   (if (get MPV-Root-Window 'composition) (print "COMPOSITION t"))
   (Print-GL-Data obj))

(de Print-GL-Data (obj is-dyn)                                      Print-GL-Data
   (print (cond ((null is-dyn)   "GL-DATA")
                ((eq is-dyn 'in) "GL-DYNAMIQUE-IN")
                (t               "GL-DYNAMIQUE-OUT")))
   (mapc (get obj (if (null is-dyn) 'gl-data (if (eq is-dyn 'in) 'gl-dyn-in 'gl-dyn-out))) 'print)
   (print "END-GL-DATA"))

(de Print-C-GL-Object (obj)                                         Print-C-GL-Object
   (print "OBJECT " (get obj 'name))
   (print "TYPE "   (get obj 'type))
   (if (get obj 'descr)     (print "DESCRIPTION"     (get obj 'descr)))
   (if (get obj 'variable)  (print "INFLUENCE"       (get obj 'variable)))
   (if (get obj 'unique)    (print "UNIQUE"          (get obj 'unique)))
   (if (get obj 'action-g)  (print "ACTION-G"        (get obj 'action-g)))
   (if (get obj 'func)      (Print-Function-Data obj (get obj 'func)))
   (if (get obj 'gl-recons) (print "GL-RECONSTRUCTION" (get obj 'reconstruct)))
   (if (get obj 'gl-data)   (Print-GL-Data obj nil))
   (if (get obj 'gl-dyn)    (Print-GL-Data obj t))
   (if (get obj 'gl-trans)
     (progn
      (print "GL-TRANSFORMATION")  (Print-Transformation (get obj 'gl-trans))
      (print "END-TRANSFORMATIONS")))
   (if (get obj 'generation)
     (progn
      (print "GENERATION")
      (mapc (get (get obj 'generation) 'objet) 'print) (print "END-OBJECTS")
      (mapc (get (get obj 'generation) 'parcour) 'print) (print "END-PARCOURS")
      (mapc (get (get obj 'generation) 'transformations) 'Print-Gen-Trans) (print "END-
TRANSFORMATIONS"))))

(de Print-Function-Data (obj func)                                  Print-Function-Data
   (ifn (get obj 'FDC) nil
   (print "FUNCTION-ROOT" func)
   (setq done (car (get obj 'FDC)))
   (Print-A-Function (car (get obj 'FDC)) t)
   (let (fdc (cdr (get obj 'FDC)))
        (cond
              ((null fdc) t)
              ((and (atom (car fdc)) (not (member (car fdc) done)))
               (Print-A-Function (car fdc))
               (setq done `(,(car fdc) ,@done))
               (self  (cdr fdc)))
              ((listp (car fdc))
```

```
                (self (car fdc))
                (self (cdr fdc)))
               (t (self (cdr fdc)))))
   (print "END-FUNCTION")))

(defmacro Print-Activity (func act label)                                    Print-Activity
  `(mapc (GetFuncData ,func ,act) (lambda (x) (print ,label x))))

(de Print-A-Function (func is-first)                                         Print-A-Function
   (let (f-putted)
    (if (or is-first
       (GetFuncData func 's-in) (GetFuncData func 's-out)
       (GetFuncData func 'o-in) (GetFuncData func 'o-out)
       (GetFuncData func 'i-in) (GetFuncData func 'i-out)
       (GetFuncData func 'v-in) (GetFuncData func 'v-out))
     (progn
      (print "FUNCTION" func)
      (print "FUNC-FILE" (car (getival func 20)))
      (if (GetFuncData func 's-in)  (print "S-IN"))
      (if (GetFuncData func 's-out) (print "S-OUT"))
      (if (GetFuncData func 'o-in)  (Print-Activity func 'o-in "O-IN"))
      (if (GetFuncData func 'o-out) (Print-Activity func 'o-out "O-OUT"))
      (if (GetFuncData func 'i-in)  (Print-Activity func 'o-in "I-IN"))
      (if (GetFuncData func 'i-out) (Print-Activity func 'o-out "I-OUT"))
      (setq f-putted t)))
     (mapc (append (%find (ival func) 0) (%find (ival func) 1))
      (lambda (var)
       (if (or (GetVarData var 'v-stop) (GetVarData var 'v-obj))
                (progn
                 (ifn f-putted
                          (progn
                           (print "FUNCTION" func)
                           (print "FUNC-FILE" (car (getival func 20)))
                           (setq f-putted t)))
                 (print "VARIABLE" var)
                 (if (GetVarData var 'v-stop) (print "V-STOP"))
                 (if (GetVarData var 'v-obj) (print "V-OBJECT" (GetVarData var 'v-obj)))))))))

(de Print-Gen-Trans (obj)                                                    Print-Gen-Trans
   (print "TRANSFORMATION")
   (Print-Transformation obj))

(de Print-Transformation (obj)                                               Print-Transformation
   (if (get obj 'translate) (print "TRANSLATION" (get obj 'translate)))
   (if (get obj 'rotate)   (print "ROTATION"    (get obj 'rotate)))
   (if (get obj 'scale) (print "SCALE"       (get obj 'scale)))
   (if (get obj 'color) (print "COLOR"       (get obj 'color))))

(de slprint (lst)                                                            slprint
   (ifn lst t
   (princ (strcat "\"" (car lst) "\""))
   (slprint (cdr lst))))
```

Annexe I.17 Integration des analogies (2) : Zuser.vlisp

```
; Définition des fonctions du système Zeugma accessibles aux utilisateurs :

; La définition d'objets : define-Z
; la consultation de la position d'objets graphiques : position-Z
; inclusion de fichiers lisp (en mode careful) : include-Z

; Syntaxe :
;
; (define-Z nom-objet
;
;              Données non spécifiques
;
;   (gl-recons [t|nil])              : reconstruction des instructions graphiques (oui/non)
```

```
;   (gl-data  gl-instr1 ... gl-instrn) : instructions graphiques de l'objet
;
;     (generation ((nom-de-l-objet1 condition1 transformation1) ... (nom-de-l-objetn conditionn transformationn)))
;           : objets generes : condition = condition de generation,
;                   : transformation = transformation graphique (syntaxe id a gl-transfo)
;
;           Données spécifiques aux objets racine
;   (titre label)               : titre de la représentation analogique
;   (root [t|nil])              : objet racine
;   (func function-name)        : fonction liée [r]
;   (descr fichier-de-description)   : desccription de l'objet [r]
;   (analyse type-d-analyse)    : type d'analyse effectuée sur le programme [r]
;   (variable nom-de-variable)  : variable point de départ pour les influences [r]
;   (composition [t|nil])       : parcour de la composition possible
;   (unique [t|nil])            : dessin unique des éléments
;   (fdd-tous-couples           : dessin de tous les couples variable - fonction
;
;           Données spécifiques aux objets "dynamiques"
;   (dynamic (i-in  [t|nil])    : objet en entrée de toute instruction
;         (i-out [t|nil])       : objet en sortie de toute instruction
;         (o-in  [t|nil])       : objet en entrée de toute fonction
;         (o-out [t|nil])       : objet en sortie de toute fonction
;         (v-obj [local|global])) : objet en accès local ou global des variables
;   (action-g action)           : type d'action lors de l'exécution
;   (mv-constraint c-x c-y c-z)     : contraintes de déplacement multiples à l'exécution
;                       : c-x, c-y et c-z sont des fonctions
;   (gl-dyn-[in|out]  gl-instr1 ... gl-instrn) : instructions graphiques de l'objet exécutés lors de l'exécution
;   (gl-transfo (translate X Y Z)
;           (rotate   X Y Z)
;           (scale    X Y Z W)
;           (color    R G B))   : transformations graphiques dynamiques
; )
;
; (include-Z nom-de-fichier     : fichier source du programme
;   (<nom-de-fonction>          : propriétés pour nom-de-fonction
;     (s-in [t|nil])            : arret en entrée
;     (s-out [t|nil])           : arret en sortie
;     (o-in  objet)             : objet à exécuter en entrée de la fonction
;     (o-out objet)             : objet à exécuter en sortie de la fonction
;     (i-in  objet)             : objet à exécuter en entrée dans les instructions de la fonction
;     (i-out objet)             : objet à exécuter en sortie dans les instructions de la fonction
;     (v-obj objet)))           : objet à exécuter en accès aux variables de la fonction

(setq Z-Root-Obj nil   ; ORS racine
    Z-Root-File nil) ; fichier de définition de l'analogie

; cette fonction doit être appellée à la fin de chaque fichier inclu :
; elle met à jour l'interface de Zeugma

; Définition d'objets : define-Z <nom-objet>

(deff define-Z (args)                                                    define-Z
  (let (obj (cond
       ((null (car args)) nil)
       ((setq aux (GetObjectByName (car args) nil t)) aux)
       (t (setq Objects-List (cons (setq aux (gensym)) Objects-List))
              (put aux 'name (car args))
              aux)))
    (when obj
     (let (values (cdr args))
       (when values
                (cond
                 ((not (listp (car values))) (print "Warning: define-Z" (get obj 'name) "instruction" (car args)))
                 ((and (eq (caar values) 'root) (cadar values))
                  (put obj 'ROOT t)
                  (put MPV-Root-Window 'new-root (cons obj (get MPV-Root-Window 'new-root)))
                  (if Z-Root-File (put obj 'source-file Z-Root-File))
                  (if exe-dynamic-dump (DD-Dump-Root-Object obj)))
                 ((eq (caar values) 'titre) (MPV-Reset-Label (strcat (cadar values))))
                 ((eq (caar values) 'dynamic)
                  (mapc (cdar values)
                        (lambda (x)
                          (put obj (car x) (cadr x)))))
                 ((member (caar values) '(func descr analyse variable
                                                composition unique action-g gl-recons))
```

```
                    (put obj (caar values) (cadar values)))
                   ((member (caar values) '(gl-data gl-dyn-in gl-dyn-out))
                    (put obj (caar values) (cdar values)))
                   ((eq (caar values) 'gl-transfo)
                    (put obj (caar values) (if (setq aux (get obj (caar values))) aux (setq aux (gensym))))
                    (ZU-Make-Transfo aux (cdar values)))
                   ((eq (caar values) 'generation)
                    (put obj (caar values) (if (setq aux (get obj (caar values))) aux (setq aux (gensym))))
                    (rplacd aux nil)    ; clear des anciennes valeurs
                    (mapc (cdar values)
                          (lambda (x)
                            (put aux 'objet (if (get aux 'objet) (append (get aux 'objet) (car x)) [(car x)]))
                            (put aux 'parcour (if (get aux 'parcour) (append (get aux 'parcour) [(cadr x)]) [(cadr
x)]))
                            (put aux 'transformations (if (get aux 'transformations)
                                                          (append (get aux 'transformations) (setq
aux1 (gensym)))
                                                          [(setq aux1 (gensym))]))
                            (ZU-Make-Transfo aux1 (cddr x)))))
                    ((eq (caar values) 'fdd-tous-couples) (put MPV-Root-Window 'fdd-all t)))
                   (self (cdr values)))))))

(de ZU-Make-Transfo (t-obj trans)                                          ZU-Make-Transfo
  (ifn trans nil
       (put t-obj (caar trans) (cadar trans))
       (ZU-Make-Transfo t-obj (cdr trans))))

; définition de include-Z

(deff include-Z (args)                                                     include-Z
  (let ((func-file (nextl args))
    (crfl careful))
    (careful t)
    (eval `(include ,(strcat func-file)))
    (careful crfl)
    (ZU-Make-Funcs args)))

(de ZU-Make-Funcs (values)                                                 ZU-Make-Funcs
  (ifn values nil
       (PutFuncData (caar values) 'func-file func-file)
       (ZU-Make-Props (caar values) (cdar values))
       (ZU-Make-Funcs (cdr values))))

(de ZU-Make-Props (func props)                                             ZU-Make-Props
  (ifn props nil
       (PutFuncData func (caar props) (if (setq aux (GetFuncData func (caar props)))
                                          (cons (GetFuncData func (caar props)) (cadar props))
                                          [(cadar props)]))
       (ZU-Make-Props func (cdr props))))

; sauvegarde des données

(de Z-Save-Definitions (obj w x y)                                         Z-Save-Definitions
  (MPV-Message "Sauvegarde des objets dans Z-Save.vlisp")
  (output "Z-Save.vlisp")
  (mapc Objects-List 'Z-Save))

(de Z-Save (obj)                                                           Z-Save
  (print "(define-Z" (get obj 'name))
  (and (get obj 'ROOT) (print "  (root t)")))

                                        ; ********************************************************* ; ;
                                        ; fonctions d'interface entre Zeugma et le dessin des objets ; ;
                                        ; ********************************************************* ; ;

; retour du type de la fonction suivant les statistiques numériques
(defmacro Z-Get-Type (vals)                                                Z-Get-Type
  (if vals
      `(or (and (get ,vals 'loop) 'boucles)
      (and (get ,vals 'lambda) 'lambda)
```

```
    (and (or (get ,vals 'conds) (get ,vals 'test)) 'test)
    (and (or (get ,vals 'io) (get ,vals 'x) (get ,vals 'gl)) 'entrees-sorties)
    (and (get ,vals 'num) 'numeriques)
    (and (get ,vals 'str) 'chaines)
    (and (get ,vals 'lst) 'listes)
    'autres)
  '(or (and com-loop 'boucles)
   (and com-lambda 'lambda)
   (and (or com-conds com-test) 'test)
   (and (or com-io com-x com-gl) 'entrees-sorties)
   (and com-num 'numeriques)
   (and com-str 'chaines)
   (and com-lst 'listes)
   'autres)))


                                        ; recuperation de la position graphique d'un objet lisp répresenté
; obj = l'objet : une instruction, une fonction ou une variable
(de Z-Get-Position (obj aPrc)                                          Z-Get-Position
  (let (info (Z-Get-Info obj aPrc))
    (if info
   (mapcar '(1 2 3) (lambda (x) (Mdiv (x (car (GL-Get-Transfo-Values info))) scale-factor))))))

(de Z-Get-X (obj aPrc)                                                 Z-Get-X
  (if (setq aux (Z-Get-Info obj aPrc)) (Mdiv (1 (car (GL-Get-Transfo-Values aux))) scale-factor)))

(de Z-Get-Y (obj aPrc)                                                 Z-Get-Y
  (if (setq aux (Z-Get-Info obj aPrc)) (Mdiv (2 (car (GL-Get-Transfo-Values aux))) scale-factor)))

(de Z-Get-Z (obj aPrc)                                                 Z-Get-Z
  (if (setq aux (Z-Get-Info obj aPrc)) (Mdiv (3 (car (GL-Get-Transfo-Values aux))) scale-factor)))


(de Z-Get-Info (body aPrc)                                             Z-Get-Info
  (cond
   ((setq aux (car (if aPrc (GI-Get-GL body aPrc) (GI-Get-a-GL body)))) aux)
   (body
    (or (and (atom body) (listp (fval body)) (Z-Get-Info (cdr (fval body))))
   (and (listp (car body)) (Z-Get-Info (car body)))
   (Z-Get-Info (cdr body))))))


; peut etre appelle lors de l'execution pour remetre a zero les transformations graphiques
; lors de l'ajout de graphique a un dessin.....
(de Z-Undo-Transformations ()                                          Z-Undo-Transformations
  (when (setq aux (car ***gl-object***))
    (while aux
      (mapc (GL-Get-Transfo-Values aux)
       (lambda (x)
         (cond
          ((eq (car x) 'r) (GLrotate (- (2 x)) (- (3 x)) (- (4 x))))
          ((eq (car x) 's) (GLscale (/ 1.0 (2 x)) (/ 1.0 (3 x)) (/ 1.0 (4 x)) (/ 1.0 (5 x))))
          ((eq (car x) 't) (GLtranslate (- (2 x)) (- (3 x)) (- (4 x)))))))
      (setq aux (GL-Get-Next aux)))))

(de Z-Get-Func-Analyses (func) (cddr (GetPrcData 'COMP func 'COMPOSITION)))  Z-Get-Func-Analyses
(de Z-Fonction-Racine (func) (eq func (GetFuncData func 'fonction-racine)))  Z-Fonction-Racine
```

Annexe I.18 Aide en ligne de Zeugma : Zhelp.vlisp

```
; fonctions d'aides

(defmacro Z-Help (section topic)  (xaide topic section Zeugma-Help-File))                Z-Help

; pour le menu principal
(de MPV-Help ()         (Z-Help "Interface principale" "Menu-principal"))                MPV-Help

(de MPV-Help-new-root () (Z-Help "Interface principale" "Nouvelle-représentation"))      MPV-Help-new-root

(de Root-ORS-Help ()     (Z-Help "Interface principale" "ORS-Racine"))                   Root-ORS-Help

(de ORS-Help ()         (Z-Help "Interface principale" "ORS-Spécification"))             ORS-Help

(de Link-Help ()        (Z-Help "Interface principale" "Attachements"))                  Link-Help
```

```
(de Trans-Help ()       (Z-Help "Interface principale" "Transformations-graphiques"))       Trans-Help

(de FM-Help ()          (Z-Help "Fonctions et Variables" "Fonction"))                       FM-Help
(de FM-Func-Help ()     (Z-Help "Fonctions et Variables" "Fonction-Attachements"))          FM-Func-Help
(de FM-Func-Help ()     (Z-Help "Fonctions et Variables" "Variable-Attachements"))          FM-Func-Help
```

Annexe II Fichiers source des définitions des analogies

Annexe II.1 Définitions de l'analogie « des programmes comme des villes »

```
; fichier contenant les définitions graphiques pour la représentation par citées
; (include "/usr/people/damien/work/Meta-Donnees/SV/maisons/cityL.vlisp")

; Inclusion du fichier source de la fonction exemple
 (include-Z "/usr/people/damien/work/lisp-expls/graph.vlisp" (rechercher))


                                        ; ********************************************** ; ;
                                        ; Objets directement liés à des opérations graphiques ; ;
                                        ; ********************************************** ; ;

(define-Z cube-fdf      ORS cube-fdf
   (gl-data (dessine-cube-vide 100)))
(define-Z toit-plein    ORS toit-plein
  (gl-data (dessine-toit-plein (Z-Get-Func-Analyses flow))) (gl-recons t))
(define-Z jardin        ORS jardin
  (gl-data (dessine-jardin)))
(define-Z chemin        ORS chemin
  (gl-data (dessine-chemin)))

; ************************* ; ;
; Partie liee a l execution ; ;
; ************************* ; ;

(define-Z man           ORS man
  (dynamic (i-in t) (o-in t))
  (gl-data (dessine-homme))
  (action-g mv-uniq)
  (mv-constraint (is-path-x is-path-y is-path-z)))

; ********************** ; ;
; traitement des atomiques ; ;
; ********************** ; ;

(define-Z traite-if-primitive       ORS traite-if-primitive
  (gl-data (dessine-primitive com-instr)) (gl-recons t))
(define-Z traite-loop-primitive     ORS traite-loop-primitive
  (gl-data (dessine-primitive com-instr)) (gl-recons t))
(define-Z traite-lambda-primitive           ORS traite-lambda-primitive
  (gl-data (dessine-primitive com-instr)) (gl-recons t))
(define-Z traite-subr-primitive     ORS traite-subr-primitive
 (gl-data (dessine-primitive com-instr)) (gl-recons t))

(define-Z traite-string  ORS traite-string
  (gl-data (dessine-chaine com-instr)) (gl-recons t))
(define-Z traite-nombre          ORS traite-nombre
  (gl-data (dessine-nombre com-instr)) (gl-recons t))
(define-Z traite-atom   ORS traite-atom
  (gl-data (dessine-atom com-instr)) (gl-recons t))

; traitement des lambda-expressions ; ;

(define-Z traite-lambda          ORS traite-lambda
  (generation
     (cube-fdf incond (translate (50 50 50)))
     (traite-lambda-primitive com-ATOM (translate (5 35 50)))
     (traite-lambda-body com-CDR (translate (0 0 100)))))

(define-Z traite-lambda-body     ORS traite-lambda-body
 (generation   (instruction-Lisp-Z com-CAR)
        (instruction-Lisp-Z com-CDR (translate (0 0 100)))))

(define-Z traite-let     ORS traite-let
  (generation
     (cube-fdf incond (translate (50 50 50)))
     (traite-lambda-primitive com-ATOM (translate (5 35 50)))
```

```
    (traite-let-vars com-CDR (translate (0 0 100)))))

(define-Z traite-let-vars          ORS traite-let-vars
 (generation
  (let-une-variable (test . (atom car-flow)))
  (let-multi-variables (test . (listp car-flow)) (scale (next-nilg next-nilg 1 next-nilg)))
  (instruction-Lisp-Z com-CDR (translate (0 0 100)))))

(define-Z let-une-variable         ORS let-une-variable
  (generation
   (traite-atom com-CAR (translate (5 35 50)))
   (instruction-Lisp com-CDR)))

(define-Z let-multi-variables      ORS let-multi-variables
  (generation
   (let-une-variable com-CAR)
   (let-multi-variables com-CDR (translate (0 0 100)))))

(define-Z traite-map   ORS traite-map
  (generation  (cube-fdf incond (translate (50 50 50)))
        (traite-lambda-primitive com-ATOM (translate (5 35 50)))
        (map-argument com-CDR (translate (0 0 -100)))))

(define-Z map-argument          ORS map-argument
 (generation  (instruction-Lisp com-CAR)
       (instruction-Lisp-Z com-CDR (translate (0 0 100)))))

; traitement des boucles ; ;

(define-Z traite-repeat            ORS traite-repeat
  (generation  (cube-fdf incond (translate (50 50 50)))
        (traite-loop-primitive com-ATOM (translate (5 35 50)))
        (repeat-corps com-CDR)))

(define-Z repeat-corps             ORS repeat-corps
  (generation
   (traite-nombre com-NUMBP (translate ((- 50 (/ (rem car-flow 100) 2))
                                        (- 50 (/ (rem car-flow 100) 2))
                                        (- 50 (/ (rem car-flow 100) 2)))))
   (instruction-Lisp com-CDR)))

(define-Z traite-while  ORS traite-while
  (generation
   (cube-fdf incond (translate (50 50 50)))
   (traite-loop-primitive com-ATOM (translate (5 35 50)))
   (instruction-Lisp-Z com-CDR (translate (0 0 100)))))

; traitement des conditions ; ;

(define-Z traite-cond  ORS traite-cond
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-if-primitive com-ATOM (translate (5 35 -50)) (rotate (0 -900 0)))
  (cond-exprs com-CDR (translate (100 0 0)))))

;  (traite-if-primitive com-ATOM (translate (5 35 -50)))

(define-Z cond-exprs  ORS cond-exprs
 (generation
  (cond-expr com-CAR (scale (next-nilg 1 next-nilg next-nilg)))
  (cond-exprs com-CDR (translate (100 0 0)))))

(define-Z cond-expr    ORS cond-expr
  (generation
   (traite-test       incond      (color (255 100 0 255)))
   (instruction-Lisp-Y com-CDR     (translate (0 100 0)))))

(define-Z traite-when  ORS traite-when
  (generation
   (cube-fdf incond (translate (50 50 50)))
   (traite-if-primitive com-ATOM (translate (5 35 -50)) (rotate (0 -900 0)))
   (traite-if-exprs     com-CDR (translate (100 0 0)))))

;   (traite-if-primitive com-ATOM (translate (5 35 -50)))

(define-Z while-test    ORS while-test
```

```
 (generation
  (traite-atom com-ATOM (translate (5 35 50)))
  (traite-test com-CAR)
  (instruction-Lisp  com-CDR)))

(define-Z traite-if        ORS traite-if
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-if-primitive com-ATOM (translate (5 35 -50)) (rotate (0 -900 0)))
  (traite-if-exprs     com-CDR (translate (100 0 0)))))

 (define-Z traite-if-exprs            ORS traite-if-exprs
  (generation
   (traite-test       incond      (color (255 100 0 255)))
   (instruction-Lisp-X com-CDR     (translate (100 0 0)))))

(define-Z traite-test     ORS traite-test
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-atom       com-ATOM (translate (5 35 50)))
  (traite-nombre      com-NUMBP (translate ((- 50 (/ (rem car-flow 100) 2))
                                            (- 50 (/ (rem car-flow 100) 2))
                                            (- 50 (/ (rem car-flow 100) 2)))))
  (traite-string      com-STRINGP (translate (5 35 50)))
  (instruction-Lisp-Y com-CAR (scale (next-nilg 1 next-nilg next-nilg)))))

; traitement des autres primitives (par type de SUBR) ; ;

(define-Z traite-subr0 ORS traite-subr0
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-subr-primitive com-ATOM (translate (5 35 50)))))

(define-Z traite-subr-1-2-3         ORS traite-subr-1-2-3
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-subr-primitive com-ATOM (translate (5 35 50)))
  (instruction-Lisp-Y com-CDR (translate (0 100 0)))))

(define-Z traite-nsubr ORS traite-nsubr
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-subr-primitive com-ATOM (translate (5 35 50)))
  (instruction-Lisp-Y com-CDR (translate (0 100 0)))))

; traitement des autres expressions ; ;

(define-Z normal-fexpr              ORS normal-fexpr
 (generation
  (cube-fdf incond (translate (50 50 50)))
  (traite-atom com-ATOM (translate (5 35 50)))
  (instruction-Lisp-Y com-CDR (translate (0 100 0)))))

(define-Z traite-quote ORS traite-quote
  (generation
   (cube-fdf incond (translate (50 50 50)))
   (traite-quote com-CDR (translate (0 100 0)))))

; branchement général suivant le type d'expression ; ;

(define-Z instruction-Lisp          ORS instruction-Lisp
 (generation
  (traite-atom       com-ATOM        (translate (5 35 50)))
  (traite-nombre      com-NUMBP       (translate ((- 50 (/ (rem car-flow 100) 2))
                                                  (- 50 (/ (rem car-flow 100) 2))
                                                  (- 50 (/ (rem car-flow 100) 2)))))
  (traite-string     com-STRINGP     (translate (5 35 50)))
  (traite-if        (i-com . if)     (scale (1 next-nilg next-nilg next-nilg)) (color (255 100 100 255)))
  (traite-if        (i-com . ifn)    (scale (1 next-nilg next-nilg next-nilg)) (color (255 100 100 255)))
  (traite-cond       (i-com . cond)   (scale (1 next-nilg next-nilg next-nilg)) (color (255 100 100 255)))
  (traite-when       (i-com . when)   (scale (1 next-nilg next-nilg next-nilg)) (color (255 100 100 255)))
  (traite-when       (i-com . unless) (scale (1 next-nilg next-nilg next-nilg)) (color (255 100 100 255)))
  (traite-while      (i-com . while)  (scale (next-nilg next-nilg 1 next-nilg)) (color (255   0 100 255)))
  (traite-while      (i-com . until)  (scale (next-nilg next-nilg 1 next-nilg)) (color (255   0 100 255)))
  (traite-map        (i-com . mapc)   (scale (next-nilg next-nilg 1 next-nilg)) (color (255   0 100 255)))
  (traite-map        (i-com . mapcar) (scale (next-nilg next-nilg 1 next-nilg)) (color (255   0 100 255)))
```

```
  (traite-lambda     (i-com . lambda) (scale (next-nilg next-nilg 1 next-nilg)) (color (255 100   0 255)))
  (traite-let        (i-com . let)    (scale (next-nilg next-nilg 1 next-nilg)) (color (255 100   0 255)))
  (traite-let        (i-com . letf)   (scale (next-nilg next-nilg 1 next-nilg)) (color (255 100   0 255)))
  (traite-quote      (i-com . quote)  (scale (next-nilg 1 next-nilg next-nilg)) (color (255   0 255 255)))
  (traite-quote      (f-com . 8)      (scale (next-nilg 1 next-nilg next-nilg)) (color (255   0 255 255)))
  (normal-fexpr      com-is-num       (scale (next-nilg 1 next-nilg next-nilg)) (color (  0   0 255 255)))
  (normal-fexpr      com-is-str       (scale (next-nilg 1 next-nilg next-nilg)) (color ( 50 100 255 255)))
  (normal-fexpr      com-is-lst       (scale (next-nilg 1 next-nilg next-nilg)) (color (100  50 255 255)))
  (traite-subr0      (f-com . 1)      (scale (next-nilg 1 next-nilg next-nilg)) (color (100 100 255 255)))
  (traite-subr-1-2-3 (f-com . 2)      (scale (next-nilg 1 next-nilg next-nilg)) (color (100 100 255 255)))
  (traite-subr-1-2-3 (f-com . 3)      (scale (next-nilg 1 next-nilg next-nilg)) (color (100 100 255 255)))
  (traite-subr-1-2-3 (f-com . 4)      (scale (next-nilg 1 next-nilg next-nilg)) (color (100 100 255 255)))
  (traite-nsubr      (f-com . 5)      (scale (next-nilg 1 next-nilg next-nilg)) (color (100 100 255 255)))
  (normal-fexpr      com-no-CAR       (scale (next-nilg 1 next-nilg next-nilg)) (color (255 255 255 255)))))

;  (normal-fexpr      (t-com . Instr-Num) (scale (next-nilg 1 next-nilg next-nilg)) (color (  0   0 255 255)))
;  (normal-fexpr      (t-com . Instr-Str) (scale (next-nilg 1 next-nilg next-nilg)) (color ( 50 100 255 255)))
;  (normal-fexpr      (t-com . Instr-Lst) (scale (next-nilg 1 next-nilg next-nilg)) (color (100  50 255 255)))

; progression dans une expression suivant une direction graphique (X, Y ou Z) ; ;

(define-Z instruction-Lisp-Z       ORS instruction-Lisp-Z
 (generation
  (instruction-Lisp incond)
  (instruction-Lisp-Z com-CDR (translate (0 0 100)))))

(define-Z instruction-Lisp-Y       ORS instruction-Lisp-Y
 (generation
  (instruction-Lisp incond)
  (instruction-Lisp-Y com-CDR (translate (0 100 0)))))

(define-Z instruction-Lisp-X       ORS instruction-Lisp-X
 (generation
  (instruction-Lisp incond)
  (instruction-Lisp-X com-CDR (translate (100 0 0)))))

; objet de dessin d'une fonction ; ;

(define-Z construit-maison         ORS construit-maison
 (gl-recons t)
 (gl-data (dessine-indic))
 (generation
  (chemin     incond (color ((ColorTypeR flow) (ColorTypeG flow) (ColorTypeB flow) 255)))
  (jardin     incond (color ((or (and (Z-Fonction-Racine flow) 150) 40) 150 40 255)))
  (toit-plein incond)
  (composition composition-fonction)))

; début de l'étude de la composition à partir d'un autre flot
; dessin du cube contenant les instructions du corps de la fonction
; et lancement du dessin de celles-ci
(define-Z composition ORS composition
  (generation
   (cube-fdf   incond (color (255 255 255 255)) (translate (50 50 50)))
   (instruction-Lisp-Y incond (scale (next-nilg 1 next-nilg next-nilg)))))

; parcours du flot de contrôle ; ;

(define-Z gener           ORS gener
 (generation
  (construit-maison types-ATOM (translate ((maison-X) 0 (maison-Z))))
  (gener  types-CDR)))

;  (gener fdc-CAR)
; objet initial de la construction des maisons ; ;

(define-Z city            ORS city
  (titre "Des programmes comme des cités")
  (root t)
  (gl-data (dessine-terre) (setq block-size 400))
  (func rechercher)
  (descr "/usr/people/damien/work/Meta-Donnees/SV/maisons/maison")
  (analyse FULL)
  (composition t)
  (unique t)
  (generation (gener incond)))
```

```
; ****************************************** ; ;
; parcour du flux de dependence des variables ; ;
; ****************************************** ; ;

(define-Z data          ORS data
  (root t)
  (func rechercher)
  (descr "/usr/people/damien/work/Meta-Donnees/SV/maisons/Help/data")
  (analyse FULL)
  (composition t)
  (unique t)
  (generation
   (do-on-var fdd-VAR (translate ((var-X fdd-func) 0 (var-Z fdd-func))))
   (data fdd-CAR)
   (data fdd-CDR)))

; Switch selon le type : s'applique a toutes les variable en local
(define-Z do-on-var    ORS do-on-var
  (dynamic (v-obj local))
  (action-g eff-recons)
  (gl-recons t)
  (gl-data (if (and (boundp 'f-entree) f-entree) (dessine-valeur-variable) (dessine-variable-vide))))

; ************************************************************************** ; ;
; Définition des fonctions Lisp utilisées pour la génération et l'animation ; ;
; des représentations de programme par des cités.                            ; ;
; ************************************************************************** ; ;

; Ces fonctions se divisent en trois groupes :
; - Les fonctions utilisées pour le calcul du positionnement des objets
; - Les fonctions définissant les primitives graphiques utilisées pour le dessin
;   d'objet particulier
; - Les fonctions utilisées pour les opérations graphiques effectuées lors de l'annimation
;   des représentations (représentation des données et dessin d'un personnage).

                                        ; ************** ; ;
                                        ; Première partie ; ;
                                        ; ************** ; ;

                                        ; Fonctions utilisées pour le calcul du positionnement ; ;


; construction d'un atome à partir d'une chaine de caractères
(defmacro MkAtome (str) `(implode (explode ,str)))  MkAtome

; positionnement des maison sur l'axe des X suivant leur type
(de maison-X ()         maison-X
  (ifn (boundp 'position-maison) (setq position-maison (gensym))) ; pile des positions

  (let (indic (Z-Get-Type))
    (let (cpt 0)
      ; nécessité de la création d'une autre rangée de maisons
      (while (> (get position-maison indic) (* block-size (fix (sqrt fdc-nelem))))
   (incr cpt)
   (setq indic (MkAtome (strcat indic "-" cpt))))
     (put position-maison 'arg cpt))
    (put position-maison indic (+ block-size (get position-maison indic)))
    (- (get position-maison indic) block-size)))

; positionnement des maison sur l'axe des Z suivant leur type
(de maison-Z ()         maison-Z
  (let (indic (MkAtome (strcat (Z-Get-Type) "-Z" (get position-maison 'arg))))
    (ifn (get position-maison indic)
    (put position-maison indic (* (- (length (cdr position-maison)) 3) (/ block-size 4))))
    (get position-maison indic)))

; récupération / affectation du positionnement des variables autours des maisons
(de var-X (func)        var-X
  (let (n-var (+ 1 (get func 'n-var)))
    (if (> n-var 4) (incr n-var)) ; pour ne pas placer de variables dans la maison
    (+ (- (Z-Get-X func 'TYPES) 10)
       (* 60 (rem (- n-var 1) 3)))))

(de var-Z (func)        var-Z
  (let (n-var (+ 1 (get func 'n-var)))
    (put func 'n-var n-var)
```

```
    (if (> n-var 4) (incr n-var))
    (+ (- (Z-Get-Z func 'TYPES) 30)
      (* 80 (/ (- n-var 1) 3)))))


(de ColorTypeR (func)          ColorTypeR
  (or (and (or (eq (setq *type-func* (Z-Get-Type (Z-Get-Func-Analyses func))) 'boucles)
          (eq *type-func* 'lambda)
          (eq *type-func* 'test))    255)
      (and (or (eq *type-func* 'entrees-sorties) (eq *type-func* 'numeriques) (eq *type-func* 'chaines))        0)
      (and (eq *type-func* 'listes)      100)
      200))

(de ColorTypeG (func)          ColorTypeG
  (or (and (or (eq *type-func* 'boucles) (eq *type-func* 'numeriques) (eq *type-func* 'listes)) 0)
      (and (or (eq *type-func* 'chaines) (eq *type-func* 'lambda) (eq *type-func* 'test))    100)
      (and (eq *type-func* 'entrees-sorties) 255)
      200))
(de ColorTypeB (func)          ColorTypeB
  (or (and (or (eq *type-func* 'boucles) (eq *type-func* 'test)) 100)
      (and (or (eq *type-func* 'entrees-sorties) (eq *type-func* 'lambda)) 0)
      (and (or (eq *type-func* 'chaines) (eq *type-func* 'numeriques) (eq *type-func* 'listes)) 255)
      200))

(defmacro ColorTypeA (func) '125)        ColorTypeA

(de color-indic (type)color-indic
   (selectq type
      (boucles        (GLcolor 4 255   0 100 125))
      (lambda         (GLcolor 4 255 100   0 125))
      (test          (GLcolor 4 255 100 100 125))
      (entrees-sorties (GLcolor 4   0 255   0 125))
      (numeriques      (GLcolor 4   0   0 255 125))
      (chaines        (GLcolor 4   0 100 255 125))
      (listes         (GLcolor 4 100   0 255 125))
      (t             (GLcolor 4 200 200 200 125))))


                                                ; ************** ; ;
                                                ; Deuxième partie ; ;
                                                ; ************** ; ;

                                                ; primitives graphique pour le dessin des objets composant la re-
présentation

(setq block-size 400) ; la taille des blocks (contenant une maison).

; Dessin de l'indicateur de quartier
(de dessine-indic ()  dessine-indic
   (let (indic (Z-Get-Type))
     ; uniquement si il n'a pas déjà été affiché
     (when (and (= (get position-maison indic) block-size)
                (> (get position-maison 'arg) 0))
  (GLcolor 3 255 255 255)
  (GLPushmatrix)
  (GLtranslate -200 0 0)
  (GLbgn "GL_LINE_LOOP")
  (GLvertex 3 0 0 0  0 20 0  -30 20 0  -30 50 0  50 50 0
            50 20 0  20 20 0 20 0 0)
  (GLend)
  (GLbgn "GL_LINE_STRIP")
  (GLvertex 3 50 50 0 70 35 0 50 20 0)
  (GLend)
  (GLboxtext -27 21 74 28 indic)
  (GLPopmatrix))))

; Dessin des valeurs atomiques (nombres, chaines, atomes)

; Dessin d'une valeur numérique
(de dessine-nombre (valeur)    dessine-nombre
  (GLcolor 3 0 0 255)
  (dessine-cube-vide (rem valeur 100)))

; Affiche un nom correspondant à une primitive ou une fonction utilisateur
(de dessine-atom (nom)          dessine-atom
   (let (f-info (and (litatom nom) (Z-Get-Func-Analyses nom)))
```

```
    ; si ce nom désigne une fonction on affiche alors sa maison
    (if f-info (dessine-maison f-info)
  (GLboxtext 0 0 90 30 (strcat nom)))))

; Affiche un nom correspondant à une chaine de caractères
(de dessine-chaine (nom) (GLboxtext 0 0 90 30 (strcat nom)))          dessine-chaine

(de dessine-primitive (nom) (GLboxtext 0 0 90 30 (strcat nom)))       dessine-primitive

; dessin d'un cube plein
(de dessine-cube-plein ()  (GLcubeSolid 100))     dessine-cube-plein

; dessin d'un cube vide
(de dessine-cube-vide (taille)  (GLcubeWire taille))dessine-cube-vide

; ****************** ; ;
; Dessin des maisons ; ;
; ****************** ; ;

; dessin d'une maison vide relative à un appel de fonction
(de dessine-maison (donnees)dessine-maison
  (GLPushmatrix)
  (GLtranslate 15 -15 -30)
  (GLscale 6 6 6 10)
  (dessine-toit-plein donnees)
  (GLcolor 4 155 155 155 100)
  (GLtranslate 50 50 50)
  (GLcubeSolid 100)
  (GLPopmatrix))

(defmacro mkcolor (n) `(1+ (* ,n 100)))    mkcolor

; Dessin du toit des maisons
(de dessine-toit-plein (vals)    dessine-toit-plein
   (setq taille 100
    tuile-tx (/ 100 6)
    tuile-tz (/ 100 4))
   ; dessin des tuiles du toit : l'intensité de la tuille dépend du nombre
   ; résultant de la numération sur le type visualisé.
   (GLcolor 4 (mkcolor (get vals 'loop))   0 0 125) (dessine-tuile 1 0)
   (GLcolor 4 (mkcolor (get vals 'lambda)) 0 0 125) (dessine-tuile 1 1)
   (GLcolor 4 (mkcolor (+ (or (get vals 'test) 0) (get vals 'conds)))  50 50 125) (dessine-tuile 1 2)
   (GLcolor 4  0  0 (mkcolor (get vals 'num)) 125)   (dessine-tuile 2 0)
   (GLcolor 4  0  0 (mkcolor (get vals 'str)) 125)   (dessine-tuile 2 1)
   (GLcolor 4  0  0 (mkcolor (get vals 'lst)) 125)   (dessine-tuile 2 2)
   (GLcolor 4  0 (mkcolor (get vals 'io))  0 125)    (dessine-tuile 3 0)
   (GLcolor 4  0 (mkcolor (get vals 'x))   0 125)    (dessine-tuile 3 1)
   (GLcolor 4  0 (mkcolor (get vals 'gl))  0 125)    (dessine-tuile 3 2)
   ; représentation en dernière ligne de la présence de variables
   (GLcolor 4 (mkcolor (get vals 'local))
        (mkcolor (get vals 'local))
        (mkcolor (get vals 'local)) 125)  (dessine-tuile 4 0)
   (GLcolor 4 (mkcolor (get vals 'global))
        (mkcolor (get vals 'global))
        (mkcolor (get vals 'global)) 125) (dessine-tuile 4 1)
   (GLcolor 4 (mkcolor (get vals 'modif))
        (mkcolor (get vals 'modif))
        (mkcolor (get vals 'modif)) 125)  (dessine-tuile 4 2)
   ; dessin des cotés du toit
   (color-indic (Z-Get-Type vals))
   (GLbgn "GL_POLYGON") (GLvertex 3   0 100   0   50  125   0 100 100   0) (GLend)
   (GLbgn "GL_POLYGON") (GLvertex 3   0 100 100   50  125 100 100 100 100) (GLend))

; dessin d'une tuille suivant sa position ligne/colonne.
(de dessine-tuile (r c)              dessine-tuile
   (let ((x (* c tuile-tx)) (y (or (and (= c 2) 50) (* (1+ c) tuile-tx)))
    (z0 (* (1- r) tuile-tz)) (z1 (* r tuile-tz)))
     (GLbgn "GL_POLYGON")
     (GLvertex 3 x (+ 100 (/ x 2)) z0  y (+ 100 (/ y 2)) z0
                 y (+ 100 (/ y 2)) z1  x (+ 100 (/ x 2)) z1)
     (GLend)
     (GLbgn "GL_POLYGON")
     (GLvertex 3
               (- 100 x) (+ (/ x 2) 100) z0
               (- 100 y) (+ 100 (/ y 2)) z0
               (- 100 y) (+ 100 (/ y 2)) z1
```

```
            (- 100 x) (+ (/ x 2) 100) z1)
    (GLend)))


; dessine le contour du jardin des maisons : place en Y = -1
(de dessine-jardin () dessine-jardin
  ; dessin du sol
  (GLbgn "GL_POLYGON")
  (GLvertex 3 -50 -1 -50     150 -1 -50    150 -1 150   -50 -1 150)
  (GLend)
  ; dessin de la bariere : porte en X: 150 Z: 75 Taille: 25
  ; hauteur = 25
  (GLcolor 3 255 255 255)
  (GLlineWidth 3)
  (GLbgn "GL_LINE_STRIP")
  (GLvertex 3
      150 -2  25           150  8  25    150  8  -2   150 -2 -50
      150  8 -50   -50  8 -50    -50 -2 -50   -50  8 -50
      -50  8 150   -50 -2 150    -50  8 150   150  8 150
      150 -2 150   150  8 150    150  8  75   150 -2  75)
  (GLend)
  (GLlineWidth 1))

; dessin du sol (suivant le nb d'objets), place en Y = -2
(de dessine-terre ()   dessine-terre
  (GLcolor 3 150 250 200)
  (GLbgn "GL_POLYGON")
  (GLvertex 3 -3000 -5 -3000  3000 -5 -3000 3000 -5 3000 -3000 -5 3000)
  (GLend))

; dessin des chemins entourant une maison
(de dessine-chemin ()              dessine-chemin
  ; dessin du chemin de sortie de la maison
  (let (c-p (- (/ (- block-size 150) 2) 25)) ; distance de la porte au milieu du chemin
    (GLbgn "GL_POLYGON")
    (GLvertex 3 150 0 25  150 0 75  (+ 150 c-p) 0 75  (+ 150 c-p) 0 25)
    (GLend))
  ; dessine les routes entourant la maison
  (GLbgn "GL_POLYGON")
  (GLvertex 3 -150                0 (/ block-size 2)
      (+ 50 (/ block-size 2)) 0 (/ block-size 2)
      (+ 50 (/ block-size 2)) 0 (+ 50 (/ block-size 2))
      -150                  0 (+ 50 (/ block-size 2)))
  (GLend)
  (GLbgn "GL_POLYGON")
  (GLvertex 3 (/ block-size 2)      0 -150
      (+ 50 (/ block-size 2)) 0 -150
      (+ 50 (/ block-size 2)) 0 (+ 50 (/ block-size 2))
      (/ block-size 2)        0 (+ 50 (/ block-size 2)))
  (GLend))

; Troisième partie : dessins liés à l'animation de la représentation

; Homme en deplacement

(de dessine-homme (size)       dessine-homme
  (ifn size (setq size 40))
;  (GLpushAttrib "GL_CURRENT_BIT")
  ; dessin du corps
  (GLcolor 3 255 255 0)
  ; les jambes
  (GLbgn "GL_LINE_STRIP")
  (GLvertex 3 (- (/ size 3)) 0 0  0 (/ size 3) 0  (/ size 3) 0 0)
  (GLend)
  ; le corps
  (GLPushmatrix)
  (GLtranslate 0 (/ size 3) 0)
  (GLlineWidth 4)
  (GLbgn "GL_LINES") (GLvertex 3 0 0 0 0 (/ size 3) 0) (GLend)
  (GLlineWidth 1)
  ; les bras
  (GLbgn "GL_LINE_STRIP")
  (GLvertex 3 (- (/ size 3)) 0 0  0 (/ size 3) 0  (/ size 3) 0 0)
  (GLend)
  ; et la tête
  (GLtranslate 0 (+ (/ size 3) (/ size 6)) 0)
```

```
  (GLellipse 0 0 (/ size 6) (/ size 6) 0 3600)
  (GLPopmatrix)
;  (GLpopAttrib)
  )

; Gestion des contraintes du mouvement de l'homme sur les chemins

; fixe le parcour
(de is-path-x (x0 y0 z0 x1 y1 z1 x y z)      is-path-x
  (if (House-Proximity) x
    (+ 20 (+ (/ block-size 2) (* (fix (/ x block-size)) block-size)))))

(de is-path-y (x0 y0 z0 x1 y1 z1 x y z)      is-path-y
  (if (House-Proximity) y 0))

(de is-path-z (x0 y0 z0 x1 y1 z1 x y z)      is-path-z
  (if (House-Proximity) z
    (+ 20 (+ (/ block-size 2) (* (fix (/ z block-size)) block-size)))))

; Arrival-Proximity :
; vrai si pas de changement de maison ou si arrive a la maison destinataire
(defmacro House-Proximity ()  House-Proximity
  '(or (and (< (abs (- x0 x1)) (/ block-size 2))
       (< (abs (- z0 z1)) (/ block-size 2)))
       (and (< (abs (- x x1)) (/ block-size 2))
       (< (abs (- z z1)) (/ block-size 2)))))


                                          ; ************************ ; ;
                                          ; Partie pour les variables ; ;
                                          ; ************************ ; ;

; definition d'un objet vide : cube blanc
(de dessine-variable-vide (size)          dessine-variable-vide
  (ifn size (setq size 40))
  (GLPushmatrix)
  (GLcolor 3 255 255 255)
  (GLtranslate 0 20 0)
  (GLcubeWire size)
  (GLPopmatrix)
  )

                                          ; ************************************** ; ;
                                          ; Dessin des valeurs au cours de l'execution ; ;
                                          ; ************************************** ; ;

; les variables d'indication des valeurs fournies par Thema sont :
;
;   exe-var   = nom de la variable
;   exe-func  = nom de la fonction utilisant la variable lors de son acces
;   exe-value = valeur actuelle de la variable
;   exe-elem-nth = nombre d'acces a la variable
;
;   exe-var-init = valeur de la variable lors de son premier access
;   exe-var-min  = valeur minimum atteinte (pour listes et nombres)
;   exe-var-max  = valeur maximum atteinte (pour listes et nombres)
;
;   GA-X GA-Y GA-Z : position de dessin de l'ancien objet graphique

(de dessine-valeur-variable () dessine-valeur-variable
; le repositionnement est inutile : le dessin est simplement remplace...
  (GLPushmatrix)
  (GLrotate 900 "Y")
  (cond
   ((listp exe-value)   (dessine-a-list   exe-value))
   ((numbp exe-value)   (dessine-number))
   ((stringp exe-value) (dessine-a-string exe-value))
   ((null exe-value)    (dessine-a-nil    exe-value))
   ((atom exe-value)    (dessine-an-atom  exe-value)))
  (GLPopmatrix)
  )


(de dessine-number ()           dessine-number
  ; dessin d'une cube rouge pour la valeur minimale
  (GLcolor 3 255 0 0)
```

```
  (GLsphere exe-var-min exe-var-min exe-var-min)
  ; dessin d'une cube bleue pour la valeur maximale
  (GLcolor 3 0 0 255)
  (GLcubeWire (* exe-var-max 2))
  (dessine-a-number exe-value))

; affichage d'un nombre : un cube de sa grandeur
(de dessine-a-number (value position)    dessine-a-number
  (when position (GLPushmatrix) (GLtranslate (car position) (cdr position) 0))
  (GLcolor 3 255 255 255)
  (GLcubeWire value)
  (when position (GLPopmatrix)))



; partie pour le dessin des listes par cons
; construit à partir du programme de dessin des listes de Patrick Greussay
(package dessine-list-pg)


 (setq scale-factor 8)

(de |dessine-a-list (sexp)        |dessine-a-list
  (setq right-margin 1)
  (GLfont "times.g")
  (GLcentertext t)
  (cons-gl-loop sexp 3 1)
  (GLcentertext nil)
  t)

(de cons-gl-loop (sexp xpos ypos)          cons-gl-loop
  (cond ((atom sexp)
    (gl-put-value sexp xpos (+ 2 ypos)))

       ((and (real-atom (cdr sexp)) (atom (car sexp)))
        (gl-cons-cell xpos ypos)
        (gl-put-value (car sexp) (+ 4 xpos) (+ 2 ypos))
    (gl-arrow (+ 1 xpos) (+ ypos 6) (+ 4 xpos) (+ ypos 6))
        (gl-put-value (cdr sexp) (+ 4 xpos) (+ 6 ypos)))

       ((real-atom (cdr sexp))
        (gl-cons-cell xpos ypos)
        (cons-gl-loop (car sexp) (+ 4 xpos) ypos)
        (gl-arrow (1+ xpos) (+ 7 ypos) (1+ xpos) (+ 11 ypos))
        (gl-put-value (cdr sexp) (+ 1 xpos) (+ 12 ypos)))

       ((eq (car sexp) (cdr sexp))
        (gl-cons-cell xpos ypos)
        (gl-arrow (+ 1 xpos)(+ ypos 6) (+ 4 xpos) (+ ypos 6))
        (cons-gl-loop (car sexp) (+ xpos 4) ypos))

       ((eq (car sexp)(cadr sexp))
        (gl-cons-cell xpos ypos)
        (gl-arrow  (+ xpos 3) (+ ypos 2) (+ xpos 6) (+ ypos 11))
        (gl-arrow  (+ xpos 1) (+ ypos 7) (+ xpos 1) (+ ypos 11))
        (gl-cons-cell xpos (+ 11 ypos))
        (cons-gl-loop (car sexp) (+ xpos 4) (+ ypos 11))
        (if (cddr sexp)
       (let (newy (+ ypos 18))
         (while (not (> newy right-margin)) (incr newy 3))
         (incr newy)
         (gl-arrow (+ 1 xpos) (+ ypos 18) (+ 1 xpos) newy)
         (cons-gl-loop (cdr sexp) xpos newy))
     (gl-nil xpos (+ ypos 16))))

       ((null (cdr sexp))
        (gl-cons-cell xpos ypos)
        (cons-gl-loop (car sexp) (+ xpos 4) ypos)
        (gl-nil xpos (+ 5 ypos)))

       (t
        (if (and (listp (car sexp))(in? (car sexp) (cdr sexp)))
            (b-error "Mutation inconnue --- non dessinable\n" sexp))
        (gl-cons-cell xpos ypos)
        (cons-gl-loop (car sexp) (+ 4 xpos) ypos)
        (let (newy (+ ypos 7))
```

```
      (while (not (> newy right-margin)) (incr newy 3))
      (incr newy)
      (gl-arrow (+ 1 xpos) (+ ypos 7) (+ 1 xpos) newy)
          (cons-gl-loop (cdr sexp) xpos newy)))))

(de real-atom (exp)   real-atom
  (and (atom exp)
       (not (null exp))))

(de in? (exp big-exp)  in?
  (cond ((eq exp big-exp) t)
        ((null big-exp) nil)
        ((listp (car big-exp))
         (or (in? exp (car big-exp))
             (in? exp (cdr big-exp))))
        (t (in? exp (cdr big-exp)))))

(de b-error (arg1 arg2)            b-error
  (print arg1 arg2)
  (error))


(de gl-cons-cell (x y) gl-cons-cell
  (setq right-margin (+ y 9))
  (gl-cons-box x y)
  (gl-arrow (+ 1 x) (+ y 2) (+ 4 x) (+ y 2)))

; partie dessin

(de gl-put-value (val y x)          gl-put-value
  (GLPushmatrix)
  (GLtranslate (* x scale-factor) (+ (* (1+ y) scale-factor) (/ scale-factor 2)) 0)
  (cond
   ((stringp val) (|dessine-a-string val))
   ((numbp   val) (|dessine-a-number val))
   (t             (|dessine-an-atom  val)))
  (GLPopmatrix)
  )

(de gl-arrow (y0 x0 y1 x1)         gl-arrow
  (setq x0 (* x0 scale-factor)
   y0 (+ (* y0 scale-factor) (/ scale-factor 2))
   x1 (* x1 scale-factor)
   y1 (+ (* y1 scale-factor) (/ scale-factor 2)))
  (if (and (= x0 x1) (not (= y0 y1))) (setq y1 (- y1 (/ scale-factor 2))))
  (GLcolor 3 255 0 0)
  (cond
   ((or (= y0 y1) (= x0 x1)) (GLbgn "GL_LINES") (GLvertex 3 x0 y0 0 x1 y1 0) (GLend))
   (t                        (GLbgn "GL_LINES") (GLvertex 3 x0 y0 0 x0 y1 0 x1 y1 0) (GLend)))

  (if (= y0 y1)
     (progn
   (GLbgn "GL_LINE_STRIP")
   (GLvertex 3 (- x1 5) (- y1 5) 0 x1 y1 0 (- x1 5) (+ y1 5) 0)
   (GLend))
    (GLbgn "GL_LINE_STRIP")
    (GLvertex 3 (- x1 5) (- y1 5) 0 x1 y1 0 (+ x1 5) (- y1 5) 0)
    (GLend))
  )


(de gl-nil (y x)           gl-nil
  (setq x (+ (* (1- x) scale-factor) (/ scale-factor 2))
   y (* y scale-factor))
  (let ((x1 (+ x (/ (* 9 scale-factor) 2)))
   (y1 (+ y (* 3 scale-factor))))
    (GLcolor 3 0 0 255)
    (GLbgn "GL_LINES") (GLvertex 3 x y 0 x1 y1 0) (GLend)
    ))

(de gl-cons-box (y x) gl-cons-box
  (setq x (* x scale-factor)
   y (* y scale-factor))
  (let ((x1 (+ x (* 9 scale-factor)))
   (y1 (+ y (* 3 scale-factor)))
   (x0 (+ x (/ (* 9 scale-factor) 2))))
```

```
    (GLcolor 3 0 255 0)
                                                ; contour
    (GLbgn "GL_LINE_LOOP") (GLvertex 3 x y 0 x y1 0 x1 y1 0 x1 y 0) (GLend)
    (GLbgn "GL_LINES")     (GLvertex 3 x0 y 0 x0 y1 0) (GLend)))

(package) ; du dessin des listes sous forme de boites de cons
```

Annexe II.2 Définitions de l'analogie « Les programmes comme des araignées en mouvement sur une toile »

```
;(include-Z "/usr/people/damien/work/Zeugma/Zanalyses.vlisp" (analyse-Dynamique))
(include-Z "/usr/people/damien/work/lisp-expls/hanoi.vlisp" (hanoi))

; ************************************; ;
; parcours générique du flux de control ; ;
; ************************************; ;

(define-Z traverse-funcs          ORS traverse-funcs
  (generation
   (une-fonction funcs-ATOM (translate ((* Rayon-Toile (cos (/ (* 2PI funcs-ilg) funcs-nelem)))
                                        0 (* Rayon-Toile (sin (/ (* 2PI funcs-ilg) funcs-nelem)))
                                        )))
   (traverse-funcs funcs-CDR)))

(define-Z une-fonction            ORS une-fonction
  (gl-data (dessine-une-araignee))
  (gl-recons t)
  (gl-dyn-in (dessine-liens))
  (gl-dyn-out (dessine-liens))
  (action-g transfo)
  (gl-transfo (translate ((deplace-araignee 'x) (deplace-araignee 'y) (deplace-araignee 'z))))
  (dynamic (o-in t) (o-out t)))

(define-Z araignées    ORS araignées
  (root t)
  (titre "Des programmes comme des araignées")
  (func hanoi)
  (gl-data (dessine-toile))
  (generation
   (traverse-funcs incond)))


; dessin de fonction par cubes ; ;

; fonctions de positionnement ; ;

(setq 2PI 6.2831853)

; les araignées sont placées, lors de la construction de la représentation, par une
; unique transformation. Ainsi, GA-X, GA-Y et GA-Z, outre la position dans un repère
; absolue, donnent également les valeurs de la translation ayant positionné l'araignée.
; Ainsi, pour modifier les position, ces variables (comme p-x, p-y et p-z) sont utilisables.
(de deplace-araignee (axis)     deplace-araignee
  (cond
   ; appel reccursif : la fonction appelante est égale à la fonction appelée
   ((eq exe-func exe-p-func)
    (selectq axis  (x GA-X)  (y (* exe-elem-nth 5)) (z GA-Z)  (t 0)))
   ; le rapprochement ne s'effectue que à l'entrée des fonctions, indiquant ainsi une
   ; attirance particulière entre deux fonction
   (exe-in
    (cond
     (t
      (selectq axis
   (x (fix (+ GA-X (/ (- p-x GA-X) 10.0))))
   (y GA-Y)
   (z (fix (+ GA-Z (/ (- p-z GA-Z) 10.0))))
   (t 0)))))) ; le dernier atome de selectq est évalué...



; fonctions de dessin ; ;

(de dessine-toile ()   dessine-toile
  (setq Rayon-Toile (or (and (> funcs-nelem 10) (* 100 (rem funcs-nelem 10))) 200))
  (GLpushmatrix)
  (GLrotate 900 "X")
  (GLcolor 3 100 110 90)
  (GLdisk 1 Rayon-Toile  funcs-nelem)
  (GLpopmatrix)
```

```
 )

; dessin des liens d'échange de données entre deux fonction au cours
; de l'exécution du programme.
; Zeugma fournit les données suivantes :
; - GA-X, GA-Y et GA-Z : position absolue de la fonction actuellement exécutée
; - p-x, p-y, p-z      : position absolue de la fonction appelante
; - exe-in             : entrée (t) ou sortie (nil) de la fonction
; - exe-args           : liste des paramètres de la fonction appelée
; - sortie             : (donnée par Xbvl) valeur retournée par la fonction

(de dessine-liens ()  dessine-liens
  ; dx, dy et dz indiquent la distance entre les deux fonctions, le dessin
  ; des liens serra effectué dans le repère relatif à la fonction appelée
  (let ((dx (- p-x GA-X))
   (dy (- p-y GA-Y))
   (dz (- p-z GA-Z))
   (x0 (/ (- p-x GA-X) 2.0))
   (y0 (/ (- p-y GA-Y) 2.0))
   (z0 (/ (- p-z GA-Z) 2.0)))
    ; premiere partie : dessin des liens symbolisant les paramètres
    (let ((n-var (length exe-args))
     (size 30.0)
     (n 1))
      (mapc exe-args
       (lambda (x)
         (cond
          ((listp (setq aux (eval x))) (GLcolor 3 255 0 0))
          ((numbp aux)   (GLcolor 3 0 255 0))
          ((stringp aux) (GLcolor 3 0 0 255))
          ((atom aux)    (GLcolor 3 255 0 255))
          (t (GLcolor 3 255 255 255)))
         (GLbgn "GL_LINES")
         (GLvertex 3 (- (/ size 2) (* n (/ size (1+ n-var)))) 0 0
                           x0 y0 z0)
         (GLend)
         (incr n))))
    ; deuxieme partie : valeur retournee
    (cond
     ; entrée dans la fonction : pas encore de valeur retournées
     (exe-in (GLcolor 3 255 255 255))
     ; examen du type de la valeur retournée
     ((listp sortie) (GLcolor 3 255 0 0))
     ((numbp sortie) (GLcolor 3 0 255 0))
     ((stringp sortie) (GLcolor 3 0 0 255))
     ((atom sortie)    (GLcolor 3 255 0 255))
     (t (GLcolor 3 155 155 155)))
    (GLbgn "GL_LINES") (GLvertex 3 x0 y0 z0 dx dy dz) (GLend)))

;      (GLbgn "GL_LINE_STRIP") (GLvertex 3  (- x0 5) y0 (- z0 5) x0 y0 z0 (+ x0 5) y0 (- z0 5)) (GLend)
; Dessin d'une fonction :
;
; utilisation des numérations sur la composition des fonctions
;
(de dessine-une-araignee ()    dessine-une-araignee
  (GLpushmatrix)
;    (GLrotate 900 "X")
    ; la rotation permet à l'araignée de regarder vers le centre de la toile
    (GLrotate (- 900 (* (/ 3600.0 funcs-nelem) funcs-ilg)) "Y")
    (GLpushmatrix)
      (GLtranslate -10 0 -10)
      (dessine-corps-araignee 20)
    (GLpopmatrix)
  (GLpopmatrix))


; normalisation de la couleur
(defmacro mkcolor (n) `(or (and (setq tc (1+ (* ,n 75))) (< tc 255) tc) 255))          mkcolor

; dessin d'un carré de couleur partie d'une face
(de dessine-partie-face (r c cr cg cb)      dessine-partie-face
    (GLcolor 4 cr cg cb 125)
    (let ((x0 (* c taille-partie-x)) (x1 (* (1+ c) taille-partie-x))
     (y0 (* r taille-partie-y)) (y1 (* (1+ r) taille-partie-y)))
      (GLbgn "GL_POLYGON")
```

```
        (GLvertex 3 x0 y0 0  x0 y1 0 x1 y1 0 x1 y0 0)
        (GLend)))

; une face est un rectangle 0,0,0 - 100,100,0 mosaique dont
; les couleurs sont fonctions du nombre d'instructions d'une classe
(de dessine-face ()   dessine-face
  (dessine-partie-face 0 0 (mkcolor com-loop)   0 50)
  (dessine-partie-face 0 1 (mkcolor com-lambda) 50  0)
  (dessine-partie-face 0 2 (mkcolor (+ (or com-test 0) com-conds))  50 50)
  (dessine-partie-face 1 0  0  0 (mkcolor com-num))
  (dessine-partie-face 1 1  0 50 (mkcolor com-str))
  (dessine-partie-face 1 2 50  0 (mkcolor com-lst))
  (dessine-partie-face 2 0  0 (mkcolor com-io)  0)
  (dessine-partie-face 2 1 50 (mkcolor com-x)   0)
  (dessine-partie-face 2 2  0 (mkcolor com-gl) 50)
  (dessine-partie-face 3 0 (mkcolor com-local) (mkcolor com-local) (mkcolor com-local))
  (dessine-partie-face 3 1 (mkcolor com-global) (mkcolor com-global) (mkcolor com-global))
  (dessine-partie-face 3 2 (mkcolor com-modif)  (mkcolor com-modif)  (mkcolor com-modif)))

(de dessine-corps (taille-corps) ; c'est en fait un cube...        dessine-corps
  (let ((taille-partie-x (/ taille-corps 3.0))
   (taille-partie-y (/ taille-corps 4.0)))
    (dessine-face) ; devant
    (GLpushmatrix)
    (GLtranslate 0 0 taille-corps) (dessine-face) ; derière
    (GLrotate -900 "X")           (dessine-face) ; haut
    (GLtranslate 0 0 taille-corps) (dessine-face) ; bas
    (GLrotate 900 "Y")            (dessine-face) ; gauche
    (GLtranslate 0 0 taille-corps) (dessine-face) ; droite
    (GLpopmatrix)))

; affectation de la couleur suivant le type de fonctions
(defmacro ColorLoop   (alpha) `(GLcolor (or (and ,alpha 4) 3) 255   0 100 (and ,alpha 125)))          ColorLoop
(defmacro ColorLambda (alpha) `(GLcolor (or (and ,alpha 4) 3) 255 100   0 (and ,alpha 125)))          ColorLambda
(defmacro ColorTest   (alpha) `(GLcolor (or (and ,alpha 4) 3) 255 100 100 (and ,alpha 125)))          ColorTest
(defmacro ColorIO     (alpha) `(GLcolor (or (and ,alpha 4) 3)   0 255   0 (and ,alpha 125)))   ColorIO
(defmacro ColorNum    (alpha) `(GLcolor (or (and ,alpha 4) 3)   0   0 255 (and ,alpha 125))) ColorNum
(defmacro ColorStr    (alpha) `(GLcolor (or (and ,alpha 4) 3)   0 100 255 (and ,alpha 125))) ColorStr
(defmacro ColorLst    (alpha) `(GLcolor (or (and ,alpha 4) 3) 100   0 255 (and ,alpha 125))) ColorLst
(defmacro ColorAutres (alpha) `(GLcolor (or (and ,alpha 4) 3) 200 200 200 (and ,alpha 125)))          ColorAutres

; vals = valeurs de numération pour la fonction
; alpha = présence d'une composante de transparence dans la couleur
(de couleur-type (alpha)         couleur-type
  (selectq (Z-Get-Type)
    (boucles (ColorLoop alpha))
    (lambda  (ColorLambda alpha))
    (test    (ColorTest alpha))
    (entrees-sorties (ColorIO alpha))
    (numeriques (ColorNum alpha))
    (chaines (ColorStr alpha))
    (listes  (ColorLst alpha))
    (t       (ColorAutres alpha))))

; de la tête au corps et la fin du corps
(de dessine-liens-corps (ls rs sz)          dessine-liens-corps
  (let (dl (/ (- rs ls) 2))
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  0 0 0  0 ls 0  ls ls 0  ls 0 0) (GLend)
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  0 0 0  (- dl) (- dl) sz  (- dl) (+ ls dl) sz  0 ls 0) (GLend)
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  0 ls 0  (- dl) (+ ls dl) sz  (+ ls dl) (+ ls dl) sz  ls ls 0) (GLend)
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  ls ls 0  (+ ls dl) (+ ls dl) sz  (+ ls dl) (- dl) sz  ls 0 0) (GLend)
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  0 0 0  (- dl) (- dl) sz  (+ ls dl) (- dl) sz  ls 0 0) (GLend)
    (GLbgn "GL_LINE_LOOP") (GLvertex 3  (- dl) (- dl) sz  (- dl) (+ ls dl) sz  (+ ls dl) (+ ls dl) sz  (+ ls dl) (- dl
  sz) (GLend)))

(de dessine-corps-araignee (taille)        dessine-corps-araignee
  (GLpushmatrix)
    ; le point 0 0 0 est en bas a gauche arriere du grand cube du corps
   (GLtranslate  (/ taille 4) (/ taille 4) (- (/ taille 2)))
   (GLpushmatrix)
    ; dessin du corp
```

```
  (GLpushmatrix)
  (couleur-type)
  (dessine-liens-corps (/ taille 2) taille (/ taille 2))
  (GLtranslate (- (/ taille 4)) (- (/ taille 4)) (/ taille 2))
  (dessine-corps taille)
  (couleur-type)
  (GLtranslate 0 0 taille)
  (dessine-liens-corps taille (/ taille 2) (/ taille 2))
  (GLpopmatrix)
  ; dessin de la tete
  (GLpushmatrix)
   (GLtranslate (/ taille 4) (/ taille 3) (- (/ taille 3)))
   (GLdodecahedraWire (/ taille 2))
  (GLpopmatrix)
  ; dessin des mandibules
  (GLpushmatrix)
   (GLtranslate 0 (- (/ taille 2) (/ taille 8))  (- (/ taille 2)))
   (GLbgn "GL_LINE_STRIP")
   (GLvertex 3 0 0 0
        (- (/ taille 3)) (- (/ taille 3)) (- (/ taille 3))
        (- (/ taille 6)) (- (/ taille 6)) (- (/ taille 2)))
   (GLend)
   (GLtranslate (/ taille 2) 0 0)
   (GLbgn "GL_LINE_STRIP")
   (GLvertex 3 0 0 0
        (/ taille 3) (- (/ taille 3)) (-  (/ taille 3))
        (/ taille 6) (- (/ taille 6)) (- (/ taille 2)))
   (GLend)
  (GLpopmatrix)
 (GLpopmatrix)
(GLpopmatrix)
; dessin des pates
(GLpushmatrix)
  (GLtranslate 0 (* (/ taille 3) 2) (- (/ taille 8)))
  (repeat 4 (GLtranslate 0 0 (/ taille 4))
     (GLbgn "GL_LINE_STRIP")
     (GLvertex 3 0 0 0  (- (/ taille 3)) (- (/ taille 3)) 0 (- (/ taille 3)) (- taille) 0)
     (GLend))
  (GLtranslate taille 0 (- taille))
  (repeat 4
     (GLtranslate 0 0 (/ taille 4))
     (GLbgn "GL_LINE_STRIP")
     (GLvertex 3 0 0 0  (/ taille 3) (- (/ taille 3)) 0 (/ taille 3) (- taille) 0)
     (GLend))
(GLpopmatrix))
```

Annexe II.3 Définitions de l'animations d'algorithmes de tris

```
(include-Z "/usr/people/damien/work/lisp-expls/qs.vlisp"
     (insert
      (o-in visu-algo)
      (o-out visu-algo)))

(define-Z maj-donnee ORS maj-donnee
  (gl-data (maj-visualisation-donnees)))

(define-Z visu-algo     ORS visu-algo
  (root t)
  (titre "Visualisation des algorithmes de tris")
  (func insert)
  (action-g recons)
  (generation
   (maj-donnee funcs-ATOM)
   (maj-donnee exe-entree)
   (maj-donnee exe-sortie)))

(setq size-Y 150 size-X 10
     X-pos 'X-pos
     X-len 'X-len
     win-num nil)
(setq im-num 0)

(de mkarg (aArg) (mklist (eval aArg)))      mkarg
(de mklist (aval) (or (and (listp aval) aval) [aval]))    mklist

(de maj-visualisation-donnees ()          maj-visualisation-donnees
  (when (and (boundp 'exe-in) (or exe-in exe-out))
    (if (null win-num) (setq win-num (xGetWindow (xGetValues Meta-GL "parent"))))
    (sh (strcat "import -frame -negate -window " win-num " " im-num ".tif")) (incr im-num)
    (let (VarsVal (if exe-in (mapcar exe-args 'mkarg) [(mklist exe-sortie)])) ; récupération des valeurs à afficher
      (when VarsVal
     (print exe-in VarsVal)
     (let ((x-pos (or (get X-pos exe-elem-nth) 0))
          (numero-variable 0)
          (nombre-donnees  0))
      (GLpushmatrix)
      ; positionnement initiale de l'affichage des données :
      ; pour l'axe des X : (+1 (and exe-out -2)) permet le positionnement en miroire par rapport à 0
      ; pour l'axe des Y : positionnement en fonction de la profondeur de récursion
      (GLtranslate (* (+ 1 (and exe-out -2)) x-pos)
                   (* (+ 5 size-Y) exe-elem-nth) 0)
      (mapc VarsVal
            (lambda (aVarValue)
              (incr nombre-donnees)    ; incrément du nombre de données affichées (pour la position)
              (mapc aVarValue
                    (lambda (aValue)
                      (incr nombre-donnees)    ; incrément du nombre de données
                      (GLtranslate size-X 0 0) ; déplacement pour afficher la donnée suivante
                      (GLpushmatrix)
                      (GLrotate 900 "X") ; rotation pour l'affichage de la valeur sur l'axe Z
                      (cond
                       ((numbp aValue) ; si la donnée à afficher est une valeur numérique
                        (GLtranslate 0 (* (/ size-X 2) aValue) 0)
                        (GLpushmatrix)
                        ; couleur de la donnée :
                        ; blanc en entrée de la fonction et noire en sortie
                        (GLcolor 3 (or (and exe-in 255) 0)
                                   (or (and exe-in (- 255 (* numero-variable 200))) 0)
                                   (or (and exe-in (- 255 (* numero-variable 100))) 0))
                        (GLscale 1 (1- aValue) 1 1)
                        (GLcubeSolid size-X)
                        (GLpopmatrix)
                        (GLscale 1 aValue 1 1)
                        ; couleur du tour de la donnée
                        (GLcolor 3 (- 255 (* exe-elem-nth 40)) ; composante rouge fonction du numéro d'ordre de la variable
                                   (+ 150 (* exe-elem-nth 30)) ; composante verte et bleue fonction de la profondeur
```

```
                                    255)
                                ; de la récursion
                                (GLcubeWire size-X))
                               ((null aValue) ; si la donnée est nil : affichage d'un cube blanc de taille fixe
                                (GLcolor 3 0 255 255)
                                (GLtranslate 0 10 0)
                                (GLscale 10 25 10 1)
                                (GLcubeSolid 1))
                               (t ; si non affichage du texte de la donnée dans une boite de taille fixe
                                (GLcolor 3 255 255 255)
                                (GLboxtext 0 0 size-X size-X (strcat aValue))))
                              (GLpopmatrix)))
                    ; incrément du nombre de variables affichées (pour la couleur)
                    (incr numero-variable)
                    ; déplacement pour la donnée suivante
                    (GLtranslate size-X 0 0)))
(GLpopmatrix)
; sauvegarde de la nouvelle position en X au moment de la sortie
(if exe-in (put X-pos exe-elem-nth (+ x-pos (* size-X nombre-donnees)))
  (put X-pos exe-elem-nth (+ x-pos 15))))))))
```

Annexe III Modifications de Xbvl

Annexe III.1 Modification de la librairie Mesa-GL : addon_dlist.c

```
/* Ce fichier doit être ajouté à la fin de src/dlist.c de la librairie MESA :
   il contient les routines nécessaire à xbvl afin de récupérer les données dont
   il a besoin pour les routines supplémentaire de gestion des listes. Si, lors de la configuration du système
   (./configure) l'argument –with-MESA=<répertoire de la librairie Mesa-GL> est spécifié, cet ajout sera effectué
   automatiquement. Pour que les fonctionnalités de gestion étendue des listes graphiques soit disponibles
   dans Xbvl il faudra alors recompiler la librairie Mesa-GL.

   DATA_TYPE sera remplacé par le type correspondant à l'architecture (32 ou 64 bits)
*/

/* Récupération des données :
   0 = taille d'un noeud
   1 = taille de la liste des opcode
   2 = tailles des données pour chaque opcode
   3 = récupération des données d'une liste
   4 = annonce de la création d'une liste
   5 = annonce de la fin de la création d'une liste
   6 = allocation d'une nouvelle instruction dans une liste
*/
DATA_TYPE mesa_xbvl_link(char data, GLuint value)
{
  GLcontext *ctx;
#ifdef THREADS
    ctx = gl_get_thread_context();
#else
    ctx = CC;
#endif
  switch(data) {
  case 0: return (DATA_TYPE) sizeof(union node);
  case 1: return (DATA_TYPE) OPCODE_END_OF_LIST; /* récupération du dernier OPCODE */
  case 11: return (DATA_TYPE) OPCODE_CALL_LIST;
  case 12: return (DATA_TYPE) OPCODE_CALL_LIST_OFFSET;
  case 2: return (DATA_TYPE) &InstSize; /* récupération de la taille des données */
  case 3: return (DATA_TYPE) HashLookup(ctx->Shared->DisplayList, value);
  case 31: return (DATA_TYPE) HashLookup(ctx->Shared->DisplayList, ctx->List.ListBase+value);
  case 4: gl_NewList(ctx, value, GL_COMPILE); return (DATA_TYPE) 1;
  case 5: gl_EndList(ctx); return (DATA_TYPE) 1;
  case 6: return (DATA_TYPE) alloc_instruction(ctx, (OpCode) value, InstSize[value]-1);
  default: return 0;
  }
}
```

Annexe III.2 Création de widgets Mesa-GL : glWidget.c

```
/*
 * glWidget.c
 * definition de la widget d'interaction entre xbvl et GL
 * ce fichier contient les callbacks adaptes a xbvl pour
 * la widget GLXDRAW.
 *
 * Damien Ploix, Decembre 93
 * Olivier Blanc et Damien Ploix, passage pour Open_GL Mars-Avril 95
 */
#if (OPEN_GL | MESA_GL)

#include <math.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#if MESA_GL
#include <GL/xmesa.h>
#endif
#include <GL/GLwDrawA.h>

/* liens avec Xbvl */
#include "vlisp.h"
/* définitions locales pour l'interface */
#include "glXbvl.h"

GLObject *wobj = NULL;              /* wobj désigne la fenêtre active */

/* ********************** */
/* Construction de l'image */
/* ********************** */

/* Construction de l'objet graphique correspondant aux transformations en cours ... */
/* Pout la matrice de projection */
static void
build_projection  (is_picking, mouse_x, mouse_y) build_projection
char is_picking;
Int  mouse_x, mouse_y;
{
  if(!is_picking){
    glViewport(0,0,wobj->width,wobj->height);
  }

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity() ;

  if(is_picking) {
    int vp[4];
    glGetIntegerv(GL_VIEWPORT, vp);
    gluPickMatrix((double) mouse_x, (double) (vp[3]-mouse_y), 5.0, 5.0, vp);
  }

  if(wobj->mode & GL_MODE_ORTHO)
    glOrtho(wobj->p_data[0], wobj->p_data[1], wobj->p_data[2],
       wobj->p_data[3], wobj->p_data[4], wobj->p_data[5]);
  else if(wobj->mode & GL_MODE_ORTHO2)
    gluOrtho2D(wobj->p_data[0], wobj->p_data[1], wobj->p_data[2],
         wobj->p_data[3]);
  else if(wobj->mode & GL_MODE_PERSP)
    gluPerspective(wobj->p_data[0] / 10.0,wobj->p_data[1],
                   wobj->p_data[2], wobj->p_data[3]);
  else if(wobj->mode & GL_MODE_FURSTRUM)
    glFrustum(wobj->p_data[0], wobj->p_data[1], wobj->p_data[2],
         wobj->p_data[3], wobj->p_data[4], wobj->p_data[5]);

  if(!is_picking) glGetDoublev(GL_PROJECTION_MATRIX, wobj->Proj_Mat);
}
/* et pour la matrice du model (de transformations de l'image) */
static void
build_modelview  ()  build_modelview
{
  Int i;
  glMatrixMode(GL_MODELVIEW);
```

```
  glLoadIdentity();
  /* on met en premier lieu en place le lookat */
  if(wobj->mode & GL_MODE_LOOKAT)
    gluLookAt( wobj->p_eye[0],  wobj->p_eye[1],
          wobj->p_eye[2],  wobj->p_eye[3],
          wobj->p_eye[4],  wobj->p_eye[5],
          wobj->p_eye[6],  wobj->p_eye[7],
          wobj->p_eye[8]);
  /* puis on effectu les transformations d'image */
  for(i = 0; i < 3; i++)
    if(wobj->t_rotate[wobj->t_rot_order[i]])
      glRotated(wobj->t_rotate[wobj->t_rot_order[i]] / 10.,
                (wobj->t_rot_order[i] == 0) ? 1.0 : 0.0,
                (wobj->t_rot_order[i] == 1) ? 1.0 : 0.0,
                (wobj->t_rot_order[i] == 2) ? 1.0 : 0.0);
  if(wobj->t_scale[0] != 1 || wobj->t_scale[1] != 1 || wobj->t_scale[2] != 1 || wobj->t_scale[3] != 1)
    glScaled(wobj->t_scale[0] / wobj->t_scale[3],
        wobj->t_scale[1] / wobj->t_scale[3],
        wobj->t_scale[2] / wobj->t_scale[3]);
  if(wobj->t_translate[0] || wobj->t_translate[1] || wobj->t_translate[2])
    glTranslated(wobj->t_translate[0], wobj->t_translate[1], wobj->t_translate[2]);
  glGetDoublev(GL_MODELVIEW_MATRIX, wobj->Model_Mat);
}

/* recuperation des transformations de l'image */

Int get_image_transformation  () /* (GLgetImageMatrix atom) */          get_image_transformation
{
  GLdouble *Mat;
  struct atome *at = (struct atome *) a1;
  Int i;

  if(!isatom(a1) || a1 == nil || !wobj) derec;
  if(at->obj != nil) Mat = (GLdouble *) at->obj;
  else at->obj = (Int *) (Mat = (GLdouble *) XtMalloc(sizeof(double) * 16));
  memcpy(wobj->Model_Mat, Mat, 16 * sizeof(double));
  derec;
}

void
redessine  ()            redessine
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_PROJECTION);
  glLoadMatrixd(wobj->Proj_Mat);
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixd(wobj->Model_Mat);
  glCallList(wobj->root_list);
  glFlush();
  glXWaitGL();
  GLwDrawingAreaSwapBuffers((Widget)wobj->w);
}

/* ******************************** */
/* gestion des widgets et objets GL */
/* ******************************** */

/* récupération d'une fenêtre GL à partir de son widget */
static GLObjectPtr
getWObject  (w)       getWObject
Widget w;
{
  GLObjectPtr ret;
  for(ret = bgl; ret < hgl; ) {
   if((Widget)ret->w == w) return ret ;
   ret++;
     }
  return NULL;
}

            /* ****************************** */
            /* callbacks liés aux fenêtres GL */
            /* ****************************** */

/* ************************************************** */
/* appelle a l'initialisation (au moment du XtRealize) */
```

```
/* ************************************************** */
static void
gl_ginit_callback  (w, client_data, call_data)          gl_ginit_callback
Widget w;
caddr_t client_data;
GLwDrawingAreaCallbackStruct *call_data ;
{
  GLObjectPtr list_shared = NULL;
  XVisualInfo *vi;
  Window windows[2];
  Display *gl_dpy;
  Arg args[1];
  Int i;

  /* creation d'un nouvel objet pour la fenetre */
  wobj = (GLObjectPtr) client_data;
  gl_dpy = (Display *) wobj->root_list;

  /* partage des listes ? */
  if(wobj->w != nil) list_shared = getWObject(wobj->w);

  wobj->w = (Int *) w;
  wobj->mode = GL_MODE_ORTHO;    /* projection orthogonale */
  /* pas de callback... */
  wobj->callback_number = 0;

  /* Taille initiale */
  wobj->width = call_data->width;
  wobj->height = call_data->height;

  /* donnees de la projection Orthogonale par defaut */
  wobj->p_data[0] = -0.5;  wobj->p_data[1] = (double)call_data->width+0.5;
  wobj->p_data[2] = -0.5;  wobj->p_data[3] = (double)call_data->height+0.5;
  /* Z min et max */
  wobj->p_data[4] = (double)(-1 * (1 << 16))-0.5;
  wobj->p_data[5] = (double)(1 << 16)+0.5;

  /* données des transformation par défaut */
  wobj->t_translate[0] = wobj->t_translate[1] = wobj->t_translate[2] = 0;
  wobj->t_rotate[0] = wobj->t_rotate[1] = wobj->t_rotate[2] = 0;
  wobj->t_rot_order[0] = 0, wobj->t_rot_order[1] = 1; wobj->t_rot_order[2] = 2;
  wobj->t_scale[0] = wobj->t_scale[1] = wobj->t_scale[2] = wobj->t_scale[3] = 1;

  XtVaGetValues(w,GLwNvisualInfo, &vi, NULL);

  wobj->glx_context = glXCreateContext(gl_dpy, vi,
                                       (list_shared) ? list_shared->glx_context : NULL,
#if OPEN_GL
                                       /* le rendu est direct en cas d'utilisation d'OPEN_GL */
                                       GL_TRUE
#else
                              /* Et Via X pour Mesa */
                                       GL_FALSE
#endif
                                       );

  GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);

  build_projection(0,0,0);
  build_modelview();

  wobj->root_list = 0; /* pas de liste a priori (GLrootlist n) */

  glDrawBuffer(GL_BACK);
  glShadeModel(GL_FLAT) ;
}

/* ************************************************** */
/* Expose : appelle a chaque redessin de la fenetre */
/* ************************************************** */
static void
gl_expose_callback  (w, client_data,call_data)      gl_expose_callback
Widget w;
caddr_t client_data;
GLwDrawingAreaCallbackStruct *call_data ;
{
```

```
  GLObjectPtr old_w = NULL;
  if(wobj != (GLObjectPtr) client_data) {
    old_w = wobj;
    wobj = (GLObjectPtr) client_data;
    GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);
  }
  redessine();
  if(old_w) {
    wobj = old_w;
    GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);
  }
}

/* ************************************************** */
/* Resize : appelle a chaque modification de la taille */
/* ************************************************** */
static void
gl_resize_callback  (w, client_data,call_data)        gl_resize_callback
Widget w;
caddr_t client_data;
GLwDrawingAreaCallbackStruct *call_data ;
{
  GLObjectPtr gl_w = wobj;

  wobj = (GLObjectPtr) client_data;

  wobj->width = call_data->width;
  wobj->height = call_data->height;

  if(wobj->mode & GL_MODE_PERSP)
    wobj->p_data[1] = (double) call_data->width / (double) call_data->height;

  else if(wobj->mode & GL_MODE_ORTHO) {
    wobj->p_data[1] = (double)call_data->width+0.5;
    wobj->p_data[3] = (double)call_data->height+0.5;
    wobj->width = call_data->width;
    wobj->height = call_data->height;
  }
  if(wobj == gl_w) build_projection(0,0,0);
  else wobj = gl_w;
}

/* ********************************** */
/* Destroy : appelle a la destruction */
/* ********************************** */
static void
gl_destroy_callback  (w, client_data,call_data)      gl_destroy_callback
Widget w;
caddr_t client_data;
GLwDrawingAreaCallbackStruct *call_data ;
{
  freegl((GLObjectPtr) client_data);
}

/* ******************** */
/* Callback des entrees */
/* ******************** */
#define CALL_BUFFER_SIZE   1024

#define incr_ptr(l)             { Int add_l = l; \
                                  if(ptr+add_l >= end_callBuffer) {  \
                                    Int pos = (Int) (ptr - callBuffer); len += CALL_BUFFER_SIZE; \
                                    callBuffer = XtRealloc(callBuffer, (unsigned) len * sizeof(unsigned
int)); \
                                    end_callBuffer = callBuffer+len; ptr = callBuffer+pos; } \
                                  ptr += add_l; }


static void
gl_input_callback  (w, client_data, call_data)        gl_input_callback
Widget w;
caddr_t client_data;
GLwDrawingAreaCallbackStruct *call_data ;
{
  char *callBuffer = XtMalloc(CALL_BUFFER_SIZE);
  char *ptr, *xbvl_mess, *end_callBuffer;
```

```
register Int i = 0, len = CALL_BUFFER_SIZE;
XEvent evnt;
GLObjectPtr oldw = NULL;
if(wobj != (GLObjectPtr) client_data) {
  oldw = wobj;
  wobj = (GLObjectPtr) client_data;
  GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);
}

/* pas de callbacks pour la fenetre */
if(!wobj->callback_number) return;

for(i = 0; i < wobj->callback_number; i++)
  if(wobj->callback_event[i] == call_data->event->type) break;

if(i >= wobj->callback_number) return;

fixeBind(w);
xbvl_mess = wobj->callback_expression[i];

/* expession, le cas echeant, de l'expression avant de la passee a XBVL */
/*
 * on insere les arguments remplacant le macros suivants :
 * $X par la positon en x
 * $Y par la positon en y
 * $B par le numero du bouton de la souris
 * $G par le(s) couple(s) (objet numero) correspondant a l'objet selectionne
 * $C par le caractere saisi au clavier
 & $W par l'atome correspondant a la fenetre dans laquelle la saisie a eut lieu
 */

memset(callBuffer, '\0', CALL_BUFFER_SIZE); /* netoyer le buffer */
ptr = callBuffer;
end_callBuffer = ptr + CALL_BUFFER_SIZE;
while( *xbvl_mess ) {
  if(*xbvl_mess == '$' )
    switch(*(xbvl_mess + 1)) {
      case 'W' :
   sprintf(ptr, "%s", (char *) (((struct atome *) wobj->atome)->pn + (2/2)));
   incr_ptr(strlen(ptr));
   xbvl_mess += 2;
   break;
 case 'B' :
   sprintf(ptr, "%d", (call_data->event->xmotion.state >> 8));
   incr_ptr(strlen(ptr));
   xbvl_mess += 2;
   break;
 case 'G' :
   if(call_data->event->type == ButtonRelease ||
      call_data->event->type == ButtonPress) {

     /* envoi de la recherche de l'objet ... */
     unsigned int pickbuff[BUFSIZ], *ui_ptr, numpicked, j, buf_pos;
     int viewport[4];

     glGetIntegerv(GL_VIEWPORT, viewport);

     glSelectBuffer(BUFSIZ, pickbuff);
     glRenderMode(GL_SELECT);
     glInitNames();
     glPushName(-1);

     build_projection(1, call_data->event->xbutton.x, call_data->event->xbutton.y);
     glMatrixMode(GL_MODELVIEW);
     glLoadMatrixd(wobj->Model_Mat);

     glCallList(wobj->root_list);

     numpicked =glRenderMode(GL_RENDER);

     /* recherche de(s) objet(s) dans la memoire... */
     *ptr = '\"'; incr_ptr(1);
     *ptr = '(';  incr_ptr(1);

     ui_ptr = &pickbuff[0];
     for(i = 0; i < numpicked; i++) {
```

```
        buf_pos = *ui_ptr;
        ui_ptr += 3;
        for(j = 0; j < buf_pos; j++) {
                sprintf(ptr, "%u ", *ui_ptr++);
                incr_ptr(strlen(ptr));
        }
       }
       *ptr = ')';  incr_ptr(1);
       xbvl_mess += 2;
     }
     else { *ptr = *xbvl_mess++; incr_ptr(1); }
     break;
   case 'C' :
     if(call_data->event->type == KeyRelease) {
       char buffer[BUFSIZ];
       long other;
       memset(buffer, '\0', BUFSIZ);
       i = XLookupString((XKeyEvent *)call_data->event,
                                  (char *)buffer, BUFSIZ,
                                        (KeySym *) &other, NULL);
       if(i) sprintf(ptr, "%s", buffer);
       incr_ptr(strlen(ptr));                xbvl_mess += 2;
     }
     else { *ptr = *xbvl_mess++; incr_ptr(1); }
     break;
   case 'X' :
     sprintf(ptr, "%d", call_data->event->xbutton.x);
     incr_ptr(strlen(ptr));                xbvl_mess += 2;
     break;
   case 'Y' :
     sprintf(ptr, "%d", call_data->event->xbutton.y);
     incr_ptr(strlen(ptr));                xbvl_mess += 2;
     break;
   default: *ptr = *xbvl_mess++; incr_ptr(1); break;
   }
    else { *ptr = *xbvl_mess++; incr_ptr(1); }
  }
  communiqExprToVlisp(callBuffer);
  if(oldw) {
    wobj = oldw;
    GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);
  }
}

                                        /* ********************** */
                                        /* Fonctions d'interaction */
                                        /* ********************** */

/* Creation d'une fenetre */
Int gl_winopen  () /* (GLwinopen nom widgetPere [listes partagees] <resource_list>) -> widget  NSUBR */
   gl_winopen
{
  Int n, type;
  CHAR *nom;
  Widget newWidget = NULL, w_pere;
  GLwDrawingAreaWidgetClass widget_class ;
  ArgList argg;
  struct atome *rep_atome;
  static Display *gl_dpy;
  XtPointer c_data;


  /* setup par defaut pour les nouvelles fenetres */
  a2 = car(a1);
  if(!((type = getNameFromArg((CHAR **)&nom, a2)) == STRING ||
       type == NAME)) {a1 = nil; derec;}

  widget_class = (GLwDrawingAreaWidgetClass)glwDrawingAreaWidgetClass ;
  a2 = car((a1 = cdr(a1)));
  if(isWidget(a2)) {
    int erB, evB;
    w_pere = GET_WIDGET_FROM_ATOME(a2);
    gl_dpy = XtDisplay(w_pere);
    if(glXQueryExtension(gl_dpy, &erB, &evB) == False) {
      wobj = NULL;
      a1 = nil;
```

```
            derec;
          }
        }
        else {
          a1 = nil;
          derec;
        }

        /* il ne nous reste que faire la creation du nouveaux widgette */
        a1 = cdr(a1);
        /* en premier lieu de verifier si la nouvelle widget OpenGL partage ses
           listes avec une autre */
        if(isWidget(car(a1))) {
        /* verification d'un partage de listes entre deux fenetres */
          a2 = (Int *)GET_WIDGET_FROM_ATOME(car(a1));
          a1 = cdr(a1);
        }
        else a2 = nil;
        /* en reccuperant des eventuels ressources */
        getArgList(XtClass(w_pere), (WidgetClass) widget_class, (ArgList *) &argg, (Int *) &n);
        argg = (ArgList) XtRealloc((String)argg, (unsigned) (n+3) * (unsigned) sizeof(Arg));

        XtSetArg(argg[n], GLwNdoublebuffer, True); n++;
        XtSetArg(argg[n], GLwNrgba, True); n++;
        XtSetArg(argg[n], GLwNdepthSize, 1); n++;

        newWidget = XbvlCreateManagedWidget(nom, (WidgetClass) widget_class, w_pere, argg, n,
&rep_atome);

        if(!newWidget){
          printf("ERRor, Impossible de creer le Widget <%s>\n", nom);
          a1 = nil;
          derec;
        }
        c_data = (XtPointer) cregl();  /* les callbacks recevrons l'objet GLObject correspondant a la fenetre */
        XtAddCallback(newWidget, GLwNginitCallback,  gl_ginit_callback,c_data);
        XtAddCallback(newWidget, GLwNinputCallback,  gl_input_callback,c_data);
        XtAddCallback(newWidget, GLwNexposeCallback, gl_expose_callback,c_data);
        XtAddCallback(newWidget, GLwNresizeCallback, gl_resize_callback,c_data);
        XtAddCallback(newWidget, "destroyCallback",  gl_destroy_callback,c_data);

        if(type == STRING) XtFree(nom);
        /* un nouveau widget */
        interneObjetWidget(rep_atome, newWidget);
        ((GLObjectPtr) c_data)->atome = a1 = (Int *)rep_atome;
        /* informations transférées à la création */
        ((GLObjectPtr) c_data)->w = a2; /* partage de listes */
        ((GLObjectPtr) c_data)->root_list = (Int) gl_dpy;

        rep_atome->tags |= GL_WINDOW;

        if((Int *) *atcareful==t && ((Int) pbind>0||(Int *) *atcrewid!=nil)) {
          if(!pbind) a5=(Int *)*atcrewid;
          else{a5= (Int *)*(pbind-2); a5 = (Int *)*(a5+2);}
          adapteival(atcrewid,a1,a5);
        }
        derec;
      }

      /* Procedure d'ajout d'un callback a une fenetre de dessin :
       * Ceci consiste uniquement a ajouter l'evenements et l'expression
       * dans la pile des callbacks de la fenetre
       */
      Int gl_addcallback  () /* (GLcallback "EventType"  "Expression") SUBR2 */          gl_addcallback
      {
        Int i = 0, eType;
        char *expr = (isstr(a2)?(char *)(a2+(2/2)):
                      (isatom(a2)?(char *)(((struct atome *)a2)->pn+(2/2)):NULL)),
             *name = (isstr(a1)?(char *)(a1+(2/2)):
                      (isatom(a1)?(char *)(((struct atome *)a1)->pn+(2/2)):NULL));

        if(!name || !expr || !wobj) { a1=nil; derec; }

        if((eType = getEventTypeByName(name)) < 0) { a1=nil; derec;}

        /* recherche d'un evenement de ce type pour la fenetre */
```

```
  if(!wobj->callback_number) {
    wobj->callback_expression =(CHAR **) XtMalloc(sizeof(CHAR *));
    wobj->callback_event =(Int *) XtMalloc(sizeof(Int));
    wobj->callback_event[0] = eType;
    ++wobj->callback_number;
  }
  else {
    for(i = 0; i < wobj->callback_number; i++)
      if(wobj->callback_event[i] == eType) break;

    if(i >= wobj->callback_number) {
      wobj->callback_expression =(CHAR **) XtRealloc((char *)wobj->callback_expression,
                                                     ++wobj->callback_number*sizeof(char *));
      wobj->callback_event =(Int *) XtRealloc((char *)wobj->callback_event,
                                              wobj->callback_number*sizeof(Int));
      wobj->callback_event[i] = eType;
    }
    else XtFree(wobj->callback_expression[i]);
  }

  wobj->callback_expression[i] = strcpy(XtMalloc(strlen(expr)+1), expr);
  a1=t;
  derec;
}


/* Procedure de choix de la fenetre de dessin GL */
Int gl_winset  () /* (GLwinset atome) SUBR1 */       gl_winset
{
  Display *dpy;
  Widget w;
  if(wobj && isWidget(a1) && (w = GET_WIDGET_FROM_ATOME(a1)) &&
     (Widget)wobj->w != w) {
     if(wobj = getWObject(w)) {
   GLwDrawingAreaMakeCurrent((Widget)wobj->w, wobj->glx_context);
   a1=t;
     } else a1=nil;
  } else a1=nil;
  derec;
}

/* gestion des attributs */

Int gl_pousse_attrib  ()          gl_pousse_attrib
{
  long mask = 0;
  Int max;

  if(a1 == nil) mask = GL_ALL_ATTRIB_BITS;
  else
    while(a1 != nil) {
      if(isstr(car(a1))) mask |= getGLvalue((char *) (car(a1)+(2/2)));
      a1 = cdr(a1);
    }
  if(mask != 0) { glPushAttrib(mask); a1 = t; }
  else a1 = nil;
  derec;
}

Int gl_tire_attrib  ()    gl_tire_attrib
{
  glPopAttrib();
  derec;
}

Int gl_copy_attrib  () gl_copy_attrib
{
  long mask = GL_ALL_ATTRIB_BITS;
  GLObjectPtr wobj1, wobj2;
  Display *dpy1, *dpy2;
  Widget w1, w2;

  if(isstr(a3)) mask = getGLvalue((char *) (a3+(2/2)));

  if(isWidget(a1) && (w1 = GET_WIDGET_FROM_ATOME(a1)) && (wobj1 = getWObject(w1)) &&
     isWidget(a2) && (w2 = GET_WIDGET_FROM_ATOME(a2)) && (wobj2 = getWObject(w2))) {
```

```
    dpy1 = XtDisplay(w1);
    glXCopyContext(dpy1, wobj1->glx_context, wobj2->glx_context, mask);
    a1 = t;
   }
   else
   a1 = nil;
   derec;
}

/* clear de la fenetre GL courante */
Int gl_clear  ()          gl_clear
{
  wobj->root_list = 0;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glFlush();
  redessine();
  derec;
}


/* ********************************* */
/* Gestion de la projection utilisée */
/* ********************************* */

/* Orthonormée */
Int gl_ortho  () /* (GLortho x1 x2 y1 y2 [z1 z2]) NSUBR */       gl_ortho
{
  Int cnt;

  /* on était en mode perspective : on repositionne normalement */
  if(wobj->mode | GL_AUTO_PERSP) {
    wobj->t_translate[2] -= wobj->p_data[2];
    build_modelview();
  }

  wobj->mode &= GL_RESET_PROJ;

  if(getNumArgFroma1((char *)wobj->p_data, 0, GLXBVL_SHORT) == 4)
    wobj->mode |= GL_MODE_ORTHO2;
  else
    wobj->mode |= GL_MODE_ORTHO;
  build_projection(0,0,0);
  redessine();
  a1 = t;
  derec;
}
#define max(v0, v1)    ((v0 > v1) ? v0 : v1)
Int gl_auto_perspective  ()      gl_auto_perspective
{
  double zmin, zfar, ratio, fovy;
  double pix_size_x, pix_size_y;
  Int w, h, wmm, hmm;
  Display *dpy = XtDisplay((Widget)wobj->w);
  Screen *scr = XtScreen((Widget)wobj->w);

  w = (Int) scr->width;
  wmm = (Int) scr->mwidth;
  h = (Int) scr->height;
  hmm = (Int) scr->mheight;
  pix_size_x = w / (double) wmm;
  pix_size_y = h / (double) hmm;

  /* on était en mode perspective : on repositionne normalement */
  if(wobj->mode | GL_AUTO_PERSP) wobj->t_translate[2] -= wobj->p_data[2];

  wobj->mode &= GL_RESET_PROJ;
  wobj->mode |= GL_AUTO_PERSP;
  /* ZMIN : distance minimale = 750mm * npixels/mm */
  wobj->p_data[2] = 750 * max(pix_size_x, pix_size_y);
  /* ZFAR : a voir... */
  wobj->p_data[3] = 2 * zmin;
  /* ASPECT */
  wobj->p_data[1] = (double) wobj->width / (double) wobj->height;
  /* FovY  = 2 atan((hauteur / 2) / distance) */
  wobj->p_data[0] = (2 * atan(wobj->height / (2.0 * wobj->p_data[2]))) * 900.0 / M_PI_2; /* résultat en 10ème
de degré
```

```
*/
  /* on ajoute la translation en Z de Zmin pour regarder vers 0... */
  wobj->t_translate[2] += wobj->p_data[2];
  /* et on reconstruit les matrices ... */
  build_projection(0,0,0);
  build_modelview();
  redessine();
  a1 = t;
  derec;
}

/* Pespective */
lnt  gl_perspective     () /* (GLperspective fovy [Znear] [Zfar]) (aspect calculated automatically)*/
   gl_perspective
{
  if(!isnum(a1)) { a1 = nil; derec; }

  /* on était en mode perspective : on repositionne normalement */
  if(wobj->mode | GL_AUTO_PERSP) {
    wobj->t_translate[2] -= wobj->p_data[2];
    build_modelview();
  }

  wobj->p_data[0] = (double)valnb(a1); /* fovy */
  wobj->p_data[1] = (double) wobj->width / (double) wobj->height; /* aspect */
  if(isnum(a2)) wobj->p_data[2] = (double)valnb(a2); /* zmin */
  if(isnum(a3)) wobj->p_data[3] = (double)valnb(a3); /* zfar */

  wobj->mode &= GL_RESET_PROJ;
  wobj->mode |= GL_MODE_PERSP;
  build_projection(0,0,0);
  redessine();
  a1 = t ;
  derec;
}

/* Window */
lnt gl_frustum  ()      gl_frustum
{
  lnt cnt;

  /* on était en mode perspective : on repositionne normalement */
  if(wobj->mode | GL_AUTO_PERSP) {
    wobj->t_translate[2] -= wobj->p_data[2];
    build_modelview();
  }
  wobj->mode &= GL_RESET_PROJ;
  wobj->mode |= GL_MODE_FURSTRUM;
  for(cnt = 0, a2 = car(a1); ; a2=car(a1 = cdr(a1)), cnt++) {
    if(!isnum(a2)) { --cnt; break; }
    wobj->p_data[cnt] = (double) valnb(a2);
    if(isnotlist(a1)) break;
  }
  a1 = t;
  build_projection(0,0,0);
  redessine();
  derec;
}

/* Positionnement de l'oeuil */
lnt gl_lookat  () /* (GLlookat vx vy vz px py pz [upx upy upz]) NSUBR */              gl_lookat
{
  lnt cnt_arg;
  if(a1 == nil) { /* si pas d'arguments on enlève le mode lookat */
    wobj->mode &= ~GL_MODE_LOOKAT;
  }
  else {
    if(!(cnt_arg = getNumArgFroma1((char *) wobj->p_eye,0,GLXBVL_DOUBLE)))
      {a1=nil; derec;}
    if(cnt_arg == 6) wobj->p_eye[6] = wobj->p_eye[8] = 0.0, wobj->p_eye[7] = 1.0;
    else if(cnt_arg != 9) { a1 = nil; derec; }
    wobj->mode |= GL_MODE_LOOKAT;
  }
  build_modelview();
  redessine();
  a1=t;
```

```
  derec;
}


Int gl_polarv  () /* (GLpolarview dist azim inc twist) NSUBR */              gl_polarv
{
  a1=t;
  derec;

}

/* ************************************************************** */
/* transformations de l'image : gestion de la matrice de l'image */
/* ************************************************************** */

/* ordre des rotations */
Int gl_rotate_order  ()              gl_rotate_order
{
  if(!isnum(a1) || !isnum(a2) || !isnum(a3)) { a1 = nil; derec; }
  wobj->t_rot_order[0] = valnb(a1);
  wobj->t_rot_order[1] = valnb(a2);
  wobj->t_rot_order[2] = valnb(a3);
  a1 = t;
  derec;
}

/* Rotation de l'image */
Int gl_rotate_image  () /* si (GLimrotate angle [axe|angle_y angle_z]) SUBR3 */   gl_rotate_image
{
  char axis;

  if(!wobj || !isnum(a1)) {a1=nil; derec; }
  if(isstr(a2)) {
    axis =  *(char *)(a2+(2/2));
    if(axis > 'Z') axis -= 'a' - 'A';
    if(axis < 'X' || axis > 'Z') {a1=nil; derec;}
  }
  else if(isnum(a2) && isnum(a3)) axis = 0;

  if(axis) {
    wobj->t_rotate[(axis-'X')] = (double) fvalue(a1);
  }
  else {
    wobj->t_rotate[0] = (double) fvalue(a1);
    wobj->t_rotate[1] = (double) fvalue(a2);
    wobj->t_rotate[2] = (double) fvalue(a3);
  }
  build_modelview();
  redessine();
  a1=t;
  derec;
}

Int gl_scale_image  () /* (GLimscale sx sy sz fsize) NSUBR */              gl_scale_image
{
  float data[4];
  if(!wobj || !getNumArgFroma1((char *)wobj->t_scale,3,GLXBVL_DOUBLE)) { a1=nil; derec; }
  if(a1 != nil) wobj->t_scale[3] = valnb(a1);
  else wobj->t_scale[3] = 1;
  build_modelview();
  redessine();
  a1 = t;
  derec;
}

Int   gl_translate_image      () /*  (GLimtranslate  x  y  z)  SUBR3  (sx,  sy  et  sz  entiers)  */
   gl_translate_image
{
  if(!wobj || !isnum(a1) || !isnum(a2) || !isnum(a3)) { a1=nil; derec; }

  wobj->t_translate[0] = (double) fvalue(a1);
  wobj->t_translate[1] = (double) fvalue(a2);
  wobj->t_translate[2] = (double) fvalue(a3);
  build_modelview();
  redessine();
  a1=t;
```

```
  derec;
}


/* flush du buffer graphique */
Int gl_flush  () /* (GLflush) */     gl_flush
{
  if(!wobj) derec;
  redessine();
  derec;
}

/* récupération des valeurs */
Int gl_get_valeur  ()  gl_get_valeur
{
  GLenum prop;     /* propriété */
  double values[4]; /* valeurs */
  if(!isstr(a1)) { a1 = nil ; derec; }
  prop = getGLvalue(STR_FROM_STRLSP(a1));
  memset(values, '\0', 4 * sizeof(double));
  glGetDoublev(prop, values);
  a1 = cons(craflo(values[0]),
       cons(craflo(values[1]),
                 cons(craflo(values[2]),
                      cons(craflo(values[3]), nil))));
  derec;
}


#endif
```

Annexe III.3 Gestion de la mémoire et des listes : glMemory.c

```
/*
 *    glMemory.c
 * Librairie de la gestion de la memoire des objets
 * utilises pour l'interface entre Xbvl et GL (Iris)
 * D.Ploix, decembre 93
 */


#if (OPEN_GL | MESA_GL)
#include "vlisp.h"

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include "glXbvl.h"

#ifdef STDC_HEADERS
extern void *calloc  ();              *calloc
extern short my_pushname  ();my_pushname
#else
extern void *calloc  (unsigned, unsigned);*calloc
extern short my_pushname  (short);        my_pushname
#endif


/* ************************************************** *
 * Procedure d'initialisation du mode d'interaction GL,
 * Principalement l'allocation de la memoire d'objets
 * GL accessibles depuis LISP
 * ************************************************** */

Int glLibInit  ()          glLibInit
{
  Int *tmp;

  aux1 = (Int) (bgl = (GLObjectPtr) (hpil+40));
  aligner(aux1);
  bgl=(GLObjectPtr) aux1; /* on cale */

  /* construction de la free list */
  for(hgl = bgl; hgl < bgl+GL_OBJECT_MEMORY_SIZE-1; hgl++) hgl->atome = (Int *) (hgl+1);
  hgl->atome =NULL;
  free_gl = bgl;
  hgl = bgl + GL_OBJECT_MEMORY_SIZE - 1;
}

/* erreur d'allocation d'objet GL */
void
ergl  ()       ergl
{
  wst("Plus d'objets GL disponibles");
}

/* creation d'un element GL (gestion de la free list) */
GLObjectPtr
cregl  ()      cregl
{
  GLObjectPtr cur;
  /* en premier lieu verification de l'etat de la pile d'objets GL */
  if(!free_gl) gci();
  if(!free_gl) {ergl(); return NULL;}
  cur=free_gl;
  free_gl=(GLObjectPtr)free_gl->atome;
  return cur;
}

/* restitution d'un objet a la free list */
void
freegl  (obj)freegl
GLObjectPtr obj;
{
```

```
  if(obj->callback_number) {
    Int i;
    for(i = 0; i < obj->callback_number; i++)
      XtFree(obj->callback_expression[i]);
    XtFree((char *) obj->callback_expression);
    XtFree((char *) obj->callback_event);
  }
  if(!free_gl) {free_gl = obj; obj->atome = NULL;}
  else
    {
      obj->atome = (Int *) free_gl;
      free_gl = obj;
    }
}
                                /* ********************************************* */
                                /* Acces au noms (pour une selection graphique) */
                                /* ********************************************* */

Int gl_loadnom  () /* (GLloadname num) */              gl_loadnom
{
  if(!wobj || !isnum(a1)) { a1 = nil; derec;}
  glLoadName(valnb(a1));
  a1=t;
  derec;
}

short
my_pushname  (name)           my_pushname
short name;
{
  if(name < 0) name = glGenLists(1);
  glPushName(name);
  return name;
}

Int gl_pushnom  () /* (GLpushname [numero]) */     gl_pushnom
{
  if(!wobj) derec;
  if(!isnum(a1)) a1 = cranb(my_pushname(-1));
  else          my_pushname(valnb(a1));
  derec;
}

Int gl_popnom  () /* (GLpopname) */        gl_popnom
{
  if(!wobj) derec;
  glPopName();
  derec;
}

/* Demande d'un nouveau tag */
Int gl_mygentag  ()    gl_mygentag
{
  if(!wobj) derec;
  a1 = cranb(glGenLists(1));
  derec;
}


                        /* ************************************************************ */
                        /* Routines liees a la gestion des Listes (specifique Open GL) */
                        /* ************************************************************ */

Int gl_islist  ()           gl_islist
{
  Int l_num;
  if(!isnum(a1)) { a1 = nil ; derec; }
  l_num = valnb(a1);
  a1 = (glIsList(l_num) ? t : nil);
  derec;
}


Int gl_setrootlist  () /* (GLrootlist [num] [widget]) : fixe la liste root ... : SUBR2 */    gl_setrootlist
{
  GLuint lst;
```

```
 Widget w;
 GLObjectPtr tmp_w;

 if(!wobj) derec;
 if(!isnum(a1)) {a1 = cranb(wobj->root_list); derec; }
 lst = (GLuint) valnb(a1);
 if(isWidget(a2) && (w = GET_WIDGET_FROM_ATOME(a2))) {
    for(tmp_w = bgl; tmp_w < hgl; ) {
  if((Widget)tmp_w->w == w) break ;
  tmp_w++;
    }
    if(tmp_w >= hgl) {a1 = nil; derec;}
  }
 else tmp_w = wobj;
 a1 = cranb((tmp_w->root_list? tmp_w->root_list : 0)) ;
 tmp_w->root_list = lst;
 redessine();
 derec;
}

Int gl_genlist () /* (GLgenlist [range]) */   gl_genlist
{
 Int range = 1;
 if(!wobj) derec;
 if(isnum(a1)) range = valnb(a1);
 a1 = cranb(glGenLists(range));
 derec;
}


Int gl_newlist   () /* (GLnewlist [num]) : creation ou remplacement de la liste num : SUBR1 */
  gl_newlist
{
 GLuint l_num;

 if(!wobj) derec;
 if(a1 == nil) l_num = glGenLists(1);
 else l_num = (unsigned int) valnb(a1);

 glNewList(l_num, GL_COMPILE);

 a1 = cranb(l_num);
 derec;
}

Int gl_endlist () /* (GLendlist) : ferme la liste courante : SUBR0 */        gl_endlist
{
 if(!wobj) derec;
 glEndList();
 if(wobj->root_list > 0) redessine();
 a1=t;
 derec;
}


Int gl_dellist () /* (GLdellist num [range]) : efface la liste num : SUBR2 */            gl_dellist
{
 if(!wobj || a1 == nil) derec;
 glDeleteLists((GLuint) valnb(a1), (isnum(a2) ? (GLuint) valnb(a2) : 1));
 a1=t;
 derec;
}

Int gl_calllist () /* (GLcalllist num) : dessin de la liste num : SUBR1 */   gl_calllist
{
 GLuint list = valnb(a1);
 if(!wobj || a1 == nil) derec;

 glCallList((GLuint) list);
 a1 = t;
 derec;
}


Int gl_pushnewlist () /* (GLpushnewlist num1 num2) */        gl_pushnewlist
{
```

```
  if(!wobj || a1 == nil || a2 == nil) derec;
  glNewList((GLuint) valnb(a1),GL_COMPILE);
  glCallList((GLuint) valnb(a2));
  glEndList();
  glNewList((GLuint) valnb(a2),GL_COMPILE);
  derec;
}

#endif /* OPEN_GL | MESA_GL */
```

Annexe III.4 Interface avec Mesa-GL : mesaListes.c

```
/* Ce fichier contiend les ajouts d'operations sur les listes propres a Xbvlisp
   Les nouvelles fonctionnalites sont :
   - mesaCopyFlatten <l_num> <new_l_num> : copie le contenue de la liste l_num
     sans les appels a des sous-listes (applatissement de la liste)
   - mesaWriteList <file_des> <l_num> : ecrit dans file_des (descripteur de fichier)
     le contenu de la liste l_num
   - mesaReadList <file_des> <l_num> : cree la liste l_num a partir des instructions
     graphiques sauvegardees dans file_des
*/

#ifdef MODIFIEDMESA
#include "vlisp.h"

#ifdef STDC_HEADERS
extern Int mesa_xbvl_link  ();   mesa_xbvl_link
#else
extern Int mesa_xbvl_link  (char, Int);      mesa_xbvl_link
#endif
extern Int mesa_write_list  ();   mesa_write_list

static int OPCODE_END_OF_LIST = -1, OPCODE_SIZE;
static int OPCODE_CALL_LIST, OPCODE_CALL_LIST_OFFSET;
static unsigned int *OPCODE_OFFSET;
typedef Int Node;

#define OPCODE_CONTINUE     (OPCODE_END_OF_LIST - 1)
static int OPCODE_NEWLIST;

#define InstSize(opcode)   (OPCODE_OFFSET[opcode])
#define get_list_node(list) (Node *) mesa_xbvl_link(3, list)
#define get_list_offset_node(list) (Node *) mesa_xbvl_link(31, list)
#define mesa_new_list(list) mesa_xbvl_link(4, list)
#define mesa_end_list()     mesa_xbvl_link(5, 0)
#define mesa_new_node(opcode) mesa_xbvl_link(6, opcode)


/* Initialisation du lien entre MESA et Xbvl */
static void init_mesa_xbvl_link  ()          init_mesa_xbvl_link
{

  OPCODE_SIZE = mesa_xbvl_link(0, 0);
  OPCODE_END_OF_LIST = mesa_xbvl_link(1, 0);
  OPCODE_CALL_LIST = mesa_xbvl_link(11, 0);
  OPCODE_CALL_LIST_OFFSET = mesa_xbvl_link(12, 0);
  OPCODE_OFFSET = (int *) mesa_xbvl_link(2, 0);
  OPCODE_NEWLIST = OPCODE_END_OF_LIST+1;
}

static mesa_do_copy  (Node *n, int is_flatten)        mesa_do_copy
{
  Node *dest;

  while(n) {
    if(*n == (Node) OPCODE_END_OF_LIST) break;
    if(*n == (Node) OPCODE_CONTINUE) {
      n = (Node *) *(n+1);
      continue;
    }

    else if(is_flatten && *n == OPCODE_CALL_LIST) {
      mesa_do_copy(get_list_node(*(n+1)), is_flatten);
    }
    else if(is_flatten && *n == OPCODE_CALL_LIST_OFFSET) {
      mesa_do_copy(get_list_offset_node(*(n+1)), is_flatten);
    }
    else {
      dest = (Node *) mesa_new_node(*n);
      memcpy((void *) dest, (void *) n, (InstSize(*n) * sizeof(Int)));
    }
    n += InstSize(*n);
  }
```

```
}

Int mesaCopy  () /* (MESAcopylist liste_origine liste_destination is_flatten) SUBR3 */          mesaCopy
{
  int i, n_s;

  if(OPCODE_END_OF_LIST < 0) init_mesa_xbvl_link();

  mesa_new_list(valnb(a2));
  mesa_do_copy((Node *) get_list_node(valnb(a1)), a3 == t);
  mesa_end_list();
  a1 = t;
  derec;
}


int mesa_write_list  ()              mesa_write_list
{
  Node *n;
  FILE *fp;
  pop_cast(fp, (FILE *));
  pop_cast(n, (Node *));
  fwrite(&OPCODE_NEWLIST, sizeof(Int), 1, fp);
  fwrite((n+1), sizeof(Int), 1, fp);
  (void)mesa_write(fp, get_list_node(*(n+1)), t);
  derec;
}

void mesa_write  (FILE *fp, Node *n, Int *save_listes)          mesa_write
{
  while(n) {

    if(*n == OPCODE_END_OF_LIST) break;

    if(*n == OPCODE_CONTINUE) {
      n = (Node *) *(n+1);
      continue;
    }

    if(*n == OPCODE_CALL_LIST) {
      if(save_listes == nil) mesa_write(fp, get_list_node(*(n+1)), save_listes);
      else {
   push(n); push(fp); pushf(mesa_write_list);
   fwrite(n, sizeof(Int), InstSize(*n), fp);
      }
    }
    else if(*n == OPCODE_CALL_LIST_OFFSET)
      mesa_write(fp, get_list_offset_node(*(n+1)), save_listes);
    else fwrite(n, sizeof(Int), InstSize(*n), fp);
    n += InstSize(*n);
  }
}

Int mesaWriteList  () /* (MESAwritelist f_des l_num [save_listes]) SUBR 3 */        mesaWriteList
{
  Node *n;
  Int l_num = valnb(a2);
  int i, s;
  FILE *fp;

  if(OPCODE_END_OF_LIST < 0) init_mesa_xbvl_link();

  fp = (FILE *) valnb(a1);
  n = (Node *) get_list_node(l_num);
  fwrite(&OPCODE_NEWLIST, sizeof(Int), 1, fp);
  fwrite(&l_num, sizeof(Int), 1, fp);
  mesa_write(fp, n, a3);
  a1 = t;
  derec;
}

Int mesaReadList  () /* (MESAreadlist f_des [l_num]) */          mesaReadList
{
  Node *n;
  Int opc, nlst = -1;
  FILE *fp;
```

```
  int cc;

  if(OPCODE_END_OF_LIST < 0) init_mesa_xbvl_link();

  fp = (FILE *) valnb(a1);

  while((cc = fread(&opc, sizeof(Int), 1, fp))) {

    if(opc == OPCODE_END_OF_LIST) break;

    if(opc == OPCODE_NEWLIST) {
      if(nlst >= 0) {
   fseek(fp, (long) (-1 * sizeof(Int)), SEEK_CUR);
   break;
      }
      fread(&nlst, sizeof(Int), 1, fp);
      if(isnum(a2)) nlst = valnb(a2);
      mesa_new_list(nlst);
    }
    else {
      n = (Node *) mesa_new_node(opc);
      fread(n+1, sizeof(Int), (InstSize(opc)-1), fp);
    }
  }

  if(nlst >= 0) { mesa_end_list(); a1 = cranb(nlst); }
  else a1 = nil;
  derec;
}

Int mesaGetList  () /* (MESAgetlist f_des) */          mesaGetList
{
  Int opc, *lst = NULL, nlst;
  FILE *fp;
  int cc;

  if(OPCODE_END_OF_LIST < 0) init_mesa_xbvl_link();

  fp = (FILE *) valnb(a1);
  a1 = nil;

  while((cc = fread(&opc, sizeof(Int), 1, fp))) {
    if(opc == OPCODE_END_OF_LIST) break;
    if(opc == OPCODE_NEWLIST) {
      fread(&nlst, sizeof(Int), 1, fp);
      if(!lst) lst = a1 = ncons(cranb(nlst));
      else {
   *CDR(lst) = ncons(cranb(nlst));
   lst = cdr(lst);
      }
    }
    else fseek(fp, (long) ((InstSize(opc)-1) * sizeof(Int)), SEEK_CUR);
  }
  rewind(fp);
  derec;
}

#endif /* modified mesa */
```

Annexe III.5 Fonctionnalités (1) : glFuncs.c

```
/*
 *      glFuncs.c
 * definition des fonctions d'interactions avec GL
 * Ce fichier contient particulierement les routines de dessin de GL
 * D.Ploix novembre 93
 */

#if (OPEN_GL | MESA_GL)
#include <math.h>
#include "glXbvl.h"
#include "vlisp.h"

/* GESTION DE LA COULEUR */

/* procedure de changement de la couleur */
lnt gl_couleur  () /* (GLcolor ...) NSUBR (les arguments dependent du mode : */          gl_couleur
{         /* si cmode : nom, si RGB : 3|4 r g b [a] */
  float cv[4];
  lnt v;
  v = valnb(car(a1));      a1 = cdr(a1);
  cv[0] = (float)valnb(car(a1)) / 255.;  a1 = cdr(a1);
  cv[1] = (float)valnb(car(a1)) / 255.;  a1 = cdr(a1);
  cv[2] = (float)valnb(car(a1)) / 255.;  a1 = cdr(a1);
  if(!wobj) derec;
  if(v == 4)
    {
      if(isnotlist(a1)) { a1=nil; derec; }
      cv[3] = (float)valnb(car(a1)) / 255.;
      glColor4fv(cv);
    }
  else  glColor3fv(cv);

  a1 = t ;
  derec;
}

lnt gl_Couleur  () /* (GLColor ...) NSUBR */            gl_Couleur
{
  unsigned short cv[4];
  if(getNumArgFroma1((char *)&cv[0], 0, GLXBVL_SHORT) == 4)
    glColor4usv(cv);
  else glColor3usv(cv);
  a1 = t;
  derec;
}

/* Style et Epaisseur de la ligne */

lnt gl_epaisseur  () /* (GLlineWidth n) SUBR1 */     gl_epaisseur
{
  if(!isnum(a1)) {a1 = nil; derec;}
  glLineWidth((float) fvalue(a1));
  a1 = t;
  derec;
}

lnt gl_style  () /* (GLlineStipple stipple [factor]) SUBR2 */       gl_style
{
  if(!isnum(a1)) { a1 = nil ; derec; }
  glLineStipple((isnum(a2) ? (lnt) valnb(a2) : 1), (unsigned short) valnb(a1));
  a1 = t;
  derec;
}

/* ROUTINES DE DESSIN */

/* debut d'une sequence de vertex  */
lnt gl_bgn  () /* (GLbgn type) SUBR1 */    gl_bgn
{
  char *type;
  long t_value;
```

```
  if(!isstr(a1) || !wobj) derec;
  type = STR_FROM_STRLSP(a1);

  a1=t;
  if((t_value = getGLvalue(type)) >= 0)  glBegin(t_value);
  else a1 = nil;
  derec;
}

/* fin d'une sequence de vertex */
Int gl_fini  () /* (GLend) SUBR0 */          gl_fini
{
  if(!wobj) derec;
  glEnd();
  a1=t;
  derec;
}


/* dessin avec les vertex */
/* (GLvertex type ['(ratio)] data)  NSUBR */
/* type = 2 3 ou 4 */
/* data = x1 y1 [z1] [t1]  (n fois) */
#define datan(n) ((n == 4) ? &data4[0] : (n == 3) ? &data3[0] : &data2[0])
Int gl_vertex  ()       gl_vertex
{
  Int dtype,i;
  float data2[2], data3[3], data4[4], *data, ratio;
  a2 = car(a1); a1 = cdr(a1);
  if(wobj && isnum(a2) && (dtype = valnb(a2)) > 1 && dtype < 5) {
    data = datan(dtype);
    if(islist(car(a1))) { ratio=fvalue(car(car(a1))); a1=cdr(a1); }
    else ratio=1.0;
    for( ;; ) {
      if(!getNumArgFroma1((char *) data, dtype, GLXBVL_FLOAT)) break;
      for(i=0;i<dtype;i++) data[i] /= ratio;
      switch(dtype) {
      case 2 : glVertex2f(data2[0], data2[1]); break;
      case 3 : glVertex3f(data3[0], data3[1], data3[2]); break;
      case 4 : glVertex4f(data4[0], data4[1], data4[2], data4[3]) ; break;
      }
    }
    a1 = t;
    derec;
  }
  else { a1 = nil; derec; }
}

/* Swaptmesh */
Int gl_swap_tmesh  () /* (GLswaptmesh) SUBR0 */          gl_swap_tmesh
{
  a1=t;
  derec;
}

/* dessin des normales (utilisation des lumieres) *
/* (GLnormal [(ratio)] data) NSUBR */
/* ratio = diviseur des donnees */
/* data = x1 y1 z1 ... (nfois) */
Int gl_normal  ()       gl_normal
{
  Int dtype;
  float data[3];
  float ratio;

  if(wobj) {

    if(islist(car(a1))) {
      ratio=fvalue(car(car(a1)));
      a1=cdr(a1);
    }


    if(isnum(a2) && (dtype = valnb(a2)) > 1 && dtype < 5)
      for( ;; ) {
   if(!getNumArgFroma1((char *) data,3,GLXBVL_FLOAT)) break;
```

```
    glNormal3fv(data);
      }
    a1 = t;
  }
  derec;
}

/* Utilisation des surfaces */

/* Beziers */

static Int my_dummy_length  (l)              my_dummy_length
Int *l;
{
  Int ret = 0;
  while(islist(l)) { ++ret; l = cdr(l); }
  return ret;
}

/* Une dimension */

Int gl_bezier_map_1  () /* (GLmap1 Target u1 u2 control_points) NSUBR */        gl_bezier_map_1
{
  Int stride = 4, n_ctrl_pt = 0, i, j;
  long map_param;
  float u1, u2, *ctrl_pts = NULL;

  if(a1 == nil) derec;

  /* recherche du type */
  a2 = car(a1);  a1 = cdr(a1);

  if(!isstr(a2) || !wobj) { a1 = nil; derec; }
  if((map_param = getGLvalue((char *) (a2+(2/2)))) < 0) { a1 = nil; derec; }

  /* recuperation de la plage */
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  u1 = fvalue(a2);
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  u2 = fvalue(a2);

  /* recuperation des points de control */
  a1 = car(a1);
  n_ctrl_pt = my_dummy_length(a1);
  ctrl_pts = (float *) XtMalloc(n_ctrl_pt * 4 * sizeof(float));

  for(i = 0; a1 != nil && i < n_ctrl_pt; i++) {
    a2 = car(a1);  a1 = cdr(a1);
    for(j = 0; j < 4 && a2 != nil; j++, a2 = cdr(a2))
      ctrl_pts[(i * 4) + j] = (float) fvalue(car(a2));
  }
  /* finalement : execute Map1... */
  if(n_ctrl_pt) {
    glMap1f(map_param, u1, u2, stride, n_ctrl_pt, ctrl_pts);
    XtFree((char *)ctrl_pts);
    a1 = t;
  }
  derec;
}

Int gl_bezier_eval_1  () /* (GLeval1 val) SUBR1 */  gl_bezier_eval_1
{
  float val;
  if(!isnum(a1) || !wobj) { a1 = nil ; derec; }
  val = (float) fvalue(a1);
  glEvalCoord1f(val);
  a1 = t;
  derec;
}

Int gl_bezier_grid_1  () /* (Ggrid1 n u1 u2) SUBR3 */          gl_bezier_grid_1
{
  Int n;
  float u1, u2;
  if(!isnum(a1) || !isnum(a2) || !isnum(a3) || !wobj) { a1 = nil ; derec; }

  n = valnb(a1);
  u1 = (float) fvalue(a2); u2 = (float) fvalue(a3);
```

```
  glMapGrid1f(n, u1, u2);
  a1 = t;
  derec;
}

Int gl_bezier_mesh_1  () /* (GLmesh1 type p1 p2) SUBR3 */                gl_bezier_mesh_1
{
  long type;
  Int p1, p2;
  if(!isstr(a1) || !isnum(a2) || !isnum(a3) || !wobj ||
     (type = getGLvalue((char *) (a1 + (2/2)))) < 0) { a1 = nil; derec; }

  p1 = valnb(a2);  p2 = valnb(a3);

  glEvalMesh1(type, p1, p2);
  a1 = t;
  derec;
}

/* Deux Dimensions */

Int gl_bezier_map_2  () /* (GLmap2 Target u1 u2 v1 v2 values) NSUBR */          gl_bezier_map_2
{
  Int u_order = 0, v_order = 0, i, j, k;
  long map_param;
  float u1, u2, v1, v2, *ctrl_pts = NULL;

  if(a1 == nil) derec;

  /* recherche du type */
  a2 = car(a1);  a1 = cdr(a1);

  if(!isstr(a2) || !wobj) { a1 = nil; derec; }
  if((map_param = getGLvalue((char *) (a2+(2/2)))) < 0) { a1 = nil; derec; }

  /* recuperation des plages */
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  u1 = fvalue(a2);
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  u2 = fvalue(a2);
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  v1 = fvalue(a2);
  a2 = car(a1);  a1 = cdr(a1);  if(!isnum(a2)) { a1 = nil; derec; }  v2 = fvalue(a2);

  /* recuperation des points de control */
  a1 = car(a1);
  v_order = my_dummy_length(a1);
  u_order = my_dummy_length(car(a1));
  ctrl_pts = (float *) XtMalloc(v_order * u_order * 4 * sizeof(float));

  for(i = 0; i < v_order && a1 != nil; i++) {
    a2 = car(a1);  a1 = cdr(a1);
    for(j = 0; j < u_order && a2 != nil; j++) {
      a3 = car(a2); a2 = cdr(a2);
      for(k = 0; k < 4 && a3 != nil; k++) {
   a4 = car(a3); a3 = cdr(a3);
   ctrl_pts[(i * u_order * 4) + (j * 4) + k] = (float) fvalue(a4);
      }
    }
  }
  /* finalement : execute Map2... */
  glMap2f(map_param,
     u1, u2, 4, u_order,
     v1, v2, 4 * u_order, v_order,
     ctrl_pts);
  XtFree((char *)ctrl_pts);
  a1 = t;
  derec;
}

Int gl_bezier_eval_2  () /* (GLeval1 u v) SUBR2 */  gl_bezier_eval_2
{
  float u, v;
  if(!isnum(a1) || !isnum(a2) || !wobj) { a1 = nil ; derec; }
  u = (float) fvalue(a1);
  v = (float) fvalue(a2);
  glEvalCoord2f(u, v);
  a1 = t;
  derec;
```

```
}

Int gl_bezier_grid_2  () /* (Ggrid2 nu u1 u2 nv v1 v2) NSUBR */          gl_bezier_grid_2
{
  Int nu, nv;
  float u1, u2, v1, v2;
  if(!wobj) { a1 = nil ; derec; }

  a2 = car(a1); a1 = cdr(a1); nu = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); nv = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); u1 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); u2 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); v1 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); v2 = valnb(a2);
  glMapGrid2f(nu, u1, u2, nv, v1, v2);
  a1 = t;
  derec;
}

Int gl_bezier_mesh_2  () /* (GLmesh2 type p1 p2 q1 q2) NSUBR */      gl_bezier_mesh_2
{
  long type;
  Int p1, p2, q1, q2;
  if(!wobj) { a1 = nil; derec; }

  a2 = car(a1); a1 = cdr(a1); type = getGLvalue((char *) (a2 + (2/2)));
  a2 = car(a1); a1 = cdr(a1); p1 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); p2 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); q1 = valnb(a2);
  a2 = car(a1); a1 = cdr(a1); q2 = valnb(a2);

  glEvalMesh2(type, p1, p2, q1, q2);
  a1 = t;
  derec;
}

/* ***************** Formes GL predefinies ***************** */

/* ******** gestion des rectangles ******** */

/* **** Zones de l'ecran ... **** */

Int gl_rectcpy  () /* (GLrcopy x0 y0 x1 y1 x-to y-to) NSUBR */             gl_rectcpy
{
  a1 = t ;
  derec;
}

Int gl_rectzoom  () /* (GLrzoom zx zy) SUBR2 */     gl_rectzoom
{
  a1 = t ;
  derec;
}


Int gl_rect  () /* (GLrect x1 y1 x2 y2) NSUBR */       gl_rect
{
  float data[4];
  if(!wobj || !getNumArgFroma1((char *) data,4,GLXBVL_FLOAT)) {a1=nil; derec;}
  glRectf(data[0],data[1],data[2],data[3]);
  a1 = t ;
  derec;
}

Int gl_rectf  () /* (GLrectf x1 y1 x2 y2) NSUBR */     gl_rectf
{
  float data[4];
  if(!wobj || !getNumArgFroma1((char *) data,4,GLXBVL_FLOAT)) {a1=nil; derec;}
  glRectf(data[0],data[1],data[2],data[3]);
  a1 = t ;
  derec;
}

Int gl_polyg  () /* (GLpoly [(ratio)] x1 y1 z1 x2 y2 z2 ...) NSUBR */         gl_polyg
{
  GLfloat parray[100][3] ;
```

```
  GLfloat ratio = 1. ;
  Int nb_v, i;

  if(!wobj) { a1 = nil; derec; }

  a2 = car(a1), a1 = cdr(a1);
  if(islist(a2)) ratio = fvalue(car(a2));
  for(nb_v = 0; a1 != nil; nb_v++) {
    parray[nb_v][0] = fvalue(car(a1)) / ratio; a1 = cdr(a1);
    parray[nb_v][1] = fvalue(car(a1)) / ratio; a1 = cdr(a1);
    parray[nb_v][2] = fvalue(car(a1)) / ratio; a1 = cdr(a1);
  }
  glBegin(GL_POLYGON) ;
  for(i = 0 ; i < nb_v  ; ++i)
    glVertex3fv(parray[i]) ;
  glEnd() ;
  a1=t;
  derec;
}

Int gl_polym    () /* (GLpolymode mode) SUBR1 || (GLpolymode face mode) SUBR2 (OpenGl)*/
   gl_polym
{
  long mode, face;
  face = getGLvalue((char *) (a1+(2/2)));
  mode = getGLvalue((char *) (a2+(2/2)));
  if(!wobj) { a1 = nil; derec; }
  glPolygonMode(face, mode);
  a1=t;
  derec;
}

#define angle_to_m(a)  (a * M_PI / (float) 1800)
#define abs(x)         ((x) < 0 ? -(x) : (x))

Int gl_get_circle_pos  () /* (GLgetXYangulaire cx cy r angle) NSUBR */              gl_get_circle_pos
{
  float data[4], ret_x, ret_y;
  if(!wobj || !getNumArgFroma1((char *) data,4,GLXBVL_FLOAT)) {a1=nil;derec;}

  ret_x = data[0] + sin(angle_to_m(data[3])) * data[2];
  ret_y = data[1] + cos(angle_to_m(data[3])) * data[2];
  a1 = cons(cranb((Int) ret_x), cons(cranb((Int) ret_y), nil));
  derec;
}

Int gl_get_real_position  () /* (GLgetposition x y z rx ry rz sx sy sz sw) */          gl_get_real_position
{
  float data[10], x, y, z, w;
  float CM[4][4];
  Int i;

  if(!wobj) { a1 = nil; derec; }

  memset((void *)data, '\0', sizeof(float)*10);
  getNumArgFroma1((char *) data,10,GLXBVL_FLOAT);

  glPushMatrix();
  glLoadIdentity();
  glRotatef(data[3]/10.,1.0,0.0,0.0);
  glRotatef(data[4]/10.,0.0,1.0,0.0);
  glRotatef(data[5]/10.,0.0,0.0,1.0);
  for(i = 6; i < 10; i++) if(data[i] == 0.0) data[i]=1;
  glScalef(data[6]/data[9],data[7]/data[9],data[8]/data[9]);
  glTranslatef(data[0], data[1], data[2]);
  glGetFloatv(GL_MODELVIEW_MATRIX, (float *)CM);
  glPopMatrix();

  a1=cons(cranb((Int)CM[3][0]),cons(cranb((Int)CM[3][1]),cons(cranb((Int)CM[3][2]),nil)));
  derec;
}


Int gl_get_matrix  () /* (GLgetMatrix atom) SUBR1 */              gl_get_matrix
{
  GLdouble *Mat;
```

```
  struct atome *at = (struct atome *) a1;

  if(!wobj || !isatom(a1) || a1 == nil) {a1 = nil; derec;}
  if(at->obj != nil) Mat = (GLdouble *) at->obj;
  else at->obj = (Int *) (Mat = (GLdouble *) XtMalloc(sizeof(GLdouble) * 16));
  glGetDoublev(GL_MODELVIEW_MATRIX, Mat);
  derec;
}

Int gl_matrix_mult  () /* (GLmultMatrix atom) SUBR1 */          gl_matrix_mult
{
  struct atome *at = (struct atome *) a1;

  if(!wobj || !isatom(a1) || a1 == nil || at->obj == nil) { a1= nil; derec;}

  glMultMatrixd((GLdouble *)at->obj);
  derec;
}

Int gl_matrix_load  ()  /* (GLloadMatrix atom) SUBR1 */          gl_matrix_load
{
  struct atome *at = (struct atome *) a1;

  if(!wobj || !isatom(a1) || a1 == nil || at->obj == nil) {a1 = nil; derec; }

  glLoadMatrixd((GLdouble *)at->obj);
  derec;
}

Int gl_get_matrix_value  () /* (GLgetMatrixValue atom row col) SUBR3 */          gl_get_matrix_value
{
  struct atome *at = (struct atome *) a1;

  if(!wobj || !isatom(a1) || a1 == nil || at->obj == nil) { a1 = nil; derec; }
  a1 = craflo((float) (((GLdouble **)at->obj)[valnb(a2)][valnb(a3)]));
  derec;
}

Int gl_set_matrix_value  () /* (GLsetMatrixValue atom row col value) NSUBR */   gl_set_matrix_value
{
  struct atome *at = (struct atome *) car(a1);
  Int row = valnb(car(cdr(a1))),
      col = valnb(car(cdr(cdr(a1))));
  float val = valflo(car(cdr(cdr(cdr(a1)))));

  if(!wobj || !isatom(a1) || a1 == nil || at->obj == nil) { a1 = nil; derec; }
  ((GLdouble **) at->obj)[row][col] = (GLdouble) val;
  derec;
}


Int gl_free_matrix  () /* (GLfreeMatrix atom) SUBR1 */          gl_free_matrix
{
  struct atome *at = (struct atome *) a1;

  if(!wobj || !isatom(a1) || a1 == nil || at->obj == nil) {a1 = nil; derec;}
  XtFree((char *) at->obj);
  at->obj = nil;
  derec;
}

Int gl_circle  () /* (GLellipse x y rx ry b_ang e_ang) NSUBR */          gl_circle
{
  float v[3], theta;
  short data[6];
  float phi, incr;

  if(!wobj || !getNumArgFroma1((char *) data,6,GLXBVL_SHORT)) {a1=nil;derec;}
  if(data[4] == data[5]) ++data[5];
  incr = (((data[2] + abs(data[2] - data[3])) >> 2) & -4);
  if(incr < 20) incr = 20;
  else if(incr > 256) incr = 256;

  glBegin(GL_LINE_LOOP);
  for(phi = (float) data[4]; phi <= (float) data[5]; phi += (data[5] - data[4]) / incr) {
    v[0] = (float) data[0] + sin(angle_to_m(phi)) * (float) data[2];
```

```
    v[1] = (float) data[1] + cos(angle_to_m(phi)) * (float) data[3];
    v[2] = 0;
    glVertex3fv(v);
  }
  glEnd();
  a1=t;
  derec;
}

/* gestion des t-mesh : swaptmesh */
Int gl_swaptmesh  () /* (GLswaptmesh) SUBR0 */  gl_swaptmesh
{
  derec;
}

/* gestion des ombres : GOURAUD ou FLAT */
Int gl_shademodel  () /* (GLshademodel type) SUBR1 */       gl_shademodel
{
  long truc;
  if(!wobj || !isstr(a1)) derec;
  if((truc = getGLvalue((char *) (a1+(2/2)))) >= 0) {
    glShadeModel(truc) ;
    a1 = t;
  }
  else a1=nil;
  derec;
}

/* type de rendering sur les polygones */
Int gl_polymode  () /* if Irix GL (GLpolymode type) SUBR1     gl_polymode
          if Open GL (GLpolymode type mode) SUBR2 */
{
  long truc;
  if(!wobj || !isstr(a1)) derec;
  if((truc = getGLvalue((char *) (a1+(2/2)))) >= 0) {
    long truc1 ;
    if(!isstr(a2)) derec;
    if((truc1 = getGLvalue((char *) (a2+(2/2)))) >= 0) {
      glPolygonMode(truc, truc1) ;
      a1 = t ;
    }
    else a1 = nil ;
  }
  else a1=nil;
  derec;
}

/* setting du mode d'utilisation des matrices */
Int gl_mmode  () /* (GLmmode mode) SUBR1 */     gl_mmode
{
  long type;
  if(!wobj || !isstr(a1)) derec;
  if((type = getGLvalue((char *) (a1+(2/2)))) >= 0)
    {
      glMatrixMode(type);
      a1 = t;
    }
  else a1 = nil;
  derec;
}


/* acces aux matrices de GL */
Int gl_pushmatrice  () /* (GLpushmatrix) */              gl_pushmatrice
{
  if(!wobj) derec;
  glPushMatrix() ;
  a1=t;
  derec;
}



Int gl_popmatrice  () /* (GLpopmatrix) */  gl_popmatrice
{
  if(!wobj) derec;
```

```
  glPopMatrix() ;
  a1=t;
  derec;
}

lnt gl_loadidmatrix  () /* (GLloadidmatrix) */          gl_loadidmatrix
{
  if(!wobj) derec;
  glLoadIdentity() ;
  a1=t;
  derec;
}

/* Routines de transforamtion */

/* Scaling */
lnt gl_echelle  () /* (GLscale sx sy sz fsize) NSUBR */          gl_echelle
{
  float data[4];

  if(!wobj || !getNumArgFroma1((char *) data,3,GLXBVL_FLOAT)) { a1=nil; derec; }
  if(a1 != nil) data[3] = fvalue(car(a1));
  else data[3] = 1;

  glScalef(data[0] /  data[3], data[1] /  data[3], data[2] /  data[3]);
  a1 = t;
  derec;
}

/* Translation */
lnt gl_translat  () /* (GLtranslate x y z) SUBR3 (sx, sy et sz entiers) */   gl_translat
{
  float x, y, z;
  if(!wobj || a1 == nil || a2 == nil || a3 == nil) { a1=nil; derec; }

  x = (float) fvalue(a1); y = (float) fvalue(a2); z = (float) fvalue(a3);
  glTranslatef(x, y, z) ;
  a1=t;
  derec;
}

/* Rotation */
lnt gl_rotate  () /* (GLrotate angle axe) SUBR2 (angle est en disieme de degre) */          gl_rotate
{
  char axis;

  if(!wobj || !isnum(a1)) {a1=nil; derec; }
  if(isstr(a2)) {
    axis = isstr(a2) ? *(char *)(a2+(2/2)) : 'X';
    if(axis > 'Z') axis -= 'a' - 'A';
    if(axis < 'X' || axis > 'Z') {a1=nil; derec;}
  }
  else if(isnum(a2) && isnum(a3)) axis = 0;

  if(axis != 0)
    glRotatef((float) fvalue(a1) / 10., 1.0 * !(axis-'X'), 1.0 * !(axis-'Y'), 1.0 * !(axis-'Z'));
  else {
    glRotatef((float) fvalue(a1) / 10., 1.0, 0.0, 0.0);
    glRotatef((float) fvalue(a2) / 10., 0.0, 1.0, 0.0);
    glRotatef((float) fvalue(a3) / 10., 0.0, 0.0, 1.0);
  }

  a1=t;
  derec;
}

lnt gl_rotate_axis  () /* (GLrotateAxis anglex angley anglez) */          gl_rotate_axis
{
  float RM[4][4], cx, sx, cy, sy, cz, sz;

#ifndef PI
#define PI 3.1592654
#endif
#define FromDeg(x) (((x)/3600.0)*PI)

  if(!wobj || !isnum(a1) || !isnum(a2) || !isnum(a3)) {a1=nil; derec; }
```

```
  cx = (float) cos(FromDeg((float)fvalue(a1)));  sx = (float) sin(FromDeg((float)fvalue(a1)));
  cy = (float) cos(FromDeg((float)fvalue(a2)));  sy = (float) sin(FromDeg((float)fvalue(a2)));
  cz = (float) cos(FromDeg((float)fvalue(a3)));  sz = (float) sin(FromDeg((float)fvalue(a3)));

  RM[0][0] = cy * cz;  RM[0][1] = -1 * sz; RM[0][2] = sy;          RM[0][3] = 0.0;
  RM[1][0] = sz;       RM[1][1] = cx * cz; RM[1][2] = -1 * sx * cz; RM[1][3] = 0.0;
  RM[2][0] = -1 * sy;  RM[2][1] = sx;      RM[2][2] = cx * cy;      RM[2][3] = 0.0;

  RM[3][0] = 0.0;  RM[3][1] = 0.0; RM[3][2] = 0.0; RM[3][3] = 1.0;

  glMultMatrixf((float *)RM);

  a1=t;
  derec;
}
/* ********************** */
/* Gestion des caracteres */
/* ********************** */
Int gl_string  () /* (GLstring x y z str) NSUBR */       gl_string
{
  if(!wobj) derec;
  glhstr(); /* utilisation des Hershey...: pas de gl string en OPEN GL */
}

/* Implementation des NURBS */

/* Modification des donnees de la fenetre pour les nurbs */

Int gl_set_nurbs  () /* if Irix gl(GLsetnurbs "DISPLAY"|"TOLERANCE" valeur) SUBR2          gl_set_nurbs
                        if Open GL(GLsetnurbs nurbs property valeur) SUBR3 */
{
  GLUnurbsObj *nObj ;
  GLenum property ;
  GLfloat value ;
  Int index ;

  if(!wobj || !isnum(a1) && !isstr(a2) && !isnum(a3) && !isflo(a3)) {a1 = nil ; derec ;}
  nObj =  (GLUnurbsObj *) valnb(a1) ;
  property = getGLvalue(STR_FROM_STRLSP(a2)) ;
  value = (float) fvalue(a3) ;
  gluNurbsProperty(nObj, property, value) ;
  wobj->nurbs_data[NURBINDEX(property)] = value;
  wobj->mode |= property ;

  a1 = t ;
  derec;
}

/* recuperation du type d'une nurbs */
static long
getNurbsType  (str)  getNurbsType
char *str;
{
  return getGLvalue(str);
}



/* sous fonction de recuperations des valeurs pour nurbs curves et surfaces */
static Int
getNurbsData  (result, dim)      getNurbsData
float *result; /* table des resultats */
Int *dim;       /* dimension (nombre de poInt resultant par donnee) */
{
  Int *tmp = NULL;
  Int nb_lus, indice = 0;

  for(nb_lus = 0; tmp != nil; nb_lus++, a1=cdr(a1)) {
    tmp = car(a1);
    if(tmp == nil) break;
    if(islist(tmp)) { /* liste de sous listes */
      result[indice++] = (float) fvalue(car(tmp)); tmp = cdr(tmp);
      result[indice++] = (float) fvalue(car(tmp)); tmp = cdr(tmp);
      if(tmp != nil) {
   result[indice++] = (float) fvalue(car(tmp)); tmp = cdr(tmp);
```

```
   if(tmp != nil)          result[indice++] = (float) fvalue(car(tmp)), *dim=4;
   else *dim=3,++indice;
     }
     else *dim=2,indice += 2;
    }
    else result[indice++] = (float) fvalue(tmp);
  }
  return nb_lus;
}

/* les donnees fournies en argument sont :
   a1 = liste des points de controle (le nombre de donnees par poInt donnant le type de la courbe)
        example : ((x0 y0 z0) (x1 y1 z1) ... (xn yn zn))
   a2 = liste des points (nombre de points + dimension de la courbe) :
        donne la liste des poInt et la dimension de la courbe.
*/
Int gl_nurbs_curve  () /* if Irix GL(GLnurbs liste-de-points-de-controle liste-de-points type) SUBR3
   gl_nurbs_curve
                if Open GL(GLnurbs nurbs liste-de-points-de-controle liste-de-points type) NSUBR */
{
  static float control_point[25][4];
  static float points[29];  /* maximum = 25 (points) + 4 (dimension) */
  Int ctrl_pts_nb = 0, dim = 2, pts_nb = 0, type;
  GLUnurbsObj *nObj ;
  if(!wobj || !isnum(car(a1)) || isnotlist(car(cdr(a1))) || isnotlist(car(cdr(cdr(a1))))) {a1=nil; derec;}
  nObj =  (GLUnurbsObj *) valnb(a1) ;
  a1 = cdr(a1) ; ctrl_pts_nb = getNurbsData(&control_point[0][0], &dim);
  a1 = cdr(a1) ; pts_nb = getNumArgFroma1((char *)&points[0],0,GLXBVL_FLOAT);
  a1 = cdr(a1) ; type = getGLvalue(STR_FROM_STRLSP(car(a1))) ;
  gluNurbsCurve(nObj, pts_nb, (float *)points, 4*sizeof(float), (float *)control_point, dim, type);
  a1=t;
  derec;
}

/* Definition des surfaces utilisant les nurbs
 * a1 : points de control (liste de sous liste de sous listes de points 1 ou 2)
 * a2 : liste de n_s sous listes contenant n_t sous listes de k points
 *      avec n_s = nombre de points en s
 *           n_t = nombre de points en t
 *             k = nombre de coordonnees par points
 * a3 : type de la surface (string)
 */
Int gl_nurbs_surface  () /* (GLsurface pts ctrl_pts type) SUBR3  */       gl_nurbs_surface
{
  static float control_points[1024];
  static float points[2][30];
  long s_nk, t_nk, nb_s, nb_t, s_order, t_order, type, indice;
  Int is_onek = 0, dim, *tmp;

  if(wobj && isustr(a3)) type = getNurbsType((char *) (a3+(2/2)));
  else {
    a1=nil; derec;
  }
    tmp = car(cdr(a1));
  a1 = car(a1);
  s_nk = getNumArgFroma1((char *) &points[0][0], 0, GLXBVL_FLOAT);
  if(tmp != nil) {
    a1 = tmp;
    t_nk = getNumArgFroma1((char *)&points[1][0], 0, GLXBVL_FLOAT);
    is_onek = 1;
  }
  else t_nk = s_nk;
  a3 = a2;
  indice = 0;
  nb_s = 0;
  while(a3 != nil) {
    a1 = car(a3);
    nb_t = getNurbsData(&control_points[indice], &dim);
    indice += 4*nb_t;
    ++nb_s;
    a3 = cdr(a3);
  }
    s_order = s_nk - nb_s;
  t_order = t_nk - nb_t;

  a1 = t;
```

```
  derec;
}

#endif
```

Annexe III.6 Fonctionnalités (2) : glLights.c

```
/*
 * glLights.c
 * definition des fonctions permettant l'utilisation des lumieres
 * et textures de GL
 */

#if (OPEN_GL | MESA_GL)

#include <math.h>

#include "vlisp.h"
#include "glXbvl.h"

#ifndef STDC_HEADERS
extern long getGLvalue  ();      getGLvalue
#else
extern long getGLvalue  (char *);            getGLvalue
#endif


/* Dans Open GL, la definition d'objet (materiaux, lumiere ou model n'existe pas
   => creation de GLlight, GLmaterial et GLmodel
*/

static float values[4] ;

static void
getProp  (GLenum *prop)        getProp
{
  Int i;
  if(!wobj || !isstr(a1)) {a1=nil; return; }
  *prop = getGLvalue(STR_FROM_STRLSP(a1));
  a1 = t;
  if(isnum(a2) || isflo(a2)) { values[0] = (float) valflo(a3); a2 = (Int *)1; }
  else if(islist(a2)) {
    for(i = 0; i < 4 && !isnil(a2); i++) {
      values[i] = (float) valflo(car(a2));
      a2 = cdr(a2);
    }
    a2 = (Int *)2;
  }
  else a1=nil;
}

Int gl_lumiere  () /* (GLlight Light Prop Value) : specification de la propriete Prop pour la lumiere Light...:
SUBR3 */              gl_lumiere
{
  GLenum lgt, pnam;
  if(!wobj || !isstr(a1)) {a1=nil; derec;}
  lgt = getGLvalue(STR_FROM_STRLSP(a1));
  a1 = a2;  a2 = a3;
  getProp(&pnam);
  if(a1 == t) {
    if((Int) a2 == 1) { glLightf(lgt, pnam, values[0]); a1=craflo((double)values[0]); }
    else       { glLightfv(lgt, pnam, &values[0]);
                 a1 = cons(craflo((double)values[0]),
                           cons(craflo((double)values[1]),
                                cons(craflo((double)values[2]),
                                     craflo((double)values[3]))));
               }
  }
  derec;
}

Int gl_model  () /* (GLmodel prop value) SUBR2 */              gl_model
{
  GLenum prop;
  if(!wobj) derec;
  getProp(&prop);
  if(a1 == t) {
    if((Int) a2 == 1) glLightModelf(prop, values[0]);
    else       glLightModelfv(prop, &values[0]);
```

```
 }
 derec;
}

Int gl_materiel () /* (GLmaterial type prop value) SUBR3 */   gl_materiel
{
 GLenum lgt, pnam;
 if(!wobj || !isstr(a1)) {a1=nil; derec;}
 lgt = getGLvalue(STR_FROM_STRLSP(a1));
 a1 = a2;  a2 = a3;
 getProp(&pnam) ;
 if(a1 == t) {
   if((Int) a2 == 1) { glMaterialf(lgt, pnam, values[0]); a1=craflo((double)values[0]); }
   else       { glMaterialfv(lgt, pnam, &values[0]);
                a1 = cons(craflo((double)values[0]),
                          cons(craflo((double)values[1]),
                               cons(craflo((double)values[2]),
                                    craflo((double)values[3]))));
              }
 }
 derec;
}

Int gl_permet () /* (GLenable cap) SUBR1 */          gl_permet
{
 if(!wobj || !isstr(a1)) {a1=nil; derec;}
 glEnable((GLenum) getGLvalue(STR_FROM_STRLSP(a1)));
 redessine();
 a1= t;
 derec;
}

Int gl_disable () /* (GLdisable cap) SUBR1 */        gl_disable
{
 if(!wobj || !isstr(a1)) {a1=nil; derec;}
 glDisable((GLenum) getGLvalue(STR_FROM_STRLSP(a1)));
 redessine();
 a1= t;
 derec;
}

Int gl_is_enable () /* (GLisenable cap) SUBR1 */   gl_is_enable
{
 if(!wobj || !isstr(a1)) {a1=nil; derec;}
 if(glIsEnabled((GLenum) getGLvalue(STR_FROM_STRLSP(a1)))) a1 = t;
 else a1 = nil;
 derec;
}

Int   gl_fonction_blend   () /* (GLblend Source-Factor Destination-Factor) SUBR2 */
  gl_fonction_blend
{
 if(!wobj || !isstr(a1) || !isstr(a2)) { a1 = nil; derec; }
 glBlendFunc((GLenum) getGLvalue(STR_FROM_STRLSP(a1)),
       (GLenum) getGLvalue(STR_FROM_STRLSP(a2)));
 a1 = t;
 derec;
}

#endif /* (OPEN_GL | MESA_GL) */
```

Annexe III.7 Fonctionnalités (3) : glFonts.c

```
/*
 * Definitions des fonctions d'interface entre la librairie de fonts
 * de caracteres Hershey et gl-xbvl...
 */
#if (OPEN_GL | MESA_GL)

#include "vlisp.h"
#include "glXbvl.h"
/* #include <device.h> */
#include <math.h>
```

```
/* fixe le font */
Int glhfont  () /* (GLfont name) SUBR1 */              glhfont
{
  char *f_name;

  if(isatom(a1))     f_name = STR_FROM_STRLSP(((ATOMEPTR) a1)->pn);
  else if(isstr(a1)) f_name = STR_FROM_STRLSP(a1);
  else {a1=nil; derec;}
  a1=t;
  hfont(f_name);
  derec;
}

/* fixe le path de recherche des font */
Int glhpath  () /* (GLfontpath path) SUBR1 */          glhpath
{
  char *f_name;

  if(isatom(a1))     f_name = STR_FROM_STRLSP(((ATOMEPTR) a1)->pn);
  else if(isstr(a1)) f_name = STR_FROM_STRLSP(a1);
  else {a1=nil; derec;}
  a1=t;
  hsetpath(f_name);
  derec;
}

/* positionne aux coordonnees donnees dans a1 */
static Int moveToa1  ()              moveToa1
{
  short data[3];

  if(!getNumArgFroma1((char *) data,3,GLXBVL_SHORT)) return 0;
  glPushMatrix();
  glTranslatef((float) data[0], (float) data[1], (float) data[2]);
  return 1;
}

/* dessine un caractere */
Int glhdrawchar  () /* (GLchar x y z char) */          glhdrawchar
{
  char c = 0;
  if(!wobj || !moveToa1()) {a1=nil; derec;}
  a1=car(a1);
  if(isatom(a1))     c = *((((ATOMEPTR) a1)->pn)+(2/2));
  else if(isstr(a1)) c = *(a1+(2/2));
  a1=nil;
  if(c) hdrawchar(c), a1=t;
  glPopMatrix();
  derec;
}

/* dessine une chaine de caracteres */
Int glhstr  () /* (GLstr x y z str) NSUBR */          glhstr
{
  char *str = NULL;
  if(!wobj || !moveToa1()) {a1=nil; derec;}
  a1=car(a1);
  if(isatom(a1))     str = STR_FROM_STRLSP(((ATOMEPTR) a1)->pn);
  else if(isstr(a1)) str = STR_FROM_STRLSP(a1);
  a1=nil;
  if(str) hcharstr(str), a1=t;
  glPopMatrix();
  derec;
}

/* dessine une chaine incluse dans une boite */
Int glhboxtext  () /* (GLboxtext x y w h str) NSUBR */          glhboxtext
{
  char *str;
  double data[4];
  if(!wobj || !getNumArgFroma1(data,4,GLXBVL_DOUBLE)) {a1=nil; derec;}
  a1=car(a1);
  if(isatom(a1))     str = STR_FROM_STRLSP(((ATOMEPTR) a1)->pn);
  else if(isstr(a1)) str = STR_FROM_STRLSP(a1);
```

```
  else {a1=nil; derec;}
  a1=t;
  hboxtext(data[0], data[1],  data[2],  data[3], (char *) str);
  derec;
}

/* fixe l'echelle du text pour qu'il soit inclus dans une boite */
Int glhboxfit  () /* (GLboxfit w h n) SUBR3 */          glhboxfit
{
  if(!wobj) derec;
  hboxfit( valflo(a1), valflo(a2),valnb(a3));
  a1=t;
  derec;
}

/* fixe l'angle d'impression des caracteres */
Int glhtextangl  () /* (GLtextangl a) SUBR 1 */        glhtextangl
{
  if(!wobj) derec;
  htextang(valflo(a1));
  a1=t;
  derec;
}

/* taille des caracteres */
Int glhtextsize  () /* (GLtextsize w h [ratio]) SUBR3 */          glhtextsize
{
  Int r = (a3 != nil) ? valnb(a3) : 1;
  if(!wobj) derec;
  htextsize(valnb(a1) / (float) r, valnb(a2) / (float) r);
  a1=t;
  derec;
}

/* text de taille fixe ... */
Int glhfixedwidth  () /* (GLfixedwidth t|nil) SUBR1 */          glhfixedwidth
{
  if(!wobj) derec;
  hfixedwidth(a1==t);
  a1=t;
  derec;
}

Int glhcentertext  () /* (GLcentertext t|nil) SUBR1 */          glhcentertext
{
  if(!wobj) derec;
  hcentertext(a1==t);
  a1=t;
  derec;
}

Int glhrigthjustify  () /* (GLrigthjustify t|nil) SUBR1 */          glhrigthjustify
{
  if(!wobj) derec;
  hrightjustify(a1==t);
  a1=t;
  derec;
}

Int glhleftjustify  () /* (GLleftjustify t|nil) SUBR1 */   glhleftjustify
{
  if(!wobj) derec;
  hleftjustify(a1==t);
  a1=t;
  derec;
}
#endif
```

Annexe III.8 Fonctionnalités (4) : glTexture.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include "libtk/tk.h"
#include "glXbvl.h"

/* définition des données liées à une image comme texture */
Int gl_def_texture  () gl_def_texture
{
  TK_RGBImageRec *image;
  char is_rgba;
  pop_cast(image, (TK_RGBImageRec *)); pop(is_rgba);

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, image->sizeX, image->sizeY, 0,
               (is_rgba ? GL_RGBA : GL_RGB), GL_UNSIGNED_BYTE, image->data);
  free_image_data(image);
  image->data = NULL;
  a1 = t;
  derec;
}


#ifdef HAVE_LIBTIFF
#include <tiffio.h>

/* possibilité de définission d'une texture à partir d'un fichier tiff */

extern TK_RGBImageRec *LoadTiffImage  (       *LoadTiffImage
#ifdef STDC_HEADERS
                                       char *f_name
#endif
                                       );


/* (GLtiffTexImage f_name) */
Int gl_def_tiff_texture  ()       gl_def_tiff_texture
{
  char *f_name;
  TK_RGBImageRec *image;

  if(!isstr(a1)) { a1 = nil; derec; }
  f_name = STR_FROM_STRLSP(a1);
  if(!(image = LoadTiffImage(f_name)) || !image->data) { a1 = nil; derec; }

  push(1); push(image);
  pc=gl_def_texture;
}
#else /* LIBTIFF */
Int gl_def_tiff_texture  ()       gl_def_tiff_texture
{
  a1 = nil;
  derec;
}
#endif

/* définition de textures à partir de fichiers rgb */
/* (GLrgbTexImage f_name) */
Int gl_def_rgb_texture  ()       gl_def_rgb_texture
{
  char *f_name;
  TK_RGBImageRec *image;

  if(!isstr(a1)) { a1 = nil; derec; }
  f_name = STR_FROM_STRLSP(a1);
  if(!(image = tkRGBImageLoad(f_name))) { a1 = nil; derec; }

  push(0); push(image);
  pc=gl_def_texture;
}

/* définition de textures à partir de données (liste de valeurs) */
/* (GLdataTexImage largeur hauteur liste_de_donnée)  SUBR 3 */
Int gl_def_data_texture  ()       gl_def_data_texture
```

```
{
  TK_RGBImageRec image;

  if(!isnum(a1) || !isnum(a2) || !islist(a3)) { a1 = nil; derec; }
  image.sizeX = valnb(a1);
  image.sizeY = valnb(a2);
  image.data = alloc_image_data(image.sizeX, image.sizeY);
  a1 = a3;
  if(!getNumArgFroma1(image.data, image.sizeX * image.sizeY * 3, GLXBVL_CHAR)) {
    free_image_data(&image);
    a1 = nil; derec;
  }
  push(0); push(&image);
  pc=gl_def_texture;
}


/* autres définitions de la librairie OpenGL à propos des textures */

/* définition des coordonnées : (GLtexCoord s [t])  SUBR2 */

Int gl_texture_coord  ()          gl_texture_coord
{
  if(!isnum(a1)) { a1 = nil; derec; }
  if(isnum(a2)) glTexCoord2f((float)fvalue(a1), (float)fvalue(a2));
  else          glTexCoord1f((float)fvalue(a1));
  a1 = t;
  derec;
}

/* définition des paramètres des textures */

/*  (GLtexEnv "GL_TEXTURE_ENV_MODE"|"GL_TEXTURE_ENV_COLOR" data) SUBR2 */
Int gl_texture_env  ()gl_texture_env
{
  GLenum pname, param;
  float color[4];

  if((pname = getGLvalue(STR_FROM_STRLSP(a1))) == -1) { a1 = nil; derec; }

  if(isstr(a2)) {
    if((param = getGLvalue(STR_FROM_STRLSP(a2))) == -1) { a1 = nil; derec; }
    glTexEnvi(GL_TEXTURE_ENV, pname, param);
    a1 = t;
  }
  else if(islist(a2) && (a1 = a2) &&
     getNumArgFroma1(color, 4, GLXBVL_FLOAT)) {
    glTexEnvfv(GL_TEXTURE_ENV, pname, color);
    a1 = t;
  }
  else a1 = nil;
  derec;
}

/* (GLtexParameter type_de_texture param valeur) SUBR3 */
Int gl_texture_parameter  ()    gl_texture_parameter
{
  GLenum dest, pname, param;
  float color[4];

  if((dest = getGLvalue(STR_FROM_STRLSP(a1))) == -1) { a1 = nil; derec; }
  if((pname = getGLvalue(STR_FROM_STRLSP(a2))) == -1) { a1 = nil; derec; }

  if(isstr(a3)) {
    if((param = getGLvalue(STR_FROM_STRLSP(a3))) == -1) { a1 = nil; derec; }
    glTexParameteri(dest, pname, param);
    a1 = t;
  }
  else if(islist(a3) && (a1 = a3) &&
     getNumArgFroma1(color, 4, GLXBVL_FLOAT)) {
    glTexParameterfv(dest, pname, color);
    a1 = t;
  }
  else a1 = nil;
  derec;
}
```

```
/* (GLtexGen coord pname param) */
Int gl_texture_gen  ()gl_texture_gen
{
  GLenum dest, pname, param;
  float color[4];

  if((dest = getGLvalue(STR_FROM_STRLSP(a1))) == -1) { a1 = nil; derec; }
  if((pname = getGLvalue(STR_FROM_STRLSP(a2))) == -1) { a1 = nil; derec; }

  if(isstr(a3)) {
    if((param = getGLvalue(STR_FROM_STRLSP(a3))) == -1) { a1 = nil; derec; }
    glTexGeni(dest, pname, param);
    a1 = t;
  }
  else if(islist(a3) && (a1 = a3) &&
      getNumArgFroma1(color, 4, GLXBVL_FLOAT)) {
    glTexGenfv(dest, pname, color);
    a1 = t;
  }
  else a1 = nil;
  derec;
}
```

Annexe III.9 Fonctionnalités (5) : gluFuncs.c

```
#if (OPEN_GL | MESA_GL)
#include <math.h>
#include "glXbvl.h"
#include "vlisp.h"

/* Implementation des librairies GLU */
/* D.Ploix & O.Blanc, 1995 */

/* reminissance de irix gl */
gl_sphere_mode () /* (GLsphmode "atribute" "value") SUBR2 */        gl_sphere_mode
{
  a1 = t ;
  derec;
}

/* Les GLU suivantes utilisent les Quadratics, leurs proprietes sont :
   - Normal      = Type de normale (GLU_NONE, GLU_FLAT, GLU_SMOOTH)
   - DrawStyle   = Style de dessin (GLU_FILL, GLU_LINE, GLU_SILHOUETTE, GLU_POINT)
   - Texture     = Utilisation de la texture (GLU_TRUE, GLU_FALSE)
   - Orientation = Orientation Interieur/Exterieur (GLU_INSIDE, GLU_OUTSIDE)
*/
/* Valeurs par defaut : */

#define GLU_NORMAL_VALUE     0
#define GLU_DRAWSTYLE_VALUE  1
#define GLU_ORIENTATION_VALUE 2
#define GLU_TEXTURE_VALUE    3

typedef GLUquadricObj *GLUquadricObjPtr;

static GLenum GLU_Values[4] = {GLU_SMOOTH, GLU_LINE, GLU_OUTSIDE, (Int) GL_FALSE} ;
static GLUquadricObjPtr glu_quadric_object = NULL;

/* creation d'un quadratic */
static GLUquadricObjPtr
glu_newquadratic ()glu_newquadratic
{
  GLUquadricObjPtr ret = gluNewQuadric();
  gluQuadricNormals    (ret, GLU_SMOOTH);
  gluQuadricDrawStyle  (ret, GLU_LINE);
  gluQuadricTexture    (ret, GLU_FALSE);
  gluQuadricOrientation(ret, GLU_OUTSIDE);
  return ret;
}

/* destruction d'un quadratic */
Int glu_delquad () /* (GLUdelquad [adr]) */            glu_delquad
{
  if(!wobj || !isnum(a1)) { a1=nil; derec;}
  gluDeleteQuadric((GLUquadricObjPtr) valnb(a1));
  a1 = t;
  derec;
}

/* Changement des valeurs :
   what = "normal" || "drawstyle" || "texture" || "orientation"
*/
Int glu_setvalue () /* (GLUsetvalue what value) SUBR2 */    glu_setvalue
{
  if(!wobj || !isstr(a1) || !isstr(a2)) {a1 = nil; derec;}
  if(!glu_quadric_object) glu_quadric_object = glu_newquadratic();
  if(!strcmp(STR_FROM_STRLSP(a1), "normal"))        gluQuadricNormals    (glu_quadric_object, getGLva-
lue(STR_FROM_STRL
P (a2)));    P
  if(!strcmp(STR_FROM_STRLSP(a1), "drawstyle"))    gluQuadricDrawStyle  (glu_quadric_object, getGLva-
lue(STR_FROM_STRL
P (a2)));    P
  if(!strcmp(STR_FROM_STRLSP(a1), "texture"))       gluQuadricTexture    (glu_quadric_object, getGLva-
lue(STR_FROM_STRL
```

```
P (a2)));    P
  if(!strcmp(STR_FROM_STRLSP(a1), "orientation"))    gluQuadricOrientation(glu_quadric_object, getGLva-
lue(STR_FROM_STRL
P (a2)));    P
  a1=t;
  derec;
}



/* Dessin de sphere */
Int gl_sphere () /* si Open GL :(GLsphere radius [slices] [stacks]) : NSUBR */      gl_sphere
{
  GLdouble radius ;
  GLint slices = 16, stack = 16 ;
  if(!wobj || !isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  if(!glu_quadric_object) glu_quadric_object = glu_newquadratic();
  radius = (GLdouble) fvalue(car(a1)) ;
  a1 = cdr(a1) ;
  if(isnotnil(a1) && isnum(car(a1))) {
    slices = valnb(car(a1)) ;
    a1 = cdr(a1) ;
    if(isnotnil(a1) && isnum(car(a1)))
      stack = valnb(car(a1)) ;
  }
  gluSphere(glu_quadric_object, radius, slices, stack) ;
  a1 = t;
  derec;
}

/* dessin d'un cylindre */
Int gl_cylinder () /*  si Open GL :(GLcylindre baseRadius topRadius height [slices] [stacks]) : NSUBR */
   gl_cylinder
{
  GLdouble baseRadius, topRadius, height ;
  GLint slices = 16, stack = 16 ;
  GLenum drawStyle = GLU_LINE ;
  if(!wobj || !isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  baseRadius = (GLdouble) fvalue(car(a1)) ;
  a1 = cdr(a1) ;
  if(!isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  topRadius = (GLdouble) fvalue(car(a1)) ;
  a1 = cdr(a1) ;
  if(!isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  height = (GLdouble) fvalue(car(a1)) ;
  if(!glu_quadric_object) glu_quadric_object = glu_newquadratic();
  a1 = cdr(a1) ;
  if(isnotnil(a1) && isnum(car(a1))) {
    slices = valnb(car(a1)) ;
    a1 = cdr(a1) ;
    if(isnotnil(a1) && isnum(car(a1)))
      stack = valnb(car(a1)) ;
  }
  gluCylinder(glu_quadric_object, baseRadius, topRadius, height, slices, stack) ;
  a1 = t;
  derec;
}

Int gl_disk () /*  si Open GL :(GLdisk inRadius outRadius [slices] [loops]) : NSUBR */          gl_disk
{
  GLdouble inRadius, outRadius ;
  GLint slices = 16, loops = 16 ;
  GLenum drawStyle = GLU_LINE ;
  if(!wobj || !isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  inRadius = (GLdouble) fvalue(car(a1)) ;
  a1 = cdr(a1) ;
  if(!isnum(car(a1)) && !isflo(car(a1))) {a1 = nil ; derec ;}
  outRadius = (GLdouble) fvalue(car(a1)) ;
  a1 = cdr(a1) ;
  if(!glu_quadric_object) glu_quadric_object = glu_newquadratic();
  a1 = cdr(a1) ;
  if(isnotnil(a1) && isnum(car(a1))) {
    slices = valnb(car(a1)) ;
    a1 = cdr(a1) ;
    if(isnotnil(a1) && isnum(car(a1)))
      loops = valnb(car(a1)) ;
```

```
 }
 gluDisk(glu_quadric_object, inRadius, outRadius, slices, loops) ;
 a1 = t;
 derec;
}

#if OPEN_GL
# include "glXbvlaux.h"
#endif

Int gl_sphere_wire () /* si Open GL :(GLsphereWire inRadius) : SUBR1 */ gl_sphere_wire
{
 if(!wobj || !isnum(a1) && !isflo(a1)) {a1 = nil ; derec ;}
 auxWireSphere((GLdouble)fvalue(a1)) ;
 a1 = t ; derec ;
}

Int gl_cube_wire () /* si Open GL :(GLcubeWire size) : SUBR1 */ gl_cube_wire
{
 if(!wobj || !isnum(a1) && !isflo(a1)) {a1 = nil ; derec ;}
 auxWireCube((GLdouble)fvalue(a1)) ;
 a1 = t ; derec ;
}

Int gl_box_wire () /* si Open GL :(GLboxWire width height depth) : SUBR3 */ gl_box_wire
{
 if(!wobj || (!(isnum(a1) || isflo(a1)) ||
       !(isnum(a2) || isflo(a2)) ||
       !(isnum(a3) || isflo(a3))))  {a1 = nil ; derec ;}
 auxWireBox((GLdouble)fvalue(a1), (GLdouble)fvalue(a2), (GLdouble)fvalue(a3)) ;
 a1 = t ; derec ;
}

Int gl_torus_wire () /* si Open GL :(GLtorusWire innerRadius outerRadius ) : SUBR2 */ gl_torus_wire
{
 if(!wobj || (!(isnum(a1) || isflo(a1)) ||
       !(isnum(a2) || isflo(a2))))  {a1 = nil ; derec ;}
 auxWireTorus((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
 a1 = t ; derec ;
}

Int gl_cylinder_wire () /* si Open GL :(GLcylinderWire radius height ) : SUBR2 */
  gl_cylinder_wire
{
 if(!wobj || !(isnum(a1) || isflo(a1)) ||
    !(isnum(a2) || isflo(a2)))  {a1 = nil ; derec ;}
 auxWireCylinder((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
 a1 = t ; derec ;
}


Int gl_icosahedra_wire () /* si Open GL :(GLicosahedraWire radius ) : SUBR1 */
  gl_icosahedra_wire
{
 if(!wobj || (!isnum(a1) && !isflo(a1)))  {a1 = nil ; derec ;}
 auxWireIcosahedron((GLdouble)fvalue(a1)) ;
 a1 = t ; derec ;
}


Int gl_octahedra_wire () /* si Open GL :(GLoctahedraWire radius ) : SUBR1 */ gl_octahedra_wire
{
 if(!wobj || (!isnum(a1) && !isflo(a1)))  {a1 = nil ; derec ;}
 auxWireOctahedron((GLdouble)fvalue(a1)) ;
 a1 = t ; derec ;
}


Int gl_tetrahedra_wire () /* si Open GL :(GLtetrahedraWire radius ) : SUBR1 */
  gl_tetrahedra_wire
{
 if(!wobj || (!isnum(a1) && !isflo(a1)))  {a1 = nil ; derec ;}
 auxWireTetrahedron((GLdouble)fvalue(a1)) ;
 a1 = t ; derec ;
}
```

```
lnt gl_dodecahedra_wire () /* si Open GL :(GLdodecahedraWire radius ) : SUBR1 */
  gl_dodecahedra_wire
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxWireDodecahedron((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}


lnt gl_cone_wire () /* si Open GL :(GLconeWire base height ) : SUBR2 */ gl_cone_wire
{
 if(!wobj || (!isnum(a1) && !isflo(a1) &&
       !isnum(a2) && !isflo(a2))) {a1 = nil ; derec ;}
 auxWireCone((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
a1 = t ; derec ;
}


lnt gl_sphere_solid () /* si Open GL :(GLsphereSolid inRadius) : SUBR1 */ gl_sphere_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidSphere((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}

lnt gl_cube_solid () /* si Open GL :(GLcubeSolid size) : SUBR1 */ gl_cube_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidCube((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}

lnt gl_box_solid () /* si Open GL :(GLboxSolid width height depth) : SUBR3 */ gl_box_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1) &&
       !isnum(a2) && !isflo(a2) &&
       !isnum(a3) && !isflo(a3))) {a1 = nil ; derec ;}
 auxSolidBox((GLdouble)fvalue(a1), (GLdouble)fvalue(a2), (GLdouble)fvalue(a3)) ;
a1 = t ; derec ;
}

lnt gl_torus_solid () /* si Open GL :(GLtorusSolid innerRadius outerRadius ) : SUBR2 */ gl_torus_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1) &&
       !isnum(a2) && !isflo(a2))) {a1 = nil ; derec ;}
 auxSolidTorus((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
a1 = t ; derec ;
}

lnt gl_cylinder_solid () /* si Open GL :(GLcylinderSolid radius height ) : SUBR2 */
  gl_cylinder_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1) &&
       !isnum(a2) && !isflo(a2))) {a1 = nil ; derec ;}
 auxSolidCylinder((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
a1 = t ; derec ;
}


lnt gl_icosahedra_solid () /* si Open GL :(GLicosahedraSolid radius ) : SUBR1 */
  gl_icosahedra_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidIcosahedron((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}


lnt gl_octahedra_solid () /* si Open GL :(GLoctahedraSolid radius ) : SUBR1 */
  gl_octahedra_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidOctahedron((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
```

```
}


Int gl_tetrahedra_solid () /* si Open GL :(GLtetrahedraSolid radius ) : SUBR1 */
  gl_tetrahedra_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidTetrahedron((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}


Int gl_dodecahedra_solid () /* si Open GL :(GLdodecahedraSolid radius ) : SUBR1 */
  gl_dodecahedra_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1))) {a1 = nil ; derec ;}
 auxSolidDodecahedron((GLdouble)fvalue(a1)) ;
a1 = t ; derec ;
}



Int gl_cone_solid () /* si Open GL :(GLconeSolid base height ) : SUBR2 */ gl_cone_solid
{
 if(!wobj || (!isnum(a1) && !isflo(a1) &&
       !isnum(a2) && !isflo(a2))) {a1 = nil ; derec ;}
 auxSolidCone((GLdouble)fvalue(a1), (GLdouble)fvalue(a2)) ;
a1 = t ; derec ;
}

#endif
```

Annexe III.10 Fonctionnalités (6) : glImages.c

```
#if (OPEN_GL | MESA_GL)
/* Ce fichier contiend les définition permettant de lire ou enregistrer des images
   extérieures, ceci en utilisant la librairie tiff pour les fichiers de format tiff.
   On définira aussi ici la possibilité d'importer des bitmaps pour leur utilisation
   en tant que telle, comme pattern stipple pour les polygones ou comme texture.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include "libtk/tk.h"
#include "glXbvl.h"

#define RGB_MAGIC 0xda01

/* Affichage d'une image lue dans un fichier */

static Int display_image_file ()display_image_file
{
  TK_RGBImageRec *image;
  float zx, zy;
  short pos[3];
  char is_rgba;

  pop(is_rgba);
  pop_cast(image,(TK_RGBImageRec *));
  pop(pos[0]); pop(pos[1]);
  pop(pos[2]); pop(zx); pop(zy);

  /* récupération de la taille */
  glRasterPos3sv(pos);
  glPixelZoom(zx, zy);
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glDrawPixels(image->sizeX, image->sizeY, ((is_rgba) ? GL_RGBA : GL_RGB), GL_UNSIGNED_BYTE,
image->data);
  free_image_data(image);
  a1 = t;
  derec;
}

/* ************************** */
/* gestion des fichiers bitmap */
/* ************************** */

/* chargement d'un fichier bitmap comme définition de style
   de remplissage des polygones. Le bitmap doit être de taille < 32x32
   (GLpolygonStipple fichier_bitmap)
*/
Int
gl_load_polygon_stipple ()  gl_load_polygon_stipple
{
#ifdef HAVE_XRBFD
  unsigned char *f_name, *data = NULL;
  unsigned int w, h, xh, yh;

  if(!isstr(a1)) { a1 = nil; derec; }
  f_name = STR_FROM_STRLSP(a1);
  if(!XReadBitmapFileData(f_name, &w, &h, &data, &xh, &yh)) {
    if(w != 32 || h != 32) {
      a1 = nil;
      derec;
    }
    glPolygonStipple(data);
    XtFree(data);
    a1 = t;
  }
```

```
  else
#endif
    a1 = nil;

  derec;
}

/* ************************************** */
/* La librairie libtiff n'est pas présente */
/* ************************************** */

#ifndef HAVE_LIBTIFF

int
gl_dump_window  ()gl_dump_window
{
  wst("La librairie libtiff est nécessaire à l'exécution de cette primitive\n");
  a1 = nil;
  derec;
}
int
gl_read_tiff  ()          gl_read_tiff
{
  wst("La librairie libtiff est nécessaire à l'exécution de cette primitive\n");
  a1 = nil;
  derec;
}

#else

/* ********************************** */
/* La librairie libtiff est présente */
/* ********************************** */

/* ***************************************** */
/* Première partie : dump sous forme de tiff */
/* ***************************************** */

/* OpenGL image dump, written by Reto Koradi (kor@spectrospin.ch) */

/* This file contains code for doing OpenGL off-screen rendering and
   saving the result in a TIFF file. It requires Sam Leffler's libtiff
   library which is available from ftp.sgi.com.
   The code is used by calling the function StartDump(..), drawing the
   scene, and then calling EndDump(..).
   Please note that StartDump creates a new context, so all attributes
   stored in the current context (colors, lighting parameters, etc.)
   have to be set again beforing performing the actual redraw. This
   can be rather painful, but unfortunately GLX does not allow
   sharing/copying of attributes between direct and nondirect
   rendering contexts. */

#include <tiffio.h>

#define RaiseError(str) fprintf(stderr, "GLdump: %s\n", str)

extern GLObject *wobj;

/* X servers often grow bigger and bigger when allocating/freeing
   many pixmaps, so it's better to keep and reuse them if possible.
   Set this to 0 if you don't want to use that. */
#define KEEP_PIXMAP 0

static FILE *TiffFileP;
static Int Orient;
static Int ImgW, ImgH;
#ifdef USE_PIXMAP
static Bool OutOfMemory;
static Display *Dpy;
static Pixmap XPix = 0;
static GLXPixmap GPix = 0;
static GLXContext OldCtx, Ctx;
#endif

#ifdef USE_PIXMAP
static void
```

```
#ifndef __SDTC__
destroyPixmap ()    destroyPixmap
#else
destroyPixmap (void)          destroyPixmap
#endif
{
  glXDestroyGLXPixmap(Dpy, GPix);
  GPix = 0;
  XFreePixmap(Dpy, XPix);
  XPix = 0;
}

static int
#if STDC_HEADERS
xErrorHandler (Display *dpy, XErrorEvent *evtP)   xErrorHandler
#else
xErrorHandler (dpy, evtP)      xErrorHandler
Display *dpy; XErrorEvent *evtP;
#endif
{
  OutOfMemory = True;
  return 0;
}

#endif

static int
#if STDC_HEADERS
writeTiff (void)        writeTiff
#else
writeTiff () writeTiff
#endif
{
  TIFF *tif;
  Int tiffW, tiffH;
  Int bufSize, rowI;
  unsigned char *buf;
  Int res;
#ifndef USE_PIXMAP
  int gl_pack_align ;
#endif

  tif = TIFFFdOpen(fileno(TiffFileP), "output file", "w");
  if (tif == NULL) {
    RaiseError("could not create TIFF file");
    return 1;
  }

#ifndef USE_PIXMAP
  glGetIntegerv(GL_PACK_ALIGNMENT, &gl_pack_align);
#endif
  if (Orient == 0) {
    tiffW = ImgW;
    tiffH = ImgH;
    bufSize = 4 * ((3 * tiffW + 3) / 4);
    glPixelStorei(GL_PACK_ALIGNMENT, 4);
  } else {
    tiffW = ImgH;
    tiffH = ImgW;
    bufSize = 3 * tiffW;
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
  }

  TIFFSetField(tif, TIFFTAG_IMAGEWIDTH, tiffW);
  TIFFSetField(tif, TIFFTAG_IMAGELENGTH, tiffH);
  TIFFSetField(tif, TIFFTAG_BITSPERSAMPLE, 8);
  TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_LZW);
  TIFFSetField(tif, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB);
  TIFFSetField(tif, TIFFTAG_FILLORDER, FILLORDER_MSB2LSB);
  TIFFSetField(tif, TIFFTAG_DOCUMENTNAME, "Xbvl-OpenGL");
  TIFFSetField(tif, TIFFTAG_IMAGEDESCRIPTION, "Image générée depuis Xbvl-Mesa/Open GL");
  TIFFSetField(tif, TIFFTAG_SAMPLESPERPIXEL, 3);
  TIFFSetField(tif, TIFFTAG_ROWSPERSTRIP, (8 * 1024) / (3 * tiffW));
  TIFFSetField(tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);

  buf = malloc(bufSize * sizeof(*buf));
```

```
  res = 0;
  for (rowI = 0; rowI < tiffH; rowI++) {
    if (Orient == 0)
      glReadPixels(0, ImgH - 1 - rowI, ImgW, 1,
        GL_RGB, GL_UNSIGNED_BYTE, buf);
    else
      glReadPixels(rowI, 0, 1, ImgH,
        GL_RGB, GL_UNSIGNED_BYTE, buf);

    if (TIFFWriteScanline(tif, buf, rowI, 0) < 0) {
      RaiseError("error while writing TIFF file");
      res = 1;
      break;
    }
  }

  free(buf);

  glPixelStorei(GL_PACK_ALIGNMENT, gl_pack_align);
  TIFFFlushData(tif);
  TIFFClose(tif);

  return res;
}

static int
#if STDC_HEADERS
EndDump  (Widget drawW)      EndDump
#else
EndDump  (drawW)  EndDump
Widget drawW;
#endif
/* Write current image to file. May only be called after StartDump(..).
   Returns 0 on success, calls RaiseError(..) and returns 1 on error. */
{
  Int res;

  res = writeTiff();
  (void) fclose(TiffFileP);

#ifdef USE_PIXMAP
  (void) glXMakeCurrent(Dpy, XtWindow(drawW), OldCtx);

#if KEEP_PIXMAP
#else
  destroyPixmap();
#endif

  glXDestroyContext(Dpy, Ctx);
#endif
  return res;
}

static int
#if STDC_HEADERS
Start_Dump  (char *fileName, Int orient, Int w, Int h, Widget drawW)      Start_Dump
#else
Start_Dump  (filename, orient, w, h, drawW)          Start_Dump
char *fileName;
Int orient, w, h;
Widget drawW;
#endif
/* Prepare for image dump. fileName is the name of the file the image
   will be written to. If orient is 0, the image is written in the
   normal orientation, if it is 1, it will be rotated by 90 degrees.
   w and h give the width and height (in pixels) of the desired image.
   Returns 0 on success, calls RaiseError(..) and returns 1 on error. */
{
  /*  Widget drawW = GetDrawW(); */  /* the GLwMDrawA widget used */
#ifdef USE_PIXMAP
  XErrorHandler oldHandler;
  Int attrList[10];
  XVisualInfo *visP;
  Int n, i;
#endif
```

```
  TiffFileP = fopen(fileName, "w");
  if (TiffFileP == NULL) {
    RaiseError("could not open output file");
    return 1;
  }

  Orient = orient;
  ImgW = w;
  ImgH = h;

#ifdef USE_PIXMAP

#if KEEP_PIXMAP
  if (GPix != 0 && (w != ImgW || h != ImgH))
    destroyPixmap();
#endif

  Dpy = XtDisplay(drawW);

  n = 0;
  attrList[n++] = GLX_RGBA;
  attrList[n++] = GLX_RED_SIZE; attrList[n++] = 8;
  attrList[n++] = GLX_GREEN_SIZE; attrList[n++] = 8;
  attrList[n++] = GLX_BLUE_SIZE; attrList[n++] = 8;
  attrList[n++] = GLX_DEPTH_SIZE; attrList[n++] = 1;
  attrList[n++] = None;
  visP = glXChooseVisual(Dpy,
      XScreenNumberOfScreen(XtScreen(drawW)), attrList);
  if (visP == NULL) {
    RaiseError("no 24-bit true color visual available");
    return 1;
  }

  /* catch BadAlloc error */
  OutOfMemory = False;
  oldHandler = XSetErrorHandler(xErrorHandler);

  if (XPix == 0) {
    XPix = XCreatePixmap(Dpy, XtWindow(drawW), w, h, 24);
    XSync(Dpy, False);  /* error comes too late otherwise */
    if (OutOfMemory) {
      XPix = 0;
      XSetErrorHandler(oldHandler);
      RaiseError("could not allocate Pixmap");
      return 1;
    }
  }

  if (GPix == 0) {
    GPix = glXCreateGLXPixmap(Dpy, visP, XPix);
    XSync(Dpy, False);
    XSetErrorHandler(oldHandler);
    if (OutOfMemory) {
      GPix = 0;
      XFreePixmap(Dpy, XPix);
      XPix = 0;
      RaiseError("could not allocate Pixmap");
      return 1;
    }
  }

  Ctx = glXCreateContext(Dpy, visP, NULL, False);
  if (Ctx == NULL) {
    destroyPixmap();
    RaiseError("could not create rendering context");
    return 1;
  }

  OldCtx = glXGetCurrentContext();
  (void) glXMakeCurrent(Dpy, GPix, Ctx);

#endif
  return 0;
}
```

```
#ifdef STDC_HEADERS
extern void build_projection  (char, int, int);          build_projection
#else
extern void build_projection  ();              build_projection
#endif

Int
gl_dump_window  ()gl_dump_window
{
  char *fname;
  Int w, h, i;
  if(isatom(a1)) fname=STR_FROM_STRLSP(ADRPNAME(a1));
  else if(isstr(a1)) fname=STR_FROM_STRLSP(a1);
  else {a1 = nil; derec;}

  if(!isnum(a2) || !isnum(a3)) {a1 = nil; derec;}

  if(!Start_Dump(fname, 0, fvalue(a2), fvalue(a3), (Widget) wobj->w)) {
    /* redrawing the scene */

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixd(wobj->Proj_Mat);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd(wobj->Model_Mat);
    glCallList(wobj->root_list);
    glFlush();
    if(!EndDump((Widget)wobj->w)) a1 = t;
    else a1 = nil;
  }
  else a1 = nil;
  derec;
}

/* ******************************************** */
/* deuxième partie : lecture d'un fichier tiff */
/* ******************************************** */

/* Maintenant la définition pour lire un tiff vers un buffer,
   le buffer retourné est à libérer en utilisant XtFree
 */
static TK_RGBImageRec r_image;

TK_RGBImageRec *
#ifdef STDC_HEADERS
LoadTiffImage  (char *f_name)LoadTiffImage
#else
LoadTiffImage  (f_name)        LoadTiffImage
char *f_name;
#endif
{
  TIFF *tif = TIFFOpen(f_name, "r");
  r_image.data = NULL;
  if(tif) {
    TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &r_image.sizeX);
    TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &r_image.sizeY);
    if((r_image.data = alloc_image_data(r_image.sizeX, r_image.sizeY)) != NULL) {
      if (!TIFFReadRGBAImage(tif, (uint32)r_image.sizeX, (uint32)r_image.sizeY, (uint32 *)&r_image.data, 0))
{
   XtFree((char *)r_image.data);
   r_image.data = NULL;
     }
    }
    TIFFClose(tif);
  }
  return (TK_RGBImageRec *)&r_image;
}

/* copy une image depuis un ficher tiff vers une position donnée
   (GLloadTiff X Y Z FileName [ZoomX [ZoomY]])
*/
Int
gl_read_tiff  ()          gl_read_tiff
{
  short pos[3];
  char *f_name;
```

```
  TK_RGBImageRec *image;
  float zx = 0.0, zy = 0.0;

  if(!getNumArgFroma1((char *) pos, 3, GLXBVL_SHORT) || !isstr(car(a1))) { a1 = nil; derec; }
  f_name = STR_FROM_STRLSP(car(a1));
  image = LoadTiffImage(f_name);
  if(!image->data) { a1 = nil; derec; }
  a1 = cdr(a1);
  if(isnum(car(a1))) {
    zx = fvalue(car(a1));
    a1 = cdr(a1);
    if(isnum(car(a1))) zy = fvalue(car(a1));
    else zy = zx;
  }
  else zx = zy = 1;

  push(zx); push(zy); push(pos[2]); push(pos[1]); push(pos[0]); push(image);  push(1);
  pc = display_image_file;
}

#endif

/* gestion des images RGB en utilisant la librairie GL/Mesatk */

/* lecture et affichage d'une image RGB */
/* (GLloadRGB X Y Z FileName [ZoomX [ZoomY]]) */
Int gl_read_rgb  ()     gl_read_rgb
{
  short pos[3];
  char *f_name;
  TK_RGBImageRec *image;
  float zx = 0.0, zy = 0.0;

  if(!getNumArgFroma1((char *) pos, 3, GLXBVL_SHORT) || !isstr(car(a1))) { a1 = nil; derec; }
  f_name = STR_FROM_STRLSP(car(a1));
  if(!(image = tkRGBImageLoad(f_name))) { a1 = nil; derec; }
  a1 = cdr(a1);
  if(isnum(car(a1))) {
    zx = fvalue(car(a1));
    a1 = cdr(a1);
    if(isnum(car(a1))) zy = fvalue(car(a1));
    else zy = zx;
  }
  else zx = zy = 1;

  push(zx); push(zy); push(pos[2]); push(pos[1]); push(pos[0]); push(image); push(0);
  pc = display_image_file;
}


#endif
```

Annexe III.11 Fonctions d'interface : glLib.c

```
/*
 *      glLib.c
 * Librairie d'interface entre gl (IRIS)
 * Ce fichier contient plus specifiquement les fonction autres que
 * les fonctions de dessin de la librairie GL.
 *     D.Ploix, novembre 93


 *
 */

#if (OPEN_GL | MESA_GL)


#include "vlisp.h"

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include "glXbvl.h"
#include "glTrans.h"

/* ****************************************************************** */
/*
 * getGLvalue : recherche dans la table de translation String -> Value
 * la chaine donnee en argument
 */
long
getGLvalue  (str)      getGLvalue
String str;
{
  register Int a = 0, b = GL_TTABLE_SIZE - 1, i = GL_TTABLE_SIZE / 2, val, k;
  if(str[0] < 'A' || str[0] > 'Z') return -1;
  for(k = 0 ; ; k--) {
    if(!(val = strcmp(str,GLTTable[i].nom))) break;
    else if(val < 0) {
      b = i;
      i -= (i - a) / 2;
      if(b == i) break;
    }
    else {
      a = i;
      i += (b - i) / 2;
      if(a == i) break ;
    }
  }
  if(!val) return GLTTable[i].value;
  return -1;
}

/* recuperation d'arguments numeriques de a1 */
/* type = GLXBVL_SHORT : short */
/* type = GLXBVL_FLOAT : float */
/* type = GLXBVL_DOUBLE : double */
Int
getNumArgFroma1  (res, num, type)      getNumArgFroma1
char *res;
short num;
char type;
{
  union { char *c; short *s; float *f; double *d; } val;
  Int *tmp, cnt=0;
  val.c = res;
  for( ; islist(a1) && (!num || cnt < num) ; a1=cdr(a1)) {
    tmp = car(a1);
    if(!isnum(tmp) && !isflo(tmp)) break;
    switch(type) {
    case GLXBVL_CHAR:   val.c[cnt++] = (char) fvalue(tmp); break;
    case GLXBVL_DOUBLE: val.d[cnt++] = (double) fvalue(tmp); break;
    case GLXBVL_FLOAT:  val.f[cnt++] = (float) fvalue(tmp); break;
    case GLXBVL_SHORT:  val.s[cnt++] = (short) fvalue(tmp); break;
```

```
    }
  }
  if(!num) return cnt;
  if(cnt >= num) return 1;
  return 0;
}

#endif
```

Annexe III.12 Librairie de construction de menus : X-menus.vlisp

```
;
; Constructeur de methodes graphiques : partie fonctionnelle
;
;
; Fonctions definies ici :
;
;
; Do-Selection-Menu : Lancement d'un menu selectionnable avec creation/destruction de fenetres
; Selection-Menu    : menu de selection pour 1 ou plusieurs choix
;
; Methode-Menu      : menu generaux compose de champs de titre, de champs selectable avec ou sans valeur apparente
; Menu-Present      : retourne t si le menu est present...
; Remove-All-Methode-Menu : effacement de tous les Methode-Menu lie a un objet
; Remove-Menu-If-Present : effacement du Methode-Menu lie a un objet si il est present
; Remove-Methode-Menu    : effacement du Methode-Menu lie a un objet
;
; Yes-No-Menu : Menu oui-non avec actions
;
; Font par defaut
(setq defun 'de deff 'df
;      default-font "-b&h-lucida-medium-r-normal-sans-12-0-75-75-p-*-iso8859-1"
       bitmap-directory "/usr/people/damien/work/bitmap/"     ; repertoire des bitmaps
       help-event nil                                         ; evennement lie au Help
       help-file nil)                                         ; fichier d'aide a lancer

(defun Method-Get-Value (obj elem)       Method-Get-Value
  (cond
   ((atom obj) (get obj elem))
   ((listp obj) (cdr (assoc elem obj)))))
;
; Si help-event ET help-file ont une valeur (par ex. "<Btn2Dwn>" et "(my-help)"), lors de
; l'avenement de l'evenement help-event dans un widget, l'action my-help sera mise en route.
;
(defun X-Aide (topic file)          X-Aide
  (if topic (aide topic nil file)))
;
; Do-Selection-Menu : menu de selection pour 1 ou plusieurs choix
;
; Parametres :
;  obj    = variable a modifier
;  field  = champ ou placer les information
;  w-info = widget a reactualiser lors de la sortie du menu
;  title  = titre de la fenetre de choix
;  data   = choix
;  action = action post-selection
;
(defun Do-Selection-Menu (obj field w-info title data r-instr . w-values)  Do-Selection-Menu
    (let (root (xCreateWidget '--OS-- "OverrideShell" (xwinp)
                                  "x" (- (car (xGetPosPointer)) 10) "y" (- (cadr (xGetPosPointer)) 10)))
      (let (bx (xCreateWidget '--BX-- "awBox" root))
   (let ((l data) (new-w))
     (if (null l) t
       (setq new-w (eval `(xCreateWidget '--SM-- "awCommand" bx "font" default-font "label" (cdar l) ,@w-values)))
       (rplacd new-w `(Selection-Componant ',obj ',field ',root  ',w-info
                                                          ',r-instr ',(caar l)))
       (xAddCallback new-w "callback" (strcat "(eval (cdr '" new-w "))"))
       (if (and help-event help-file)
               (xAugment new-w help-event (strcat "(X-Aide '" (caar l) " help-file)")))
       (self (cdr l)))))
      (xPopup root)))
;
; Callback des selections
;
(defun Selection-Componant (obj field w w-info feed-back val)          Selection-Componant
    (xPopdown w)
    (if (and obj field) (put obj field val))
    (if w-info (xSetValues w-info "label" val))
    (if feed-back (eval `(,feed-back ',obj ',field ',w-info))))
;
;
; Construction des Menus des methodes
```

```
;
; Parametres :
;  obj     = objet
;  w       = widget pere
;  x0, y0  = position initiale
;  menu    = contenant du menu
;  package = package d'origine du menu
;
(deff Methode-Menu (params)  Methode-Menu
 (let ((obj (eval (1 params)))
  (w (eval (2 params)))
  (x (eval (3 params)))
  (nom-menu (5 params))
  (data (eval (5 params)))
  (pkg  (6 params))
  (w-values (nth 7 params))
  (w-create))
  (setq w-create `(xCreateWidget '--MI-- "awCommand" w "font" default-font  "x" x "y" y ,@w-values))
  (ifn w (setq w (Methode-Menu-Window nom-menu (length data))))
  (setq x0 x)
  (if (member nom-menu (get obj 'Methode-Menu)) nil
    (let ((l data) (y (eval (4 params))) (wl) (isbmp))
   (if isbmp
      (cond
       ((listp (caar l)) (setq x (+ x (+ 4 (xGetValues (car wl) "width")))))
       (t                      (setq x x0 y (+ y (+ (xGetValues (car wl) "height") 7))))))
   (cond
    ((null l)
     (put obj 'Methode-Menu (cons nom-menu (get obj 'Methode-Menu)))
     (put obj nom-menu wl))
    ((stringp (car l))
     (setq wl `(,(xCreateWidget '--ML-- "awLabel" w "font" default-font
                                        "label" (car l) "x" x "y" y
                                        "background" "#33f" "foreground" "#fff") ,@wl))
     (self (cdr l) (+ y 25) wl))
    ((null (cadar l))
     (setq wl `(,(eval w-create) ,@wl))
     (when (cddar l)
               (if pkg
                  (xAddCallback (car wl) "callback" (strcat "(" pkg "|" (cddar l) " '" pkg "|" obj " '|" w " '|" (car wl)
")"))
                 (rplacd (car wl) `(,(cddar l) ',obj ',w ',(car wl)))
                 (xAddCallback (car wl) "callback" (strcat "(eval (cdr '" (car wl) "))")))
               (if (and help-event help-file)
                  (xAugment (car wl) help-event (strcat "(X-Aide '" (cddar l) " help-file)"))))
     (cond
      ((stringp (caar l))
       (xSetValues (car wl) "label" (caar l))
       (self (cdr l) (+ y 25) wl))
      (t
       (xSetValues (car wl) "bitmap" (strcat bitmap-directory (car (caar l))))
       (self (cdr l) y wl t))))
    (t
     (let ((tmp-c (eval w-create))
               (tmp-l))
       (if (stringp (caar l)) (xSetValues tmp-c "label" (caar l))
         (xSetValues tmp-c "bitmap" (strcat bitmap-directory (car (caar l)))))
       (if (atom (cadar l)) (setq tmp-l (xCreateWidget '--ML-- "awLabel" w "label"
                                                                    (strcat (Method-Get-Value obj (cadar l)))))
         (setq tmp-l (xCreateText '--MT-- w (strcat (Method-Get-Value obj (car (cadar l)))) t)))
       (xSetValues tmp-l  "font" default-font
                          "x" (+ x (+ 10 (xGetValues tmp-c "width"))) "y" y)
       (setq wl `(,tmp-c ,tmp-l ,@wl))
       (if (and help-event help-file)
               (xAugment tmp-c help-event (strcat "(X-Aide '" (cadar l) " help-file)")))
       (if (cddar l)
               (if pkg
                  (xAddCallback tmp-c "callback" (strcat "("  pkg "|" (cddar l) " '" pkg "|" obj
                                                                    " '" pkg "|" (cadar l) " '|" (cadr wl) "'|" w
                                                                    " " (+ x 250) " " y ")"))
                 (rplacd tmp-c `(,(cddar l) ',obj ',(cadar l) ',(cadr wl) ',w ,(+ x 250) ,y))
                 (xAddCallback tmp-c "callback" (strcat "(eval (cdr '" tmp-c "))")))))
     (self (cdr l) (+ y 25) wl (listp (caar l)))))))))

; Creation du widget du menu si besoin
(defun Methode-Menu-Window (title n-y-objs)       Methode-Menu-Window
```

```
  (let (root (xCreateWidget '--AS-- "ApplicationShell" "title" title "iconName" title))
    (let (drw (xCreateWidget '--AD-- "Drawxbvl" root "background" "#4a2"
                             "width" (+ 8 (* 25 n-y-objs)) "height" (+ 8 (* 25 n-y-objs))))
      (xRealize root)
      drw)))

; Y a t'il un menu "menu" pour l'objet "obj"
(defun Menu-Present (obj menu) (member menu (get obj 'Methode-Menu)))        Menu-Present

; effacement des menus lies a un objet
(defun Remove-All-Methode-Menu (obj) Remove-All-Methode-Menu
   (mapc (get obj 'Methode-Menu)
    (lambda (x) (Remove-Methode-Menu obj x))))

; effacement d'un menu si present
(defun Remove-Menu-If-Present (obj nom-menu)  Remove-Menu-If-Present
   (if (member nom-menu (get obj 'Methode-Menu)) (Remove-Methode-Menu obj nom-menu)))

; effacement d'un menu lie a un objet
(defun Remove-Methode-Menu (obj nom-menu)    Remove-Methode-Menu
   (mapc (get obj nom-menu) 'xRemoveWidget)
   (remprop obj nom-menu)
   (let (x (delete-one (get obj 'Methode-Menu) nom-menu))
    (ifn x (remprop obj 'Methode-Menu)
    (put obj 'Methode-Menu x))))

; reaffectation des valeurs affichees dans un menu
(deff Fill-Methode-Menu-Values (values)Fill-Methode-Menu-Values
   (ifn (cadr values) nil
   (let ((obj (eval (car values)))
        (nom-menu (cadr values))
        (menu (eval (cadr values))))
    (ifn (member nom-menu (get obj 'Methode-Menu)) nil
            (let ((vals (reverse menu)) (ws (get obj nom-menu)))
              (cond
               ((null vals) t)
               ((or (stringp (car vals)) (null (cadar vals))) (self (cdr vals) (cdr ws)))
               (t
                (xSetValues (cadr ws) "label" (strcat (Method-Get-Value obj (cadar vals))))
                (self (cdr vals) (cddr ws)))))))))

;
; Menu oui non : arguments = label, action si oui, action si non.
;
(defun Yes-No-Menu (label act-yes act-no)          Yes-No-Menu
   (let (root (xCreateWidget '--AS-- "OverrideShell" (xwinp)
                             "x" (- (car (xGetPosPointer)) 10) "y" (- (cadr (xGetPosPointer)) 10)))
    (let (bx (xCreateWidget '--AB-- "awBox" root))
  (xCreateWidget '--AL-- "awLabel" bx "label" label)
  (put root 'yes act-yes)
  (put root 'no  act-no)
  (xAddCallback (xCreateWidget '--AC-- "awCommand" bx "label" "Oui" "foreground" "#fff" "background"
"#666")
                "callback" (strcat "(Yes-No-Return '" root " 'yes)"))
  (xAddCallback (xCreateWidget '--AC-- "awCommand" bx "label" "Non" "foreground" "#fff" "background"
"#666")
                "callback" (strcat "(Yes-No-Return '" root " 'no)" )))
    (xPopup root)))

(defun Yes-No-Return (w act)  Yes-No-Return
   (xPopdown w)
   (if (get w act) (eval (get w act)))
   (rplacd w nil))
```

Annexe III.13 Librairie de navigation graphique : GL-drive.vlisp

```
;  $Id: GL-drive.vlisp,v 1.5 1995/12/20 13:21:57 damien Exp damien $
;; ;
; Pilotage d'une fenetre GL par la souris (Translation, Rotation, Scaling)
; D.P. 12/93
;; ;
;
; Resource globale : Active-GL-Window : contien la fenetre actuellement conduite par Sdrive (et active pour
GL)
;
;
(package GLD)
(setq defun 'de defvar 'setq)

(defvar Sdrive-Window (gensym))


                                                ; ********************** ; ;
                                                ; Definition des widgets ; ;
                                                ; ********************** ; ;
;
;
; Control de la fenetre ...
; Parametre :
;
;   glWindow     = atome descripteur de la fenetre GL
;
(defun |GLdrive (glWindow)     |GLdrive
  (cond
   ((null glWindow) nil)
   ((get glWindow 'speed-value) (SetSdrive glWindow))                    ; deja geree par GLdrive : change-
ment pour glWindow
   (t
    (put glWindow 'trans '(0 0 0))
    (put glWindow 'rot '(0 0 0))
    (put glWindow 'scale FixSize)
    (put glWindow 'proj 'ortho)
    (if (get Sdrive-Window 'up) t
      (let (root (xCreateWidget '--AS-- "ApplicationShell" "title" "GL Drive" "iconName" "GL Drive"))
    (put Sdrive-Window 'speed-value 1)
    (let (form (xCreateWidget '--BW-- "awBox" root "width" 230 "height" 263 "background" "#494"))
      (NewW 'object "awLabel" "                    " "background" "#F5F")
      (xCreateWidget '--AL-- "awLabel" form "label" "   Btn1 | Btn2 | Btn3   \n   Auto | Set  | Step   ")
      (NewCW 'rotate-x+ "rotate-x+")    (NewCW 'rotate-x- "rotate-x-")
      (NewCW 'rotate-y+ "rotate-y+")    (NewCW 'rotate-y- "rotate-y-")
      (NewCW 'rotate-z+ "rotate-z+")    (NewCW 'rotate-z- "rotate-z-")
      (NewCW 'right     "right")            (NewCW 'left      "left")
      (NewCW 'up        "up")               (NewCW 'down      "down")
      (NewCW 'front     "front")            (NewCW 'deep      "deep")
      (NewCW 'zoom-in   "zoom-in")      (NewCW 'zoom-out  "zoom-out")
      (xAddCallback (xCreateWidget '--CW-- "awCommand" form "bitmap" (mkbmp "zoom-") "background"
"#888" "foreground" "#0
0")
                    "callback" (strcat "(GLD|UnZoom (get 'GLD|" Sdrive-Window " 'GLD|driven))"))
      (NewW 'drive-stop "awCommand" "Stop" "background" "font" "a14" "#888" "foreground" "#000")
      (NewW 'speed "awLabel" "1")
      (NewCBK "Vitesse +" "IncrementSpeed" "1") (NewCBK "Vitesse -" "IncrementSpeed" "-1")
      (NewCBK "Réinit" "reinit-sdrive" " ")
      (xAddCallback (xCreateWidget '--CW-- "awCommand" form "label" "Ortho" "font" "a14" "background"
"#888" "foreground"
"#000")
                    "callback" (strcat "(GLD|sdrive-projection "' glWindow " $W)"))
        (NewCBK "Status GL" "GL-Status-Window" " ")
        (NewCBK "Détruit GL" "Kill-GL-Window" " ")
        (NewCBK "Image TIFF" "Dump-GL-Window" " ")
        (xAddCallback (xCreateWidget '--CW-- "awCommand" form "label" "Aide"  "background" "#888" "font"
"a14" "fore
round" "#000")
                  "callback" "(GLD|help)")
      (MkClbk 'rotate-x+ 'rot (strcat "\"x\" (get 'GLD|" Sdrive-Window " 'GLD|speed-value)"))
      (MkClbk 'rotate-x- 'rot (strcat " \"x\" (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value))"))
      (MkClbk 'rotate-y+ 'rot (strcat " \"y\" (get 'GLD|" Sdrive-Window " 'GLD|speed-value)"))
```

```
        (MkClbk 'rotate-y- 'rot (strcat " \"y\" (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value))"))
        (MkClbk 'rotate-z+ 'rot (strcat " \"z\" (get 'GLD|" Sdrive-Window " 'GLD|speed-value)"))
        (MkClbk 'rotate-z- 'rot (strcat " \"z\" (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value))"))
        (MkClbk 'right    'trans (strcat " (get 'GLD|" Sdrive-Window " 'GLD|speed-value) 0 0"))
        (MkClbk 'left     'trans (strcat " (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value)) 0 0"))
        (MkClbk 'up       'trans (strcat " 0 (get 'GLD|" Sdrive-Window " 'GLD|speed-value) 0"))
        (MkClbk 'down     'trans (strcat " 0 (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value)) 0"))
        (MkClbk 'front    'trans (strcat " 0 0 (get 'GLD|" Sdrive-Window " 'GLD|speed-value)"))
        (MkClbk 'deep     'trans (strcat " 0 0 (- (get 'GLD|" Sdrive-Window " 'GLD|speed-value))"))
        (MkClbk 'zoom-in   'scale " ") (MkClbk 'zoom-out  'scale " t")

        (MkBtn2 'rotate-x+ 'rot)      (MkBtn2 'rotate-x- 'rot)     (MkBtn2 'rotate-y+ 'rot)
        (MkBtn2 'rotate-y- 'rot)      (MkBtn2 'rotate-z+ 'rot)     (MkBtn2 'rotate-z- 'rot)
        (MkBtn2 'right 'trans)        (MkBtn2 'left 'trans)        (MkBtn2 'up 'trans)
        (MkBtn2 'down 'trans)         (MkBtn2 'front 'trans)       (MkBtn2 'deep 'trans)
        (MkBtn2 'zoom-in 'scale)      (MkBtn2 'zoom-out 'scale)

        (MkBtn3 'rotate-x+ 'rot 1 1)   (MkBtn3 'rotate-x- 'rot 1 -1) (MkBtn3 'rotate-y+ 'rot 2 1)
        (MkBtn3 'rotate-y- 'rot 2 -1)  (MkBtn3 'rotate-z+ 'rot 3  1) (MkBtn3 'rotate-z- 'rot 3 -1)
        (MkBtn3 'right 'trans 1 1)     (MkBtn3 'left 'trans 1 -1)    (MkBtn3 'up 'trans 2 1)
        (MkBtn3 'down 'trans  2 -1)    (MkBtn3 'front 'trans 3 -1)   (MkBtn3 'deep 'trans 3 1)
        (MkBtn3 'zoom-in 'scale 1 -1)  (MkBtn3 'zoom-out 'scale 1 1)
        (xRealize root)
        (put Sdrive-Window 'dialog (Screate-dialog)))))
       (xSetValues (get Sdrive-Window 'object) "label" (|xGetParentTitle glWindow))
       (xOverride (xGetSupWidget glWindow "ApplicationShell") "<Enter>" (strcat "(GLD|SetSdrive '|" glWindow
")"))
        (GLcallback "MotionNotify"   "(GLD|DriveMouseMotion '$W $B $X $Y)")
        (GLcallback "ButtonPress"    "(GLD|DriveButtonPress '$W $G)")
        (GLcallback "ButtonRelease"  "(GLD|DriveButtonRelease '$W $G $B $X $Y)")
        (put Sdrive-Window 'driven glWindow)
        (setq |Active-GL-Window glWindow))))

(defun help ()          help
  (aide nil nil (strcat |VLISPDIR "GL-Drive-Help")))

(defun SetSdrive (win)          SetSdrive
  (GLwinset win)
  (put Sdrive-Window 'driven win)
  (setq |Active-GL-Window win)
  (xSetValues (get Sdrive-Window 'object) "label" (|xGetParentTitle win)))


(defun Screate-dialog ()          Screate-dialog
    (let (root (xCreateWidget '--AS-- "ApplicationShell"))
      (let (bx (xCreateWidget '--BX-- "awBox" root))
     (put root 1 (xCreateWidget '--AD-- "awDialog" bx))
     (put root 2 (xCreateWidget '--AD-- "awDialog" bx))
     (put root 3 (xCreateWidget '--AD-- "awDialog" bx))
     (put root 'conf (xCreateWidget '--AC-- "awCommand" bx "label" "OK" "background" "#888" "foreground"
"#000"))
      (xAddCallback (xCreateWidget '--AC-- "awCommand" bx "label" "Cancel" "background" "#888" "fore-
ground" "#000")
                    "callback" (strcat "(GLD|Cancel-Dialog '" root ")")))

      root))

; Dump de la fenetre dans un fichier tiff
(defun Dump-GL-Window ()     Dump-GL-Window
  (ifn (get Sdrive-Window 'driven) nil
       (GLdump (strcat "I" (get Sdrive-Window 'driven) ".tif")
         (xGetValues (get Sdrive-Window 'driven) "width")
         (xGetValues (get Sdrive-Window 'driven) "height"))
       (print "Window dumped into" (strcat "I" (get Sdrive-Window 'driven) ".tif"))))

; Destruction de la fenetre GL
(defun Kill-GL-Window ()          Kill-GL-Window
    (ifn (get Sdrive-Window 'driven) nil
    (xRemove (get Sdrive-Window 'driven))
    (put Sdrive-Window 'driven nil)
    (xSetValues (get Sdrive-Window 'object) "label" "        ")))


                                        ; ********************* ; ;
                                        ; Conduite de la fenetre ; ;
                                        ; ********************* ; ;
```

```
; transformations
;
; Switch des transformations
;
(defun Transform (op win v0 v1 v2)        Transform
  (SetSdrive win)
  (let ((r-data) (r-obj))
    (setq r-data (get win op))
    (let (new-Trans  (cond
                       ((eq op 'rot) (Rotate v0 v1))
                       ((eq op 'trans) (Translate v0 v1 v2))
                       ((eq op 'scale) (Scale v0))))
      (if new-Trans (eval new-Trans)))))

                                                ; ********************************** ; ;
                                                ; Lancement de la boucle d'execution ; ;
                                                ; ********************************** ; ;

(defun Translate (dx dy dz)       Translate
   (let ((s-fact (get win 'scale)) (x) (y) (z) (ax) (ay) (az))

     (setq ax (car (get win 'p-add)))
     (setq ay (cadr (get win 'p-add)))
     (setq az (caddr (get win 'p-add)))
     (ifn r-data (setq r-data '(0 0 0)))
     (setq x (car r-data))     (setq y (cadr r-data))   (setq z (caddr r-data))

     (TransformLoop '(progn (setq r-data `(,x ,y ,z)) (put win 'trans r-data))
                    '(progn (incr x dx) (incr y dy) (incr z dz)
                            (GLimtranslate (+ x ax) (+ y ay) (+ z az))))))

(defun Rotate (o qua)Rotate
   (let (dir)

     (setq dir (if (equal o "x") 1 (if (equal o "y") 2 3)))
     (ifn r-data (setq r-data '(0 0 0)))

     (let (rot (car (nth dir r-data)))
   (TransformLoop '(progn (set (nth dir r-data) rot) (put win 'rot r-data))
                  '(progn (incr rot qua) (GLimrotate rot o))))))

(defun Scale (zooming)            Scale
   (ifn r-data (setq r-data FixSize))
   (let ((z-incr (* (get Sdrive-Window 'speed-value) (if zooming 1 -1)))
    (s-value r-data))

     (TransformLoop '(progn (setq r-data s-value) (put win 'scale r-data))
                    '(progn (setq s-value (+ s-value z-incr)) (GLimscale s-value s-value s-value FixSize)))))

                                                ; ****************** ; ;
                                                ; Boucle d'execution ; ;
                                                ; ****************** ; ;

(defun TransformLoop (op-end op-loop run)          TransformLoop
  (setq rotate-x+ (get Sdrive-Window 'rotate-x+) rotate-x- (get Sdrive-Window 'rotate-x-) rotate-y+ (get
Sdrive-Window
'rotate-y+)
  rotate-y- (get Sdrive-Window 'rotate-y-) rotate-z+ (get Sdrive-Window 'rotate-z+) rotate-z- (get Sdrive-
Window 'rotat
-z-)
  right    (get Sdrive-Window 'right   ) left     (get Sdrive-Window 'left    ) up       (get Sdrive-Window 'up
  )
  down     (get Sdrive-Window 'down    ) front    (get Sdrive-Window 'front   ) deep     (get Sdrive-Window
'deep
  )
  zoom-in   (get Sdrive-Window 'zoom-in ) zoom-out  (get Sdrive-Window 'zoom-out)  stop     (get Sdrive-
Window 'drive
stop)
  speed    (get Sdrive-Window 'speed-value)
  trans nil
  run t)
 (while run
  (setq run nil)
  (cond
   ((xCheckEvent rotate-x+ "ButtonPress") (setq trans `(Transform 'rot ,win "x" ,speed)))
```

```
    ((xCheckEvent rotate-x- "ButtonPress") (setq trans `(Transform 'rot ,win "x" ,(- speed))))
    ((xCheckEvent rotate-y+ "ButtonPress") (setq trans `(Transform 'rot ,win "y" ,speed)))
    ((xCheckEvent rotate-y- "ButtonPress") (setq trans `(Transform 'rot ,win "y" ,(- speed))))
    ((xCheckEvent rotate-z+ "ButtonPress") (setq trans `(Transform 'rot ,win "z" ,speed)))
    ((xCheckEvent rotate-z- "ButtonPress") (setq trans `(Transform 'rot ,win "z" ,(- speed))))
    ((xCheckEvent right "ButtonPress")     (setq trans `(Transform 'trans ,win ,speed 0 0)))
    ((xCheckEvent left "ButtonPress")      (setq trans `(Transform 'trans ,win ,(- speed) 0 0)))
    ((xCheckEvent up "ButtonPress")        (setq trans `(Transform 'trans ,win 0 ,speed 0)))
    ((xCheckEvent down "ButtonPress")      (setq trans `(Transform 'trans ,win 0 ,(- speed) 0)))
    ((xCheckEvent front "ButtonPress")     (setq trans `(Transform 'trans ,win 0 0 ,speed)))
    ((xCheckEvent deep "ButtonPress")      (setq trans `(Transform 'trans ,win 0 0 ,(- speed))))
    ((xCheckEvent zoom-in "ButtonPress")   (setq trans `(Transform 'scale ,win)))
    ((xCheckEvent zoom-out "ButtonPress")  (setq trans `(Transform 'scale ,win t)))
    ((xCheckEvent stop "ButtonPress"))
    (t
     (setq run t)
     (eval op-loop))))

  (eval op-end)
  trans)
;
; Changement de la Vitesse
;
(defun IncrementSpeed (win i) IncrementSpeed
  (let (n-s (+ i (get Sdrive-Window 'speed-value)))
    (put Sdrive-Window 'speed-value n-s)
    (xSetValues (get Sdrive-Window 'speed) "label" (strcat n-s)))
  t)

                                          ; ************************* ; ;
                                          ; Incrementation des valeurs ; ;
                                          ; ************************* ; ;

(defun IncrValue (what win n mult)        IncrValue
  (SetSdrive win)
  (let (r-data)
    (setq r-data (get win what))
    (ifn r-data (setq r-data '(0 0 0)))
    (let ((v1) (v2) (v3))
      (if (numbp r-data) (setq v-1 (+ (* mult (get Sdrive-Window 'speed-value)) r-data))
    (rplaca (nth n r-data) (+ (* mult (get Sdrive-Window 'speed-value)) (car (nth n r-data))))
    (setq v-1 (car r-data)
          v-2 (cadr r-data)
          v-3 (caddr r-data)))
      (cond ((eq what 'trans) (SetDialogTrans))
       ((eq what 'rot)   (SetDialogRot))
       ((eq what 'scale) (SetDialogScale))))))

                                          ; ************************* ; ;
                                          ; Saisie directe des valeurs ; ;
                                          ; ************************* ; ;

(defun SetValue (what win)       SetValue
  (SetSdrive win)
  (let (root-dialog (get Sdrive-Window 'dialog))
    (print win root-dialog)
    (let ((d-1 (get root-dialog 1)) (d-2 (get root-dialog 2)) (d-3 (get root-dialog 3)) (r-data))
      (setq r-data (get win what))
      (cond
       ((eq what 'rot) (SetRotation))
       ((eq what 'trans) (SetTranslation))
       ((eq what 'scale) (SetScale))))))

(|defmacro SetLabels (l1 l2 l3 v1 v2 v3)
  `(progn
     (xSetValues d-1 "label" " " "icon" (mkbmp ,l1) "value" (strcat ,v1))
     (xSetValues d-2 "label" " " "icon" (mkbmp ,l2) "value" (strcat ,v2))
     (xSetValues d-3 "label" " " "icon" (mkbmp ,l3) "value" (strcat ,v3))))

(defun SetTranslation ()          SetTranslation
  (ifn r-data (setq r-data '(0 0 0)))
  (SetLabels "right" "up" "front" (car r-data) (cadr r-data) (caddr r-data))
  (xAddCallback (get root-dialog 'conf) "callback"
                (strcat "(GLD|Confirm-Dialog 'GLD|" root-dialog
                        " 'GLD|trans (get 'GLD|" Sdrive-Window " 'GLD|driven))"))
  (xRealize root-dialog))
```

```
(defun SetRotation ()SetRotation
  (ifn r-data (setq r-data '(0 0 0)))
  (SetLabels "rotate-x+" "rotate-y+" "rotate-z+" (car r-data) (cadr r-data) (caddr r-data))
  (xAddCallback (get root-dialog 'conf) "callback"
                (strcat "(GLD|Confirm-Dialog 'GLD|" root-dialog
                        " 'GLD|rot (get 'GLD|" Sdrive-Window " 'GLD|driven))"))
  (xRealize root-dialog))

(defun SetScale ()     SetScale
  (ifn r-data (setq r-data FixSize))
  (SetLabels "zoom-in" "zoom-in" "zoom-in" r-data r-data r-data)
  (xAddCallback (get root-dialog 'conf) "callback"
                (strcat "(GLD|Confirm-Dialog 'GLD|" root-dialog
                        " 'GLD|scale (get 'GLD|" Sdrive-Window " 'GLD|driven))"))
  (xRealize root-dialog))

; retour des dialoges
(defun Cancel-Dialog (root)      Cancel-Dialog
   (xUnrealize root))

(defun Confirm-Dialog (root what win)     Confirm-Dialog
   (xUnrealize root)
   (let ((r-data) (v-1) (v-2) (v-3))
     (setq v-1 (implode (explode (xGetValues (get root 1) "value"))))
     (setq v-2 (implode (explode (xGetValues (get root 2) "value"))))
     (setq v-3 (implode (explode (xGetValues (get root 3) "value"))))
     (setq r-data (get win what))
     (cond ((eq what 'trans) (SetDialogTrans))
      ((eq what 'rot)   (SetDialogRot))
      ((eq what 'scale) (SetDialogScale)))))

(defun SetDialogTrans ()        SetDialogTrans
   (let ((tx (+ v-1 (car (get win 'p-add))))
    (ty (+ v-2 (cadr (get win 'p-add))))
    (tz (+ v-3 (caddr (get win 'p-add)))))
     (put win 'trans `(,tx ,ty ,tz))
     (GLimtranslate tx ty tz)))

(defun SetDialogRot ()          SetDialogRot
   (mapc `((,v-1 "X") (,v-2 "Y") (,v-3 "Z"))  (lambda (x) (GLimrotate (car x) (cadr x))))
   (put win 'rot `(,v-1 ,v-2 ,v-3)))

(defun SetDialogScale ()        SetDialogScale
   (put win 'scale v-1)
   (GLimscale v-1 v-1 v-1 FixSize))

                                        ; ********************************************** ; ;
                                        ; Zoom dirige par le bouton 4 (de droite) de la souris ; ;
                                        ; ********************************************** ; ;

(setq MotionDrawingList nil
     MotionOrig-X nil
     MotionOrig-Y nil
     MotionOrig-Root nil
     InMotionZoom nil)

(defun MouseInitZoom (win x y)          MouseInitZoom
  (cond
   (InMotionZoom
    (GLnewlist MotionDrawingList)
    (GLcolor 3 150 150 150)
    (GLpushmatrix)
    (GLloadidmatrix)
    (GLbgn "GL_LINE_LOOP")
    (GLvertex 3 MotionOrig-X MotionOrig-Y 0  x MotionOrig-Y 0  x y 0  MotionOrig-X y 0)
    (GLend)
    (GLpopmatrix)
    (GLendlist))
   (t
    (setq MotionDrawingList (GLgenlist)
     MotionOrig-Root (GLrootlist)
     MotionOrig-X x
     MotionOrig-Y y
     InMotionZoom t)
    (let (new-root (GLnewlist))
```

```
    (GLcalllist MotionOrig-Root)
    (GLcalllist MotionDrawingList)
    (GLendlist)
    (GLrootlist new-root)))))

(defun MouseSetZoom (win x y)          MouseSetZoom
  (ifn InMotionZoom nil
      (setq InMotionZoom nil)
      (GLdellist (GLrootlist MotionOrig-Root))
      (GLdellist MotionDrawingList)
      (let ((dx (abs (- x MotionOrig-X)))
       (dy (abs (- y MotionOrig-Y)))
       (tot (xGetValues win "width"))
       (sf 0))
    (if (< dx dy) (setq dx dy tot (xGetValues win "height")))
    (setq sf (/ (* tot 1.0) dx))
    (let ((t-data (get win 'trans))
        (r-data (get win 'rot))
        (s-data (get win 'scale))
        (move))
      (put win 'z-stack (cons (cons (1 t-data) (cons (2 t-data) (cons (3 t-data) nil)))
                              (cons s-data (get win 'z-stack))))
      ; translation
      (setq move (GLgetXYZ (if (< x MotionOrig-X) x MotionOrig-X) (if (< y MotionOrig-Y) y MotionOrig-Y) 0
                           (- (1 r-data)) (- (2 r-data)) (- (3 r-data))
                           FixSize FixSize FixSize s-data))
      (rplaca t-data (- (car t-data) (1 move)))
      (rplaca (cdr t-data) (- (cadr t-data) (2 move)))
      (rplaca (cddr t-data) (- (caddr t-data) (3 move)))
      (GLimtranslate (1 t-data) (2 t-data) (3 t-data))
      ; scale
      (put win 'scale (* (setq sf (* (/ s-data (* FixSize 1.0)) sf)) FixSize))
      (GLimscale sf sf sf)))))

(defun UnZoom (win) UnZoom
  (ifn (get win 'z-stack) nil
      (let ((t-d (car (get win 'z-stack)))
       (s-d (cadr (get win 'z-stack))))
    (put win 'z-stack (cddr (get win 'z-stack)))
    (GLimtranslate (1 t-d) (2 t-d) (3 t-d))
    (put win 'trans t-d)
    (GLimscale (/ s-d (* FixSize 1.0)) (/ s-d (* FixSize 1.0)) (/ s-d (* FixSize 1.0)))
    (put win 'scale s-d))))

                                        ; ********** ; ;
                                        ; Gestion de la souris ; ;
                                        ; ********** ; ;

(defun DriveButtonPress (win obj)      DriveButtonPress
    (SetSdrive win)
    (setq ButtonPress t
     ObjectPress obj))

;
; Gestion de la selection dans une fenetre :
; Selection simple = selection de la fenetre
; Programmation des action: (fonction = ButtonPress<n> ou n est le numero du bouton)
;

(defun DriveButtonRelease (win obj b x y)          DriveButtonRelease
    (SetSdrive win)
    (if (or (not ButtonPress) FromMotion) (setq FromMotion nil)
      (setq ButtonPress nil)
      (setq y (- (xGetValues win "height") y))
      (SetSdrive win)
      (cond
       ((and (= b 1) (boundp '|ButtonPress1)) (|ButtonPress1 win obj x y))
       ((and (= b 2) (boundp '|ButtonPress2)) (|ButtonPress2 win obj x y))
       ((and (= b 3) (boundp '|ButtonPress3)) (|ButtonPress3 win obj x y))
       ((and (= b 4) (boundp '|ButtonPress4)) (|ButtonPress4 win obj x y))
       ((and (= b 5) (boundp '|ButtonPress5)) (|ButtonPress5 win obj x y))
       ((and (= b 6) (boundp '|ButtonPress6)) (|ButtonPress6 win obj x y))
       ((and (= b 7) (boundp '|ButtonPress7)) (|ButtonPress7 win obj x y))
       ((= b 4) (MouseSetZoom win x y))
       (t (print win obj b x y) nil))))
```

```
;
; Gestion du deplacement par la souris
;
(defun DriveMouseMotion (win b x y)      DriveMouseMotion
   (SetSdrive win)
   (setq y (- (xGetValues win "height") y))
   (cond
    ((and (= b 1) (boundp '|ButtonMotion1)) (|ButtonMotion1 win ObjectPress x y))
    ((and (= b 2) (boundp '|ButtonMotion2)) (|ButtonMotion2 win ObjectPress x y))
    ((and (= b 3) (boundp '|ButtonMotion3)) (|ButtonMotion3 win ObjectPress x y))
    ((and (= b 4) (boundp '|ButtonMotion4)) (|ButtonMotion4 win ObjectPress x y))
    ((and (= b 5) (boundp '|ButtonMotion5)) (|ButtonMotion5 win ObjectPress x y))
    ((and (= b 6) (boundp '|ButtonMotion6)) (|ButtonMotion6 win ObjectPress x y))
    ((and (= b 7) (boundp '|ButtonMotion7)) (|ButtonMotion7 win ObjectPress x y))
    ((= b 1) (setq FromMotion t) (MouseTranslate win x y))
    ((= b 4) (MouseInitZoom win x y))))

; Deplacement en cas de Motion

(defun MouseTranslate (win x y)          MouseTranslate
  (SetSdrive win)
  (let ((t-data (get win 'trans))
   (r-data (get win 'rot))
   (s-data (get win 'scale))
   (move))
    (setq move (GLgetXYZ x y 0 (- (1 r-data)) (- (2 r-data)) (- (3 r-data))
                          FixSize FixSize FixSize s-data))
    (rplaca t-data (1 move))
    (rplaca (cdr t-data) (2 move))
    (rplaca (cddr t-data) (3 move))
    (GLimtranslate (1 t-data) (2 t-data) (3 t-data))))


                                          ; ************************ ; ;
                                          ; Changement de Projections ; ;
                                          ; ************************ ; ;

(defun sdrive-projection (win wid)       sdrive-projection
   (let (t-data (get win 'trans))
     (cond
      ((eq (get win 'proj) 'ortho)    ; passage en perspective
   (xSetValues wid "label" "Persp")
;  (if (listp (fval  '|GLdrivePerspective)) (put win 'p-add (|GLdrivePerspective t-data))
     (put win 'p-add `(,(- (/ (xGetValues win "width") 2)) ,(- (/ (xGetValues win "width") 2)) -261))
     (GLperspective 900 250 100000)
     (GLimtranslate (- (car t-data) (/ (xGetValues win "width") 2))
                          (- (cadr t-data) (/ (xGetValues win "height") 2))
                          (- (cddr t-data) 270))
;)
   (put win 'proj 'persp))
     ((eq (get win 'proj) 'persp)    ; passage en orthogonal
      (xSetValues wid "label" "Ortho")
      (GLortho 0 (xGetValues win "width") 0 (xGetValues win "height") -32000 32000)
      (GLimtranslate (car t-data) (cadr t-data) 0)
      (set (cddr t-data) 0)
      (remprop win 'p-add)
      (put win 'proj 'ortho)))))

                                          ; ***************** ; ;
                                          ; Reinitialisation
                                          ; ***************** ; ;
(defun reinit-sdrive (win)          reinit-sdrive
   (SetSdrive win)
   (put Sdrive-Window 'speed-value 1)
   (xSetValues (get Sdrive-Window 'speed) "label" "1")
   (set-vals (get win 'trans) 0 0 0)
   (set-vals (get win 'rot) 0 0 0)
   (put win 'scale FixSize)
   (GLimrotate 0 "X")
   (GLimrotate 0 "Y")
   (GLimrotate 0 "Z")
   (if (get win 'p-add)
  (GLimtranslate (+ 0 (car (get win 'p-add))) (+ 0 (cadr (get win 'p-add))) (+ 0 (caddr (get win 'p-add))))
    (GLimtranslate 0 0 0))
   (GLimscale 1 1 1))
```

```
                                        ; ****************************************** ; ;
                                        ; Control des variables de status de OPEN GL ; ;
                                        ; ****************************************** ; ;

(de xCreateDialogBox (label value expr) xCreateDialogBox
   (let (root (xCreateWidget '--AW-- "ApplicationShell" "title" label))
     (let (tmp (xCreateWidget '--DB-- "awDialog" root "label" label "value" (strcat value)))
    (let ((cmd-ok (xCreateWidget '--CW-- "awCommand" tmp "label" "Ok" "background" "#888" "foreground"
"#000"))
          (cmd-cl (xCreateWidget '--CW-- "awCommand" tmp "label" "Cancel" "background" "#888" "foreground"
"#000")))
      (setq xDialogBoxConfirm expr)
      (xAddCallback cmd-ok "callback" (strcat "(GLD|xDialogBoxConfirm t   'GLD|" tmp ")"))
      (xAddCallback cmd-cl "callback" (strcat "(GLD|xDialogBoxConfirm nil 'GLD|" tmp ")"))
      (xRealize root)))))

(de xDialogBoxConfirm (is-ok dwidget)  xDialogBoxConfirm
   (let (xDialogValue (if is-ok (xGetValues dwidget "value")))
     (|xRemove dwidget)
     (if is-ok (eval xDialogBoxConfirm))))

(de xGetDialogValue (wdialog) xGetDialogValue
   (implode (explode (xGetValues wdialog "value"))))

;
; Variables de Status
;
(GLenable "GL_DEPTH_TEST")
(GLenable "GL_DITHER")
(GLenable "GL_BLEND")

(defvar GL-Status-Menu '("Variables de Fonctionnement de OpenGL"
                         ("Polymode    " . (GLD|polymode    . GLD|Set-Polymode))
                         ("Z-Buffering " . (GLD|depth       . GLD|Toggle-Status-Value))
                         ("Dither      " . (GLD|dither      . GLD|Toggle-Status-Value))
                         ("Transparence" . (GLD|blend       . GLD|Toggle-Status-Value))
                         ("Tran. Source" . (GLD|blend-src   . GLD|Set-Blend))
                         ("Tran. Dest. " . (GLD|blend-dst   . GLD|Set-Blend))
                         ("Quitter"      . (()              . GLD|Kill-Status-Menu))))

;
; Initialisation
;
(defun GL-Status-Window ()   GL-Status-Window
  (ifn (boundp '|Methode-Menu) (eval `(include ,(strcat VLISPDIR "X-Menus.vlisp"))))
  (if (|Menu-Present Sdrive-Window 'Status-Menu) t
      (put Sdrive-Window 'polymode "GL_FILL")
      (put Sdrive-Window 'depth (GLisenable "GL_DEPTH_TEST"))
      (put Sdrive-Window 'dither (GLisenable "GL_DITHER"))
      (put Sdrive-Window 'blend  (GLisenable "GL_BLEND"))
      (GLblend "GL_SRC_ALPHA" "GL_ONE_MINUS_SRC_ALPHA")
      (put Sdrive-Window 'blend-src "GL_SRC_ALPHA")
      (put Sdrive-Window 'blend-dst "GL_ONE_MINUS_SRC_ALPHA")
      (|Methode-Menu Sdrive-Window nil 4 4 GL-Status-Menu GLD
                  "background" "#888" "foreground" "#000" "font" "a14"))))))

;
; Terminaison
;
(defun Kill-Status-Menu (obj w) (|Remove-All-Methode-Menu obj) (|xRemove w)) Kill-Status-Menu
;
; changement d'un status
;
(defun Toggle-Status-Value (obj field lw w x y)       Toggle-Status-Value
  (let (what (cond ((eq field 'depth) "GL_DEPTH_TEST")
                   ((eq field 'dither) "GL_DITHER")
                   ((eq field 'blend) "GL_BLEND")))
    (if (get obj field) (GLdisable what) (GLenable what))
    (put obj field (not (get obj field)))
    (xSetValues lw "label" (strcat (or (and (get obj field) "Oui") "Non")))))
;
; Polymode
;
(defun Set-Polymode (obj field lw x y)      Set-Polymode
  (let (root (xCreateWidget '--AS-- "ApplicationShell" "title" "Mode de dessin des polygones" "iconName"
```

```
"Blend"))
          (let (vp (xCreateWidget '--AV-- "awViewport" root "allowVert" "True" "allowHoriz" "True"
                                  "width" 100 "height" 100))
            (let (lst (xCreateWidget '--AL-- "awList" vp
                                  "list"  '("GL_FILL" "GL_LINE" "GL_POINT")
                                  "defaultColumns" 1 "forceColumns" "True"))
              (xAddCallback lst "callback" (strcat "(GLD|Set-Polymode-Value '" root " '$O '" obj " '" field " '" lw ")"))
              (xRealize root)))))

        (defun Set-Polymode-Value (rwdg val obj field lw)  Set-Polymode-Value
          (xRemoveWidget root)
          (put obj field (strcat val))
          (GLpolymode (get obj field))
          (xSetValues lw "label" val))
        ;
        ; Modification des valeurs des fonctions de blend
        ;

        (defun Set-Blend (obj field lw w x y)        Set-Blend
          (let (root (xCreateWidget '--AS-- "ApplicationShell" "title" "Fonctions de Blend" "iconName" "Blend"))
            (let (vp (xCreateWidget '--AV-- "awViewport" root "allowVert" "True" "allowHoriz" "True"
                                  "width" 100 "height" 100))
              (let (lst (xCreateWidget '--AL-- "awList" vp
                                  "list"  '("GL_ZERO" "GL_ONE"
                                            "GL_DST_COLOR" "GL_SRC_COLOR"
                                            "GL_SRC_ALPHA" "GL_DST_ALPHA"
                                            "GL_ONE_MINUS_SRC_COLOR"
"GL_ONE_MINUS_DST_COLOR"
                                            "GL_ONE_MINUS_SRC_ALPHA"
"GL_ONE_MINUS_DST_ALPHA"
                                            "GL_SRC_ALPHA_SATURATE")
                                  "defaultColumns" 1 "forceColumns" "True"))
              (xAddCallback lst "callback" (strcat "(GLD|Set-Blend-Value '" root " '$O '" obj " '" field " '" lw ")"))
              (xRealize root)))))

        (defun Set-Blend-Value (w val obj field lw)            Set-Blend-Value
          (xRemoveWidget w)
          (put obj field (strcat val))
          (GLblend (get obj 'blend-src) (get obj 'blend-dst))
          (xSetValues lw "label" val))



                                            ; ************************************************************ ; ;
                                            ; Definition des macros (precedement dans des fichiers separes ;
;
                                            ; ************************************************************ ; ;

        (defvar FixSize 100)     ; nombre de decimale pour les virgules fixes
        (defvar FromMotion nil)
        (defvar bitmap-dir (strcat VLISPDIR "bitmap/"))

        (defvar |default-font   "a14")

        (|defmacro mkbmp (n) `(strcat ,bitmap-dir ,n))

        (|defmacro MkClbk (field trans args)
          `(xAddCallback (get ',Sdrive-Window ,field)
            "callback" (strcat "(GLD|Transform 'GLD|" ,trans " (get 'GLD|" ',Sdrive-Window " 'GLD|driven) " ,args ")")))

        (|defmacro MkBtn2 (field trans)
          `(xAugment (get ',Sdrive-Window ,field)
            "<Btn2Down>" (strcat "(GLD|SetValue 'GLD|" ,trans " (get 'GLD|" ',Sdrive-Window " 'GLD|driven))")))

        (|defmacro MkBtn3 (field trans n mult)
          `(xAugment (get ',Sdrive-Window ,field)
            "<Btn3Down>" (strcat "(GLD|IncrValue 'GLD|" ,trans " (get 'GLD|" ',Sdrive-Window " 'GLD|driven) " ,n " "
,mult ")"
        ))

        (|defmacro NewCW (name bmp)
          `(put ',Sdrive-Window ,name (xCreateWidget '--CW-- "awCommand" form "bitmap" (mkbmp ,bmp)
                                  "font" "a14" "background" "#888" "foreground" "#000")))

        (|defmacro NewCBK (label func args)
          `(xAddCallback (xCreateWidget '--CW-- "awCommand" form "font" "a14" "label" ,label "background" "#888"
```

```
"foreground" "#000")
                "callback" (strcat "(GLD|" ,func " (get 'GLD|" ',Sdrive-Window " 'GLD|driven) " ,args ")")))

(de NewW (obj type label . w-values)        NewW
  (put Sdrive-Window obj (eval `(xCreateWidget '--W-- type form ;"font" "a14"; ; "label" label ,@w-values))))

(|defmacro set-vals (v v1 v2 v3)
  `(progn
    (set ,v ,v1) (set (cdr ,v) ,v2) (set (cddr ,v) ,v3)))

(package)
```