USTL LIFL U.R.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° Ordre : 769 Année : 1991

THESE

présentée à

L'Université des Sciences et Technologies de LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Olivier LAGNEAU

IA et CAO : ETUDE DE PROBLEMES LIES AUX LANGAGES ET ETUDE DE CAS EN SCHEMATIQUE

Soutenue le 25 Septembre 1991 devant la commission d'examen.

Membres du Jury :

Président :	Michel MERIAUX
Rapporteur :	Gérard COMYN
Rapporteur :	Yvon GARDAN
Examinateur :	Eric DELIGNIERES
Examinateur :	Cléo JULLIEN

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

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

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

PROFESSEUR EMERITE

M. A. LEBRUN

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

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

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

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

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

PROFESSEURS - 1ère CLASSE

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

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

PROFESSEURS - 2ème CLASSE

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

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

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

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

Je remercie Michel Mériaux de me faire l'honneur de présider le jury de cette thèse. Je lui suis très reconnaissant de m'avoir proposé ce sujet intéressant et je lui exprime toute ma gratitude pour avoir rendu possible ce travail par l'amitié et la confiance constantes qu'il m'a manifestées, par son soutien sans cesse renouvelé et par le soin attentif avec lequel il a dirigé mes travaux. Qu'il soit également remercié pour ses nombreuses idées critiques, ainsi que pour son support essentiel lors des démarches administratives.

Je remercie Gérard Comyn d'avoir accepté d'être rapporteur de cette thèse. Qu'il reçoive toute ma reconnaissance pour avoir initié en grande partie mon travail, pour ses nombreuses remarques et idées critiques, pour son suivi constant et ses grandes qualités humaines. Ce travail n'aurait pas pu être mené à bien sans sa compréhension et l'intérêt attentif qu'il m'a manifesté depuis le début de cette étude.

Je remercie Yvon Gardan qui s'est intéressé à mon travail et m'a fait l'honneur de le juger, ainsi que du temps qu'il a bien voulu consacrer à son examen.

Je remercie Eric Delignières d'avoir accepté de participer à ce jury et de l'intérêt qu'il a toujours porté à mes travaux, à la fois comme examinateur et comme collègue de travail dans la société METADESIGN.

Je remercie Cléo Jullien de l'intérêt qu'elle porte à mon travail et de me faire l'honneur de participer à ce jury.

Je tiens également à remercier les collègues de la société CAP GEMINI INNOVATION avec qui j'ai pu entretenir des discussions intéressantes et enrichissantes sur le modèle MAPS, en particulier Marie-Stéphane Doize, Fano Ramparany et Ariane Sorba pour leur lecture attentive de ce document et leurs remarques judicieuses.

Je tiens aussi à remercier Eric Verdin et André Dalmasso, anciens collègues de METADESIGN, pour leur contribution importante à la réalisation des interfaçes logicielles décrites dans ce document.

Je remercie enfin tous ceux qui m'ont aidé ou soutenu d'une manière ou d'une autre pendant ces années de thèse. Je pense à mes collègues de travail, aussi bien dans la société METADESIGN qu'à CAP GEMINI INNOVATION. Je pense aussi à ma famille et à mon épouse Corinne dont la patience et l'optimisme sont sans limite.

Le travail décrit dans cette thèse a été réalisé dans le cadre du contrat de recherche CIFRE No 315/84 dont j'ai bénéficié, au sein de la société METADESIGN et du Laboratoire d'Informatique Fondamentale de Lille 1 (U.A. 369 du C.N.R.S). Je remercie Messieurs Philippe Daubresse et Christian Traisnel, respectivement directeur technique et président directeur général de METADESIGN, de m'avoir proposé ce sujet intéressant. Je tiens également à remercier Messieurs Maurice Schlumberger et Paul Decitre, respectivement directeur scientifique et directeur du centre de recherche de CAP GEMINI INNOVATION Grenoble de leur soutien, ainsi que de m'avoir permis d'utiliser les logiciels et matériels nécessaires à la rédaction de cette thèse, tâche qui a été en partie réalisée au sein de cette dernière société.

A Corinne et Antoine,

Introduction

I CAO : CONCEPTS ET DEFINITIONS

II : INTERFACAGE IA-CAO

III MAPS : UN MODELE DECENTRALISE POUR SYSTEME DE PLACEMENT INTERACTIF

Liste des figures numérotées

Cette thèse est structurée en trois chapitres. Le premier porte sur une introduction des concepts et caractères de la CAO. Le second concerne l'interfaçage logiciel entre programmes d'intelligence artificielle et programmes procéduraux. Le dernier chapitre concerne l'application du paradigme de décentralisation issu des techniques de l'intelligence artificielle distribuée pour la réalisation d'un système d'aide à la conception destiné à assister l'utilisateur d'un logiciel de schématique dans les tâches d'agencement spatial de composants.
Ces trois chapitres peuvent être abordés de façon indépendante.

Introduction

Les systèmes de Conception Assistée par Ordinateur (C.A.O) actuels fournissent des outils puissants d'aide à la spécification et à la description de produit : logiciels de schématique permettant d'éditer graphiquement et interactivement une représentation descriptive de l'objet à concevoir, logiciels de modélisation numérique pour fournir une représentation visuelle réaliste, logiciels de simulation pour aider le concepteur à évaluer le produit virtuel manipulé par ordinateur. Cependant ces outils sont finalement insuffisants pour aider l'expert dans l'activité de conception elle-même. Il existe des logiciels automatiques spécialisés dans la réalisation d'une tâche particulière de conception, comme les logiciels de placement-routage en électronique par exemple. Malheureusement ces logiciels basés sur des modèles numériques et algorithmiques, sont souvent incapables de fournir une solution tout à fait satisfaisante, car ils n'intègrent pas le savoir faire des experts du domaine.. Les logiciels de CAO n'assistent donc pas vraiment les concepteurs dans leur tâche.

Pour pouvoir réellement aider l'utilisateur d'un système de CAO dans une activité de conception, il faut introduire ou intégrer des capacités de raisonnement dans ce système. Les recherches actuelles ou récentes sur l'intégration de l'Intelligence Artificielle (I.A.) dans les systèmes de CAO concernent d'une part l'utilisation de techniques d'IA pour fabriquer des logiciels automatiques de conception [Dejesus & Callan 85, Favard 89, Orchampt 87, Tsang 87], et d'autre part l'intégration effective de l'IA et de la CAO [Gardan 91], en particulier pour réaliser des systèmes assistant l'utilisateur dans ses tâches de conception [Trousse 89].

Le travail décrit dans cette thèse porte à la fois sur les problèmes d'intégration de programmes codés en langage d'intelligence artificielle (comme LISP ou PROLOG) avec des langages procéduraux classiques (C, FORTRAN, ADA) utilisés dans les systèmes de CAO classiques, et sur la réalisation d'une interface usager dotée d'un système d'aide à la conception. L'architecture de cette interface est basée sur un paradigme de décentralisation issu des techniques d'intelligence artificielle distribuée.

Un système d'aide à la conception au sens où nous l'entendons ici peut être défini comme un système capable de conseiller le concepteur d'un produit : en signalant des incohérences de conception, en conseillant des choix particuliers de solutions, en réalisant les tâches routinières de conception que l'utilisateur est amené à effectuer lors de modifications ou de reconceptions. Le système d'aide que nous avons réalisé porte sur l'aménagement spatial de composants. Ce travail d'aménagement est très fréquent lors de l'utilisation de logiciels de schématique, où l'utilisateur doit disposer dans l'espace (ou assembler) des éléments constituants de l'objet à concevoir.

Au cours d'une étude réalisée en 85 dans la société METADESIGN [Carre 85a 85b] concernant la faisabilité d'un système expert de placement-routage pour carte de circuit imprimé, il a été constaté la nécessité de disposer d'une représentation des connaissances adaptée au problème de la conception électronique, ce qui a donné lieu à des travaux et résultats concernant un modèle de représentation "multiple et évolutive" des objets [Carre & Comyn 87a, 87b, Carre 89]. Nous avons d'autre part identifié le besoin de communications bidirectionnelles entre programmes écrits avec des langages d'IA et avec des langages

procéduraux, pour pouvoir intégrer les deux techniques, tout en réutilisant les logiciels existants de la CAO.

Cette intégration et cette communication de base peuvent être réalisées au sein d'un seul processus contenant à la fois les parties IA et procédurales, auquel cas la capacité de communication directe entre les deux langages doit être étudiée. Ceci implique un couplage fort entre la partie IA et la partie procédurale. Mais on peut également envisager un couplage faible pour lequel on utilisera des facilités fournies par le système d'exploitation hôte, et qui soient autant que possible portables.

Nous proposons ici des modèles de communication entre des parties IA et procédurales. Nous avons identifié ce que nous appelons l'interfaçage direct et l'interfaçage séparé, pour lesquels nous mettons en évidence leurs avantages et inconvénients lors de la réalisation des communications. Nous décrivons également nos expériences d'interfaçage pour divers langages IA et procéduraux. Il s'agit en quelque sorte de coupler les deux types de programmes pour qu'ils puissent être utilisés ensemble.

C'est finalement le choix d'un type de couplage fort ou faible entre les parties IA et procédurale, et les caractéristiques spécifiques des deux types d'interfaçage qui devraient déterminer le choix entre une interface de type direct ou indirect. Nous constatons qu'un couplage faible (interface indirecte) facilite la réalisation de l'interface au prix d'une efficacité faible et d'une consommation mémoire élevée. A l'inverse le couplage fort (interface directe) favorise la vitesse d'exécution et l'économie de mémoire, mais pose de gros problèmes de compatibilité entre langages dus à la fois à la nature des langages et à leur implémentation par les constructeurs de logiciels.

Le point de départ de nos travaux en matière d'aide à la conception vient d'une observation initiale : les concepteurs utilisant un système de CAO, et en particulier un logiciel de schématique doivent faire de nombreuses adaptations de disposition spatiale de composants, pour respecter un équilibre nécessaire des contraintes géométriques et fonctionnelles pesant sur eux. Les systèmes de CAO ne peuvent prendre en compte, de par leur caractère algorithmique, les contraintes issues du savoir-faire ou de la technologie du domaine d'application. Ce problème est amplifié à partir d'un certain niveau de complexité géométrique (en nombre de composants ou en fonction de leur forme), et cette nécessité d'aménagement spatial devient alors constante. Les rectifications d'aménagement spatial deviennent alors des tâches routinières fastidieuses pour le concepteur, parfois difficiles à effectuer.

Dans le cas de la CAO en mécanique ou en architecture, le problème de l'aménagement spatial fait partie des aspects fonctionnels que doit satisfaire le produit. Dans d'autre domaines tels que l'électronique, c'est plutôt les limitations de fabrication ou d'encombrement physique de la carte de circuit imprimé ou du circuit intégré qui entraîne ces contraintes spatiales et nécessite un réarrangement constant de la disposition des éléments dans l'espace.

Le caractère de modification locale nous a guidé pour réaliser notre modèle MAPS (Multi-Agent model for interactive Placement System), qui définit un cadre de représentation et de résolution décentralisés pour le problème de l'aménagement spatial de composants dans un

espace à deux dimensions. D'autre part, nous avons également voulu profiter de la modularité naturelle des composants en choisissant un modèle issu directement de la technologie de l'IA décentralisée [Demazeau 90], c'est-à-dire basé sur la notion d'agent élémentaire. Le système est alors constitué d'un ensemble d'agents élémentaires travaillant ensemble à la réalisation d'une tâche donnée. On a alors un système multi-agents, où chaque composant à (dé)placer est un agent du système.

Il fallait néanmoins, pour proposer un système convivial et évaluer un tel système d'aide, disposer d'une interface utilisateur classique pour un système de CAO (mais limitée dans ses fonctions au problème de l'agencement spatial). C'est pourquoi nous avons réalisé notre implantation à partir du système Smalltalk qui permet un prototypage rapide et dispose d'une boite à outils de fabrication d'interfaces intéressante. De plus, le langage Smalltalk est un langage à objet, et peut être adapté au paradigme d'agent ou d'acteur [Briot 89] que nous avons choisi. Remarquons toutefois que ce prototype était destiné par la suite à être porté dans un langage procédural et intégré au logiciel de CAO électronique réalisé et vendu par la société METADESIGN contractante du contrat de recherche CIFRE; ceci a influé quelquefois sur les choix techniques de réalisation de la maquette.

Enfin il est nécessaire, pour disposer réellement d'un système d'aide à la conception en aménagement spatial, de fabriquer des systèmes de résolution de conflits spatiaux, qui seront au coeur du système d'aide, faisant appel à la fois au modèle décentralisé MAPS et aux connaissances du domaine d'aplication.

Nous décrivons donc ici à la fois un modèle décentralisé de représentation des connaissances pour l'aménagement spatial appelé MAPS, deux "moteurs" de résolution de conflit géométrique basés sur des stratégies différentes mais pour lesquelles on peut faire un parallèle avec le comportement naturel du concepteur utilisant un logiciel de schématique, et enfin une interface utilisateur intégrant le modèle MAPS.

Les contributions de ce travail sont les suivantes :

- la faisabilité d'un système d'aménagement spatial interactif basé sur un modèle distribué,
- la "socialisation" de composants géométriques vus comme des agents par la notion de voisinage et la création d'un modèle distribué adapté (MAPS),
- la gestion décentralisée de ce voisinage géométrique, ainsi que l'identification de propriétés intéressantes pour celui-ci,
- l'étude de deux stratégies de base pour la résolution de conflit géométrique, et de leurs moteurs de résolution associés,
- la nécessité de fabriquer des mécanismes de contrôle évolués pour résoudre le problème des cycles dans la résolution distribuée de conflit géométrique.

Le document est structuré comme suit : introduction et description générale du problème de la CAO, description de nos travaux en matière d'interfaçage IA-CAO, système d'aide à la conception basé sur MAPS.

Nous décrivons tout d'abord dans le premier chapitre la problématique des systèmes de CAO, *du point de vue de l'activité de conception, des objectifs initiaux de la CAO classique*, de la structure habituelle de tels systèmes, et concluons ce chapitre en mettant en lumière les problèmes actuels des systèmes de CAO, du point de vue informatique ou de leur utilisation.

Après avoir rappelé les raisons de l'étude de l'interfaçage logiciel entre parties IA et CAO, nous précisons divers aspects de l'interfaçage IA-CAO, explicitons le problème de la communication entre une partie IA et une partie algorithmique, décrivons les modèles d'interface directe et indirecte que nous définissons, puis en donnons deux exemples typiques en précisant les mécanismes internes de communication. La partie suivante est consacrée aux études d'interfaçage IA-CAO que nous avons réalisées avec les langages LISP, PROLOG, et SMALLTALK. Un paragraphe traite du problème particulier des langages à objets hybrides, tels que C++, ou OBJVLISP. Nous tentons enfin d'évaluer brièvement les diverses possibilités de connexion entre logiciels IA et logiciels procéduraux.

Le dernier chapitre concerne l'aide à la conception. Nous présentons tout d'abord les caractères courants d'une interface de système de CAO, puis nous présentons le problème de l'aide à la conception et ses aspects spécifiques provenant du domaine de la CAO. Nous décrivons ensuite le modèle MAPS, les raisons du choix distribué pour la représentation du problème, la socialisation des composants obtenue par la notion de voisinage, et introduisons le modèle de voisinage que nous avons choisi pour MAPS.

Le modèle de voisinage choisi nous a permis de distribuer la gestion de celui-ci pour respecter le caractère décentralisé de MAPS. Nous décrivons formellement le voisinage choisi, ses propriétés et expliquons formellement comment la gestion du voisinage peut être gérée de façon décentralisée. En particulier les algorithmes de création de voisinage, d'apparition et de disparition de voisins sont expliqués très précisément car ils sont essentiels pour être capable de toujours maintenir la cohérence des voisinages et surtout des relations entre voisins.

Nous avons réalisé en Smalltalk une maquette d'interface intégrant le modèle MAPS et la gestion distribuée du voisinage choisi. Nous décrivons son implantation et les classes Smalltalk qui la composent, ainsi que les relations étroites entre cette implantation et notre modèle.

La dernière partie de ce chapitre concerne notre travail sur la résolution distribuée de conflit géométrique. Nous présentons deux stratégies que nous avons envisagées et qui se rapprochent de deux types de comportement naturel des concepteurs : soit en travaillant d'abord sur le composant en conflit, puis en agissant sur ses voisins, soit en raisonnant à une profondeur plus importante par tentatives d'évaluation des conséquences d'une action ou des possibilités de solution pour les composant voisins. Finalement nous faisons une évaluation des deux stratégies de résolution en discutant leur avantages et inconvénients. Nous

considérons également la prise en compte des contraintes foncionnelles et pas seulement géométriques, l'évolution de la résolution et en particulier le problème des cycles. Nous comparons finalement le système de résolution vis-à-vis des systèmes courants de satisfaction de contraintes portant sur des problèmes d'agencement spatial [Andre 86, Orchampt 87, Du verdier 91].

Nous concluons en faisant le point sur les résultats issus de cette thèse et sur les possibilités d'études et de réalisation futures.

Introduction au chapitre I.

Le but de ce premier chapitre est de présenter, d'une façon synthétique et succincte, les concepts reconnus généralement par tous les spécialistes des techniques de Conception Assistée par Ordinateur, ainsi que d'introduire un modèle de référence simple pour la constitution logicielle d'un système de CAO, modèle qui soit adapté aux travaux décrits dans ce document. Puis nous citerons certaines déficiences des systèmes de CAO actuels qui apparaissent aujourd'hui suffisamment limitatives pour mettre en cause la fonction principale d'un système d'aide à la conception. Nous terminons par la présentation de trois exemples de documents typiques de systèmes de CAO.

Concepts des systèmes de CAO :

Nous introduisons tout d'abord le contexte d'utilisation des systèmes de CAO. Ceci concerne les paragraphes I.1 à I.6.

Définir la CAO n'est pas chose facile, car ce concept peut être considéré sous beaucoup de points de vue différents suivant l'application choisie et les fonctions que l'on considère comme importantes pour ce genre de logiciel; ainsi chaque point de vue particulier pourrait faire l'objet d'une définition. Nous nous contentons ici de définir cette technique par sa caractéristique universellement reconnue : construire ou concevoir, à l'aide d'une simulation informatique, un produit ou un objet quelconque (I.1).

Les paragraphes I.2 à I.5 introduisent le contexte industriel dans lequel sont plongés les systèmes de CAO, autant du point de vue de la vie d'un produit (I.2), de sa conception "à la main" (I.3), que des méthodes de travail couramment adoptées dans le milieu industriel par les concepteurs pour arriver à leurs fins (I.4, I.5). Notons ici que bien que nous parlions dans le texte d'un concepteur, dans la réalité il s'agira plus souvent d'une équipe de spécialistes travaillant de concert et se communiquant des informations. Cette assimilation outrancière d'une équipe à une personne n'a d'autre but que de simplifier l'exposé.

Nous précisons ensuite les objectifs que doivent atteindre ces logiciels. Leur intérêt principal réside dans l'économie et la qualité de la conception-réalisation du produit industriel. Ceci introduit naturellement des sous-objectifs dus à l'introduction de la technique CAO dans le processus industriel classique de conception (I.6.1 à I.6.4), ainsi que d'autres objectifs créés indirectement par l'apport de l'informatique (I.6.4 et I.6.5).

Structure et contenu des systèmes de CAO :

Enfin, en considérant les systèmes de CAO comme des entités avant tout logicielles, nous décrivons de façon volontairement simplifiée leur structure et leur contenu.

Nous insistons d'abord sur la communication, essentielle dans ce genre de systèmes, entre l'homme et la machine : de la même façon qu'il est difficile, mais pas impossible, de scier une bûche avec un couteau cranté, on ne pourra réaliser que difficilement la conception d'un produit avec un système informatique, quand l'ergonomie de celui-ci est déficiente (I.7.1.1 à I.7.1.3).

Vient ensuite le fait que simuler un objet avec une machine nécessite beaucoup de traitements de gestion des données : pour décrire l'objet, pour le fabriquer, pour le documenter (I.7.2).

Après quoi nous décrivons la réalisation interne de ces systèmes : une multitude de logiciels et d'algorithmes divers. Ils coopèrent entre eux ou agissent seuls (I.7.3). Nous décrivons finalement ce que sont ces logiciels internes par leur(s) fonction(s) au sein du système de CAO(I.7.4).

Le modèle adopté dans ce document :

Nous décrivons dans cette partie un système de CAO simplifié, sans aucune référence à un domaine d'application particulier, par ses fonctionnalités générales. Il s'agit de classer les entités logicielles par leurs fonctions. Cette description nous servira tout au long du document pour introduire et localiser de façon précise nos travaux dans ce schéma (I.8).

Nous précisons ensuite quelques uns des défauts et problèmes des systèmes de CAO dans leur support de conception d'objets complexes, ainsi que quelques domaines de recherche et d'évolution de ces logiciels (I.9).

Exemples de documents CAO :

Nous donnons finalement trois exemples de documents tels qu'ils sont ou pourraient être produits par un système CAO sur une table traçante :

- Un projet de circuit imprimé électronique (Fig I-7 à I-10).
- Une vue éclatée d'une pièce mécanique (Fig I-11).
- Une élévation d'un bâtiment avec son plan (Fig I-12).
- Les divers documents d'un projet de CAO électronique (Fig I-13 à I-23)

I. CAO : CONCEPTS ET DEFINITIONS

I.1. Tentative de définition

Si l'on pose aux techniciens habitués à utiliser un système de Conception Assistée par Ordinateur (C.A.O.) la question "qu'est ce que la CAO ?" , on obtiendra souvent une réponse spécifique pour chacun d'eux.

Pour certains, il s'agira de l'utilisation d'un ordinateur avec un périphérique graphique en vue d'obtenir un dessin (schéma électronique, plan d'architecture); d'autres y ajouteront les contrôles et les simulations d'un produit par ordinateur; d'autres encore penseront d'abord à la préparation de documents techniques en vue d'une phase ultérieure de fabrication. Leur réponse dépendra en fait de leur métier et des logiciels qu'ils utilisent.

Mais si chacun possède sa propre définition de la chose, tout le monde est d'accord pour affirmer qu'il s'agit d'un système où l'homme et la machine informatique sont associés en vue de faciliter la conception et la fabrication d'un produit industriel [Gardan 86b, Giambasi & al 83, Lebahar 85].

Citons ici D.T. ROSS, l'un des pionniers :

"La CAO est une technique dans laquelle l'homme et l'ordinateur sont rassemblés pour la solution de problèmes techniques en une équipe qui allie étroitement les meilleures qualités de chacun d'eux, de telle manière que l'équipe travaille mieux que chacun séparément."

Il s'agit donc de mettre à profit au mieux les qualités de chacun : pour l'homme l'analyse, la synthèse et la création; pour l'ordinateur le stockage des informations et le traitement routinier des données à manipuler pour concevoir un produit (documents de description, vérifications parcellaires, simulations et vérifications à chaque étape de la conception.

Notons dès à présent que chaque système de CAO est dédié à un domaine précis de conception d'un produit industriel. Les domaines d'applications les plus communément abordés par ces systèmes sont : la mécanique [Gardan 86a], l'électronique, l'architecture [Lebahar 85, Quintrand & others 85], l'industrie textile; ce sont les domaines d'application auxquels nous ferons allusion dans ce document.

Note : nous utiliserons toujours l'abréviation "SCAO" comme contraction de "*système(s) de CAO*".

I.2. Les phases de développement d'un produit industriel

Pour les domaines précités, le développement du produit peut être découpé en étapes élémentaires regroupées ici en deux phases [Chaillou 86, Gardan 86b, Giambasi & al 83].

Phase A : (la conception est majoritaire)

- Etablissement du cahier des charges : fonctionnalités et performances désirées du produit.
- Vérification des spécifications : complétude et cohérence du produit.
- Schéma de principe : description du produit, sans détails de réalisation, avec vérification globale du fonctionnement.
- Constitution physique : choix des composants (matériaux et pièces).
- Vérifications technologiques avec remise en cause éventuelle du cahier des charges et/ou des spécifications du produit.

Phase B : (la fabrication est majoritaire)

- Préparation pour la fabrication : listes et documents nécessaires.
- Documentation relative au produit.
- Vérification des données de fabrication.
- Fabrication du produit.
- Contrôle du produit fini.
- Maintenance.

L'enchaînement des étapes de développement est communément le suivant :

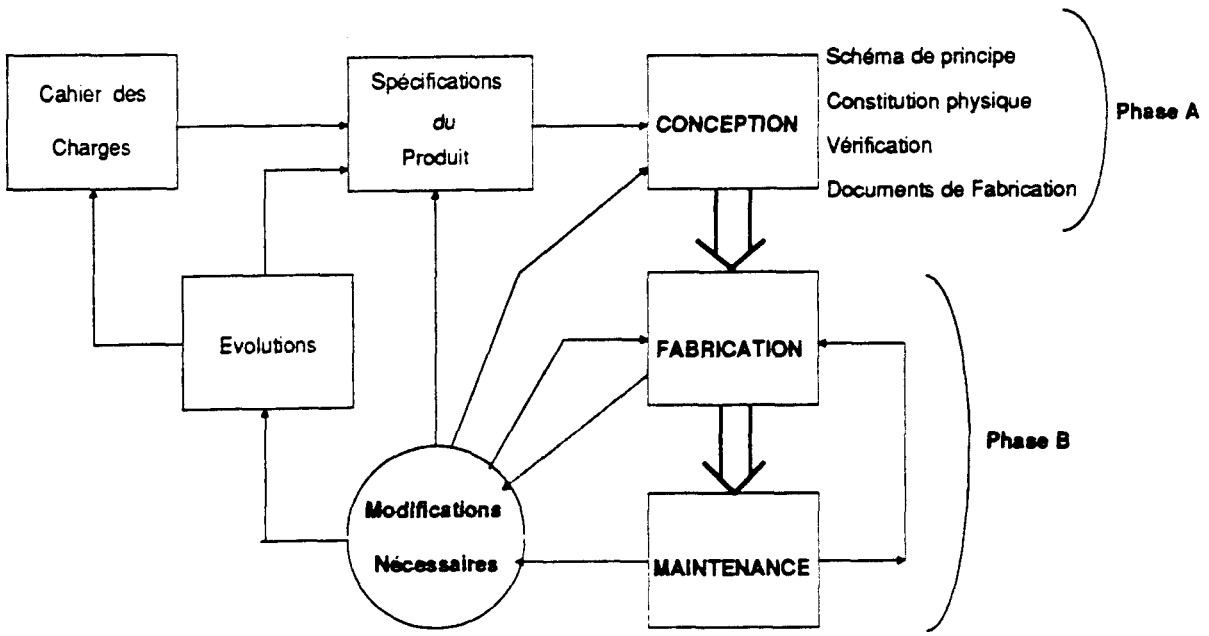


Fig I-1 : *cycle de vie d'un produit industriel*

En fait, les systèmes de CAO concernent essentiellement les étapes de la phase A, que nous appellerons la "**phase de conception**". Les logiciels couvrant en partie les aspects de la phase B, dite "**phase de fabrication**", portent des noms différents :

- CFAO, "Conception et Fabrication par Ordinateur", qui regroupe l'ensemble du processus de réalisation d'un produit.
- FAO, "Fabrication Assistée par Ordinateur", qui concerne essentiellement la fabrication et la maintenance.
- GPAO, "Gestion de Production Assistée par Ordinateur", dont le but est d'améliorer et d'automatiser les techniques de production.

Enfin on a l'habitude de regrouper les logiciels d'aide et de conception touchant divers domaines (ingénierie, enseignement, publication, ...) sous le vocable XAO.

I.3. Les étapes de la conception d'un produit

On distingue généralement quatre étapes distinctes [Giambasi & al 83] :

- Le projet conceptuel où l'on spécifie les fonctions du produit au vu de considérations commerciales, en particulier à partir des études de marché.
- L'analyse du projet.
- La constitution détaillée du produit en composants et son évaluation.
- La documentation relative à l'étude.

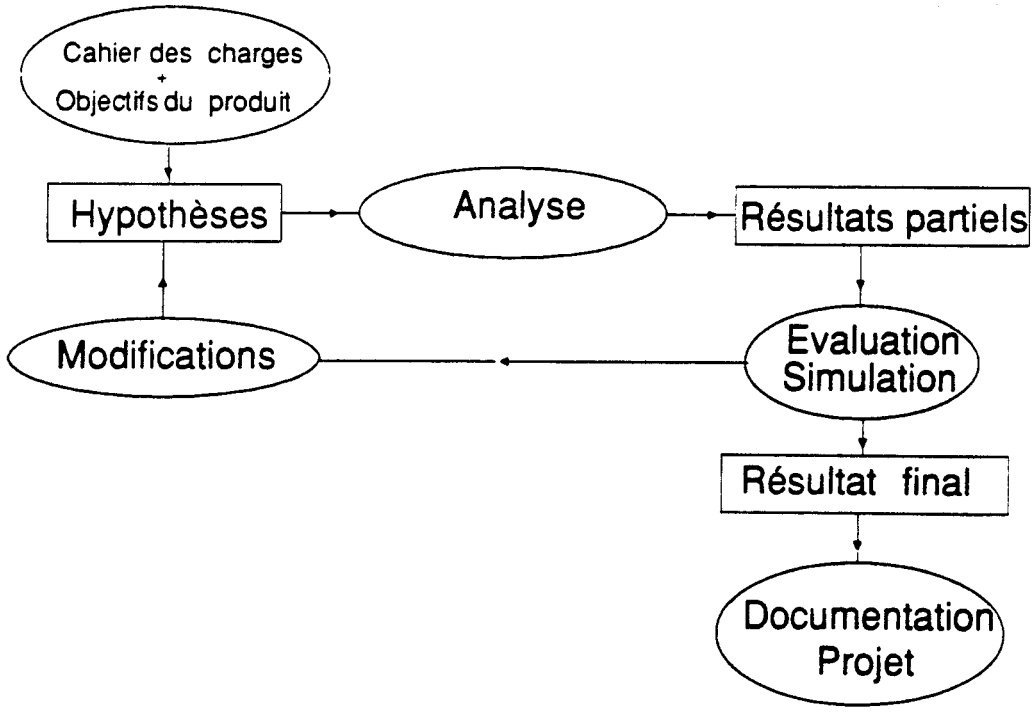


Fig I-2 : *Les phases de conception d'un projet*

I.4. Démarche ascendante et descendante dans la conception

Le projet initial étant souvent trop complexe pour être abordé d'emblée dans sa globalité, on le décompose généralement en sous-ensembles spécialisés par le rôle qu'ils jouent dans le produit et par leur caractère matériel. On décrit alors le projet par une représentation arborescente avec ses constituants, qu'ils représentent ou non des parties réelles du produit.

Ceci engendre deux méthodes idéales de conception [Giambasi & al 83, Rueher 80] :

- La **démarche descendante** qui consiste à spécifier le produit par niveaux de plus en plus précis vis à vis des constituants de l'objet conçu. Elle permet une progression logique dans la conception mais impose qu'il n'y ait jamais (ou le moins possible) de remise en cause dans ce travail.
- La **démarche ascendante** qui consiste à élaborer l'objet complet à partir de sous-ensembles simples qui ont été créés précédemment. C'est la démarche inverse de la précédente. Elle peut être plus rapide que la première, à condition que la tâche de conception soit confiée à un expert, que celui-ci connaisse parfaitement le type des problèmes qu'il rencontrera dans le projet courant et qu'il sache les résoudre aisément. Ces conditions assez draconiennes expliquent l'insuccès relatif de cette méthode.

Bien souvent, vu la complexité des problèmes à résoudre, les deux démarches coexistent au cours de la conception du produit. C'est pourquoi les systèmes de CAO doivent permettre d'utiliser ces deux démarches et aider le concepteur à le faire.

L'une ou l'autre des méthodes peut être utilisée indifféremment, à condition que l'objet à concevoir puisse être décrit aisément en sous-ensembles structurels et /ou fonctionnels, lesquels sont indépendants entre eux.

Pour concevoir l'objet, on en fait une description cohérente à l'aide de ses sous-ensembles, on conçoit chacun d'eux indépendamment des autres, et on recompose le tout. Comme il n'y a pas de remise en cause globale du projet (c'est du moins le but recherché), on peut utiliser à sa guise et de manière imbriquée la méthode ascendante ou descendante. Nous insistons sur le fait que la méthode de conception ascendante est réservée aux experts de conception dans le domaine d'application visé.

Remarques sur ces deux méthodes :

La démarche descendante permet de réaliser la conception de façon progressive et logique. Elle oblige l'utilisateur à adopter un plan de travail précis, qui devient par la suite une aide précieuse si l'objet est très complexe ou si le concepteur n'est pas un spécialiste. Malheureusement elle n'est quasiment jamais applicable car la réalisation de sous-ensembles suffisamment indépendants pour ne pas engendrer d'incompatibilités avec l'ensemble est souvent impossible. Le concepteur doit faire des retours en arrière; le schéma de conception descendante est alors brisé.

La démarche ascendante n'a pas les avantages de la méthode précédente pour guider le travail, car le concepteur est immédiatement confronté aux problèmes les plus ardus et doit décrire directement les liens entre sous-ensembles. Cependant, les spécialistes du problème de conception visé, roués aux problèmes classiques et spécifiques de l'application, sont capables d'utiliser une telle méthode car ils entrevoient d'avance, de par leur expérience, les obstacles qu'ils vont rencontrer et la façon de les résoudre.

Finalement, dans la pratique, aucune de ces deux méthodes n'est utilisable seule, sauf pour des cas très précis et assez simples. Les objets à concevoir sont trop complexes et les relations entre sous-ensembles trop nombreuses pour pouvoir appliquer d'une manière puriste de telles démarches.

En résumé, nous retiendrons que les méthodes de conception ont pour but de [Giambasi & al 83, Guillot & al 85] :

- Réduire la durée du processus de conception.
- Forcer le concepteur à adopter une démarche formalisée.
- Permettre la détection rapide des erreurs.
- Favoriser une approche naturelle et proche du raisonnement du concepteur [Lebahar 85, Moulin 90].

I.5. Les modèles, supports pour la conception

Pour faire son travail, le concepteur se réfère toujours à une représentation concrète ou abstraite du projet ou d'une partie de celui-ci, selon qu'il réalise une partie fonctionnelle ou physique du produit; cette représentation temporaire est appelée "**modèle**" [Gardan 86a 86b, Giambasi & al 83, Lebahar 85, Quintrand & others 85].

Le modèle est un support de base pour le concepteur, lui permettant de manipuler ou de raisonner sur l'objet à concevoir. Grâce à lui, il peut faire intervenir, au cours du processus d'élaboration, des contraintes physiques, mathématiques, logiques, ou plus spécifiquement liées au domaine d'utilisation de l'objet ou à sa fabrication. Le modèle à pour but de donner une référence adaptée au travail de conception. Chaque étape de l'élaboration d'un produit nécessite un modèle particulier.

On classifie communément les modèles couramment utilisés de la manière suivante :

- **Modèle descriptif** : il contient la description de l'objet ; plus généralement des caractéristiques géométriques et technologiques. On l'appelle aussi "maquette virtuelle" dans les SCAO. Ce pourra être une vue graphique (plan d'architecture) ou alphanumérique (liste des connexions d'un circuit).
- **Modèle de fonctionnement** : il explique comment fonctionne le projet ou l'un de ses sous-ensembles. Il permet des simulations, des optimisations. Certains l'appellent "schéma de principe".
- **Modèle de communication** : il est destiné à permettre la communication entre l'utilisateur et le système. Par exemple, c'est une représentation symbolique d'un circuit électronique.

Cette classification est basée sur la fonction des modèles plutôt que sur leur contenu. Dans la réalité des SCAO, un modèle de fonctionnement peut aussi bien être représenté par un schéma que par une documentation détaillée, cela dépendra du projet étudié et de son domaine d'application.

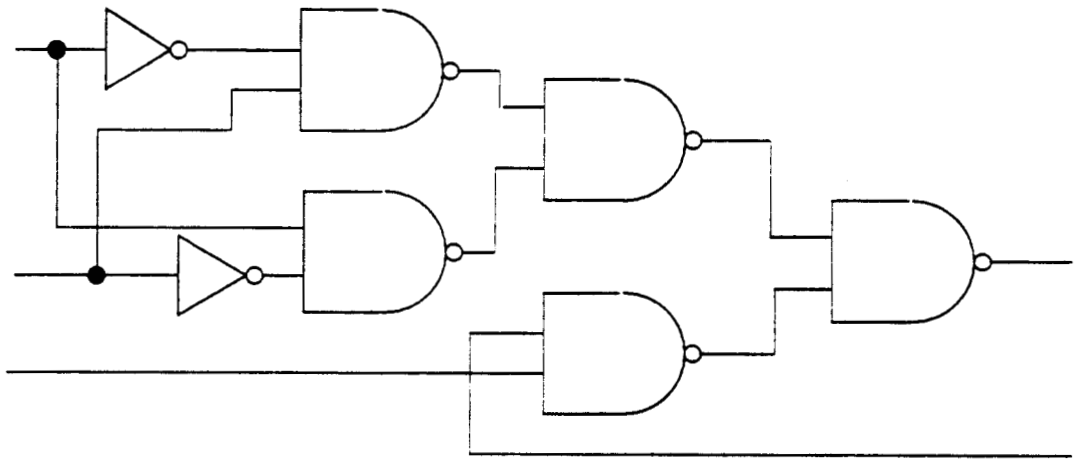
Pour éclaircir quelque peu notre propos, nous montrons ci-après, pour un transformateur de puissance, projet en électrotechnique, les différents modèles que l'on peut envisager .

Exemples :

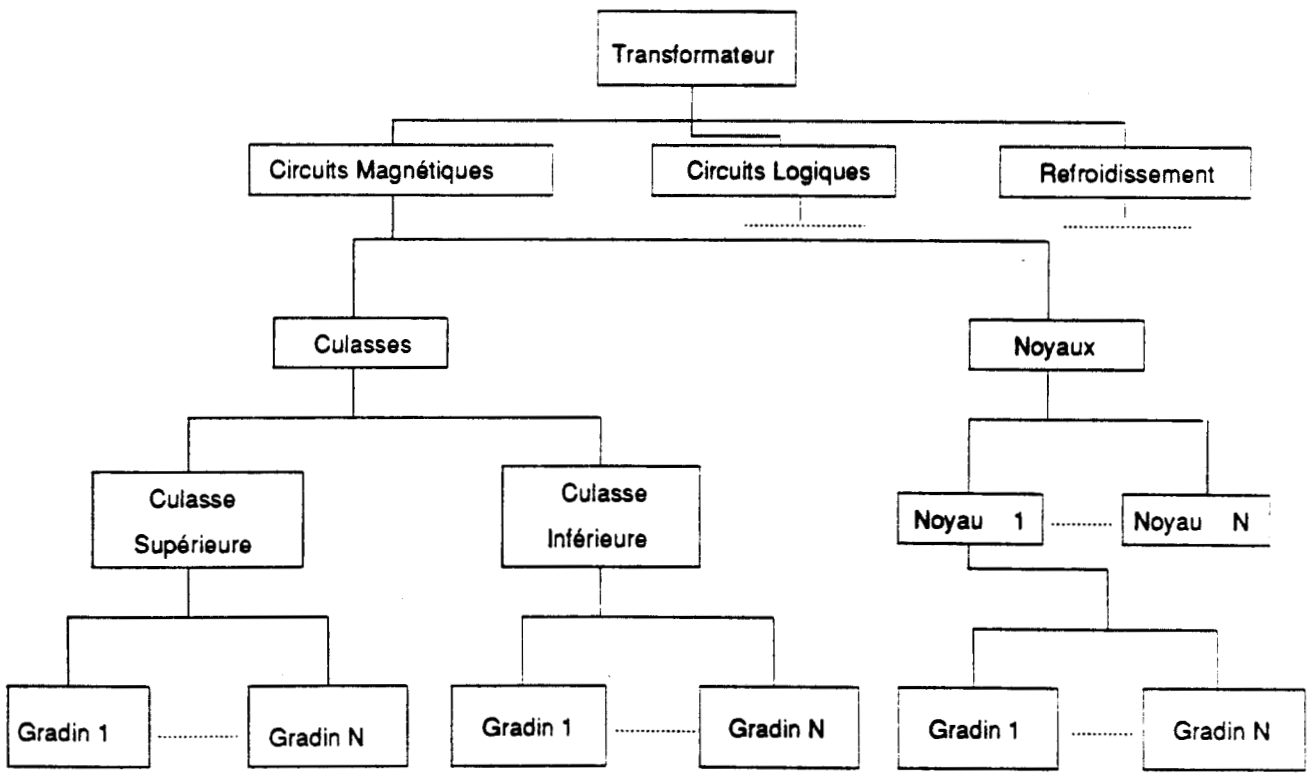
Equations mathématiques (modèle de fonctionnement) :

- $$U_1 = (R_1 + j\ w\ L_1)\ I_1 + j\ w\ M\ I_2$$
- $$U_2 = -(R_2 + j\ w\ L_2)\ I_2 + j\ w\ M\ I_1$$

Description du circuit logique (modèle descriptif et de communication) :



Description arborescente des constituants (modèle de communication) :



I.6. Les objectifs essentiels de la CAO

Nous venons de décrire précédemment les étapes de conception d'un produit industriel, du point de vue méthode et modèle de référence. Voyons maintenant les objectifs des systèmes de CAO et leurs avantages pour la réalisation d'un produit industriel. Les buts des SCAO sont les suivants [David 81, Gardan 86b, Giambasi & al 83, Rueher 80] :

- Augmenter la créativité.
- Améliorer la qualité des produits.
- Réduire les délais et les coûts de conception.
- Vaincre la complexité.
- Faciliter l'archivage et la circulation de l'information.

I.6.1. Augmenter la créativité

La CAO permet d'atteindre cet objectif en autorisant le concepteur à essayer un nombre plus important de solutions sans les créer réellement ni réaliser pour chacune un prototype complet, qui ne serait sans doute pas valide. L'outil informatique permet de modifier, remanier, travailler une solution de façon économique - par visualisation et simulation - et de corriger rapidement et efficacement les erreurs. Tout cela grâce à l'objet virtuel existant dans le SCAO, aux algorithmes et aux outils disponibles, qui permettent à la fois la manipulation de l'objet et la simulation de son comportement.

I.6.2. Améliorer la qualité des produits

Grâce aux outils intégrés dans les SCAO, l'utilisateur peut trouver plus facilement une solution satisfaisante aux problèmes de conception qu'il doit résoudre.

- La vérification du projet par l'examen des réalisations précédentes et par simulation donnent rapidement les meilleures solutions connues par l'entreprise.
- Les programmes de calculs (de résistance des matériaux, de calculs de structures, de placement-routage) permettent de faire des investigations rapides en vue d'obtenir une bonne solution.
- L'obligation de décrire les solutions d'une manière formelle et permettant leur analyse automatique, le respect des normes et standards du domaine, l'intégration de copie et sauvegarde automatique augmentent la qualité et la fiabilité des produits.
- Le stockage des solutions dans des bases de données "projet" permet au concepteur de mémoriser la totalité des caractéristiques d'une solution intermédiaire ou temporaire. Cette capacité peut être étendue à la notion de **version** d'un produit qui décrit l'état d'un projet en cours de développement.

La CAO permet un prototypage aisé et des améliorations progressives par étapes du produit. C'est un atout majeur de ces systèmes.

I.6.3. Réduire les délais et les coûts de conception

C'est un argument important pour qui veut favoriser la généralisation de ce type de logiciel. Une bonne partie de la conception et du développement de produits industriels, dans les approches traditionnelles, est freinée par les tâches routinières nécessaires à la description du produit et à la réalisation des documents pour la fabrication du prototype et son lancement industriel.

La CAO, en réduisant les temps par l'automatisation et la standardisation des documents produits, permet une meilleure coordination entre les services de conception et de fabrication. Ainsi les documents techniques des bureaux d'études et de fabrication, compris et manipulés aisément de tous du fait de leur caractère standard, rendent la communication inter services et la réalisation plus facile. De plus, les modifications du projet, à n'importe quel niveau de réalisation, sont prises en compte immédiatement et simplement.

I.6.4. Vaincre la complexité

La CAO ne permet pas de résoudre tous les cas de conception. Néanmoins, de nombreux produits industriels ne pourraient être conçus et fabriqués si les concepteurs ne disposaient pas de tels outils, en particulier dans les domaines où les objets à concevoir sont complexes - circuits intégrés VLSI, aéronautique, centrales nucléaires - et cette nécessité s'accroîtra sans doute dans l'avenir.

A l'aide de certains procédés qui agissent sur des éléments "virtuels" (n'existant que dans la machine) décrivant le produit, la CAO permet en partie d'améliorer les techniques de conception et de gérer la complexité d'un produit. Citons à titre d'exemple quelques uns de ces procédés :

- Réalisation automatique de parties standards (classiques).
- Visualisation des contraintes apportées par l'environnement extérieur dans lequel doit fonctionner le produit (dans les applications en mécanique, aéronautique, automobile).
- Simulation du fonctionnement de l'appareil sans risque de destruction du projet.
- Contrôle de fonctionnement d'un sous-ensemble du projet.

La richesse des outils intégrés aux SCAO et la décomposition des problèmes en constituants fonctionnels et physiques permettent d'explorer un plus grand nombre de solutions et donc de tenter de résoudre des problèmes de plus en plus complexes.

I.6.5. Faciliter l'archivage et la circulation de l'information

Les quantités impressionnantes de données stockées par les systèmes de CAO facilitent l'archivage. Toutes ces données peuvent être sélectionnées, triées, modifiées, lues individuellement ou suivant certaines caractéristiques choisies par l'utilisateur. Cette souplesse de manipulation des données enrichit les transferts d'informations portant sur le projet et en simplifie la circulation.

Tous les systèmes de CAO modernes disposent de ces outils[Gardan 86b, Quintrand & others 85] . L'archivage informatique des données relatives au produit présente entre autres avantages les suivants :

- Faciliter le transfert des idées dans une même discipline ou entre plusieurs disciplines (le "concepteur" et le "fabricant" parlent le même langage).
- Mémorisation, même primaire, du savoir faire industriel de la société.
- Simplification du transfert et des modifications de données d'une opération à l'autre.
- Simplification de la gestion des documents de toutes sortes concernant le produit.
- Standardisation et guidage de la gestion des projets.

Nous venons de décrire l'utilité des SCAO dans le processus de conception d'un produit industriel. Voyons maintenant quels sont les constituants usuels de ce type de logiciel.

I.7. Structure des systèmes de CAO

Nous nous intéresserons dans cette partie au contenu logiciel des systèmes de CAO classique : fonctions de communication avec l'utilisateur, archivage et gestion des données, traitement et gestion des logiciels internes, logiciels spécialisés et utilitaires [Gardan 86b, Giambasi & al 83].

I.7.1. Fonctions de communication

Le système de communication est l'organe qui permet à l'utilisateur de converser avec la machine et le logiciel. D'où l'importance primordiale de cette partie qui est en fait la seule qui soit partiellement visible de l'extérieur et utilisée directement par l'homme.

L'interaction entre le concepteur et la machine déclenche des opérations concernant l'état et les données relatives à l'objet en cours de conception. Mais cette interaction ne doit pas nuire au raisonnement, à la réflexion du concepteur.

Ceci implique, afin que le logiciel ait de bonnes qualités ergonomiques, que les outils de communications soient à la fois conviviaux (agréables à utiliser) et naturels (d'apprentissage immédiat). Que ces interactions soient réalisées par un langage de commandes, par des menus, ou par des "boutons" (ou touches de fonction) spécialisés, cela importe peu, il faut avant tout que l'utilisateur n'éprouve aucune gêne en utilisant le système.

C'est pourquoi nous étudions cette partie des SCAO en nous concentrant sur son aspect d'utilisation. La réalisation interne dépendant intimement des modèles de l'interaction, de l'application, des structures de données utilisées, et puisque nous ne pouvons prétendre synthétiser la variété des systèmes existants, nous aborderons seulement l'aspect ergonomique de l'interface utilisateur.

I.7.1.1.Langages de communication

Pour réaliser une bonne interaction conversationnelle, le SCAO doit permettre aux concepteurs d'effectuer facilement les tâches suivantes :

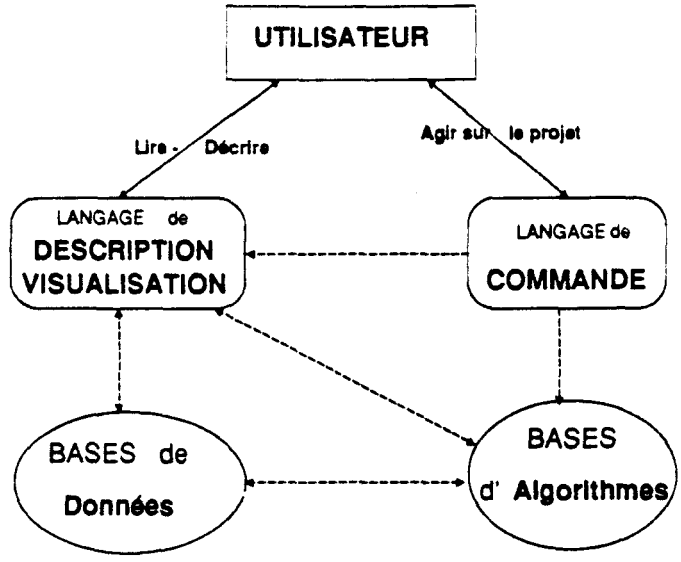
- Elaboration et description des problèmes techniques par une description aisée des informations nécessitées par le projet.
- Lancement de mécanismes partiels ou spécifiques de résolution permettant l'évaluation, la simulation, la modification des maquettes en cours de réalisation.
- Obtention rapide et exploitation simplifiée des résultats.

L'ensemble des logiciels permettant la communication avec la machine est regroupé sous le vocable "**langages de communication**" [Giambasi & al 83].

Lancer une action sur le projet en cours s'apparente à utiliser un langage de commande de machine, tandis que décrire des problèmes techniques ou extraire des résultats nécessite de puiser dans des données. C'est pourquoi on distingue généralement deux types de langages de communication (Cf. Fig I-3) :

- Le langage de description et visualisation qui permet d'échanger des informations entre l'utilisateur et le système; c'est un dialogue avec la ou les bases de données. Il y aura dialogue entre l'homme et la machine pour la description, et monologue de la part du système pour la visualisation, celle-ci étant le résultat visible d'une action de l'utilisateur ou d'un traitement de la machine.

- Le langage de commande que le concepteur utilise pour activer les traitements spécifiques du domaine d'application sur sa maquette.



------ *Liaison interne entre les éléments du SCAO.*
_____ *Liaison directe entre l'utilisateur et le système.*

Fig I-3 *Langages de communication*

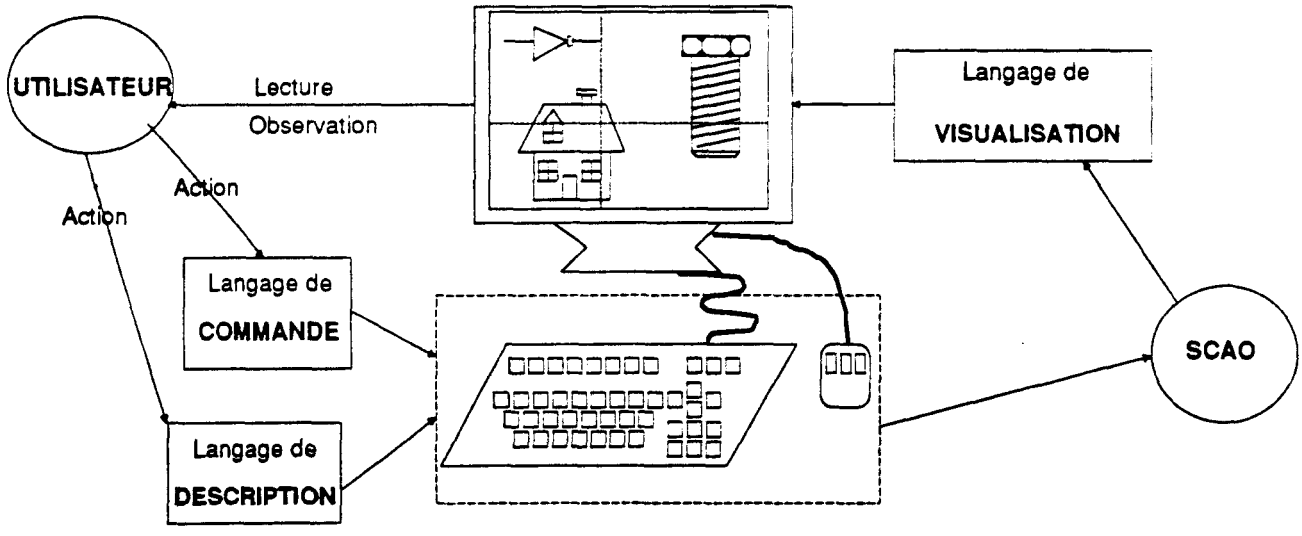


Fig I-4 *Les communications dans un SCAO*

Notons que la partie visualisation des objets et des données est une partie de la communication strictement réservée au système qui lui seul est capable de donner une représentation visuelle, sur terminal graphique ou alphanumérique [Foley & Van Dam 84, Newman & Sproull 81, Pavlidis 82] (en fait toujours un écran pour les parties interactives), de la maquette en cours de réalisation et de ses éléments.

Des principes élémentaires d'ergonomie et de psychologie [Giambasi & al 83, Lebahar 85] sont nécessaires pour disposer d'un bon système de communication.

I.7.1.2.Contraintes de nature ergonomique

Les langages de communication doivent être complets : ils doivent permettent d'exprimer l'ensemble des idées, des actions, des objets relevant du domaine d'application et de l'application concernée, et de décrire (et visualiser en retour) toutes les informations nécessaires. Si ce sont effectivement des langages en tant que tels, leur écriture doit paraître naturelle et leur grammaire aussi simple que possible, sans en restreindre pour autant la puissance d'expression, afin de permettre une adaptation facile de l'utilisateur au système.

De plus, il ne doivent pas nuire à la concentration ou au raisonnement du concepteur qui doit porter ses efforts sur le contenu sémantique des informations transmises ou reçues et non sur leur forme. En outre, les interactions doivent le plus souvent possible se dérouler d'une façon standard; la partie de communication doit être consistante. Il est très important qu'au cours d'une session d'utilisation du système, et pour des opérations ayant des caractères communs -- par exemple un "choix" ou une "destruction" d'éléments -- les appareils physiques et logiques de communication soient utilisés de la même façon.

L'uniformité du mode physique de l'interaction évite une bonne partie des tâtonnements de l'utilisateur qui peuvent engendrer des erreurs de manipulation et, par suite des corrections nécessitées, distraient l'utilisateur de son travail principal. L'interface usager doit être consistante et présenter un aspect et un style d'interaction uniforme pour la totalité des modules du système.

I.7.1.3.Contraintes de nature psychologique

Temps de réaction du système et sentiment d'ennui

Bien souvent, le système doit réagir rapidement à la demande de l'utilisateur (au moins en signalant son activité), sinon des temps de réponse trop longs provoquent ce que nous appellerons "l'ennui", c'est à dire une démotivation pour utiliser la machine.

Dans certains cas extrêmes, il s'agira de "panique", car le concepteur croit avoir provoqué une panne. Il se pose alors les questions suivantes classiques : 'le logiciel ne "répond" plus, il doit être "en panne" ?; ai je perdu tout mon projet ?'

Nous classifions ce type de défaut de la partie de communication, engendrant des nuisances d'utilisation, en trois niveaux : lexical, syntaxique, sémantique.

Au niveau lexical, les opérations sont généralement élémentaires, et les manipulations de l'utilisateur sont assimilables à des "réflexes". Dans ce cas le système doit répondre immédiatement à la demande. Les actions concernées par cet ordre de temps de réponse ont souvent

une faible influence sur l'état du projet (désignation d'attributs graphiques de dessins par exemple, visualisation de parties différentes du projet, etc...) et ne déclenchent pas de longs traitements . Si cela n'est pas respecté, le concepteur aura l'impression de "perdre son temps à attendre la machine".

Au niveau syntaxique, c'est à dire quand l'utilisateur doit formuler de façon syntaxique, grâce aux langages de communication dont il dispose, l'enchaînement de la série d'actions qu'il désire réaliser (macros, icones désignant une suite d'actions standards, ...), le temps de réponse du système peut être plus long, mais doit rester "raisonnable" (de l'ordre de une ou deux secondes).

Enfin, les actions réellement sémantiques, c'est à dire celles qui ont une influence profonde sur l'état du projet, imposent au concepteur un délai de réflexion non négligeable. C'est pourquoi la durée totale d'exécution de l'action par le système peut être plus grande (une dizaine de secondes), car l'utilisateur acceptera facilement, par référence au temps qui lui a été nécessaire pour élaborer mentalement et physiquement sa commande, d'attendre plus longtemps la disponibilité d'interaction du logiciel.

D'une façon générale, on s'aperçoit que le temps de réaction du système pour une action donnée doit être sensiblement du même ordre que le temps nécessaire à l'utilisateur courant pour activer ou affecter les paramètres de la commande concernée. A un degré extrême, si les écarts entre la réaction de l'utilisateur et celle de la machine sont trop grands, le concepteur va éprouver un sentiment de panique qui lui fera rejeter en bloc le système.

Souplesse d'utilisation et frustration

L'utilisateur peut d'autre part se sentir frustré si le système de communication manque de souplesse. Il est exaspéré par l'utilisation incommode de la machine.

Citons, comme exemple trivial, le cas d'une erreur de frappe qui ne peut être corrigée immédiatement et mène immanquablement à l'erreur puis à une redéfinition et réactivation de la commande. Un bon système doit obligatoirement permettre des corrections arrières instantanées sur la dernière commande. Notons que cela n'est pas toujours facile à réaliser, car certaines actions engendrent des modifications globales importantes. Cependant, pour des actions de style réflexe, on devrait toujours disposer d'une telle possibilité.

De même quand une action correcte mais involontaire produit des modifications importantes de données ou de la maquette en cours, l'utilisateur doit pouvoir revenir aisément à l'état précédant cette commande, par une séquence restreinte et si possible formalisée (ne tenant pas de l'astuce d'utilisation).

Autre exemple : dans le cas d'une interaction se faisant au moyen de menus hiérarchiques, il est important que l'utilisateur puisse changer très rapidement le "niveau" du menu qu'il utilise, le système revenant alors à l'environnement d'exploitation précédent, et que ce niveau soit lisible rapidement (on peut choisir par exemple d'afficher les niveaux actifs avec une couleur spécifique) et sans ambiguïté (il faut que les noms de commande ou les motifs d'icônes choisis pour représenter la commande soient explicites).

Facilité de compréhension et confusion

Parfois, le nombre d'informations à transmettre à l'utilisateur du système est très grand. Dans ce cas, si le système de communication livre l'information dans sa totalité, le concepteur ne sera plus capable d'appréhender celle-ci. Il faut donc dans ces cas là morceler, découper le paquet d'informations, et ne présenter que l'information intéressante à l'utilisateur.

Si son contenu renseigne sur des caractéristiques d'objets de la base de données, on aura intérêt à présenter cette information sous forme de descriptions spécialisées (dessins avec cotations, page alphanumérique de valeurs, ...).

Si le contenu est un dessin complexe, l'utilisateur doit pouvoir en extraire aisément certaines parties constitutives (ex : pour le routage de circuits imprimés les parties visibles une à une, pour un dessin d'architecture les différents éléments) ou visualiser un sous-ensemble seulement (zooming).

Enfin, l'emploi de menus dit "déroulants" et hiérarchisés est hautement souhaitable, à condition qu'ils évitent les sentiments de frustration précités, c'est à dire qu'ils soient d'une bonne souplesse d'utilisation.

Uniformité physique de l'interaction et manque de confort

Lors du dialogue avec la machine, le concepteur se sert d'appareils d'entrée très spécialisés : souris ou tablettes pour désigner des éléments graphiques ou des points à l'écran, pour choisir les options de menus, clavier pour préciser des données alphanumériques.

Il est important que l'interaction s'effectue au moyen d'appareils d'entrée physique adaptés à l'opération interactive en cours. De plus, l'utilisateur du système ne doit pas être gêné ou "perdu" par une profusion de matériels souvent redondants. Il est bon qu'une station de CAO - c'est à dire l'ensemble des matériels mis à disposition du concepteur - comporte au maximum deux terminaux de visualisation graphique interactive (écrans), deux moyens physiques d'entrée graphique (une souris et une table à digitaliser), un clavier alphanumérique, un traceur (ou imprimante laser).

La station CAO "standard" est généralement constituée des éléments suivants :

- Un écran graphique.
- Une souris ou tablette à digitaliser.
- Un clavier.
- Un traceur.

Nous avons insisté sur la partie communication des systèmes de CAO car elle nous a semblé essentielle pour réaliser de "bons" systèmes de CAO. En effet, l'utilisateur doit travailler de façon interactive avec la machine, il est donc important qu'il n'éprouve pas de difficulté particulière à se servir du système, que la "conversation" soit aussi naturelle que possible, et que les moyens d'interaction autant logiciels que matériels soient les plus adaptés.

I.7.2. Archivage et gestion des données

Les données à manipuler lors du processus de conception sont très nombreuses. Il s'agit de données numériques, dessins, documents explicatifs, documents d'archives, projets et maquettes en cours, etc...

La partie du système de CAO qui permet d'archiver et de gérer l'ensemble des données utilisées est aussi importante que celle décrite précédemment, même si ses fonctionnalités ne sont pas visibles directement à l'utilisateur. Nous n'aborderons pas ici les questions techniques de gestion des données en CAO, mais nous décrivons leur état en cours d'utilisation, à savoir leur aspect statique ou dynamique.

Données statiques

Ce sont des données que le concepteur consulte et qui au cours de la session de travail avec la machine ne seront jamais modifiées : bibliothèques de composants électroniques avec leurs caractéristiques physiques et électriques, normes et standards à respecter pour la conception de l'objet en cours, archives et documents relatifs aux réalisations passées... Elles constituent en quelque sorte le "support bibliographique" nécessaire au concepteur.

Données dynamiques

Ce sont des données qui sont en perpétuelle évolution quand on utilise le SCAO. Elles portent essentiellement sur l'objet qui est en cours de conception (les dernières modifications apportées, le contexte d'utilisation du système, etc).

Le volume important des données à manipuler en CAO (textes, caractéristiques, documents) implique une structuration très forte de celles-ci et nécessite des modèles de gestion des données puissants et adaptés. La partie logicielle assurant cette structuration devra donc satisfaire à des exigences particulières, en particulier celles citées ci-après :

- Stocker et gérer l'ensemble des informations relatives à l'objet en cours et au domaine d'application. Cela se fera grâce à la ou aux bases de données diverses concernant le projet, le domaine d'application, les objets manipulés.
- Permettre aux membres de l'équipe de conception d'accéder à certaines informations, de façon hiérarchique (le responsable de projet accédant à toutes les données et ses subordonnées ne "voyant" que la partie les concernant), ou de façon locale à la phase de conception en cours (il est néfaste de modifier le schéma électronique d'un circuit lors du placement des composants).
- Autoriser des accès faciles aux données en utilisant des critères d'accès ou de choix sélectifs. On fait référence ici à tout ce qui concerne les modèles d'interrogation de bases de données.
- Mettre à jour régulièrement les informations de l'objet en cours de conception et en particulier réorganiser périodiquement l'ensemble des informations le concernant (suppression d'entrées redondantes ou inutiles, réorganisation de l'image virtuelle du projet, mémorisation régulière et automatique de l'état du projet, ...).
- Permettre aux parties logicielles spécialisées d'accéder dynamiquement aux informations dont elles ont besoin. Encore un aspect spécifique des langages d'interrogation de base de données et à leur utilisation à l'intérieur des programmes.
- Permettre une sortie externe de documents, de dessins, de nomenclature, de liste des composants facilement et à n'importe quel moment de la session de travail.

On admet en général l'existence de deux types de bases de données spécialisées pour un SCAO :

- La(es) base(s) de données projet contenant les informations d'un projet en cours de conception, ou réalisé antérieurement.
- La(es) base(s) de données techniques contenant des informations spécialisées du domaine d'application.

En bref, les fonctions d'archivage et de gestion des données doivent, pour un SCAO donné, être adaptées aux types d'applications de conception prévues et aux demandes éventuelles des utilisateurs de ces systèmes. Il semble aujourd'hui que la technologie des bases de données soit suffisamment avancée pour répondre à ces exigences [Delobel & Adiba 82] et que les travaux sur les bases de données spécialisées en CAO se généralisent [Cholvy & Foisseau 85, Fauvet & Rieu 87, Valette & Foisseau 85].

I.7.3. Traitement et gestion des logiciels internes

L'algorithmique intervient de façon prépondérante dans les SCAO : programmes de calcul, de simulation, de vérification et de contrôle, logiciels graphiques de tracés, de visualisation, gestion des interactions spéciales (graphiques, analyse syntaxique des langages de commande); ce sont autant de logiciels spécialisés fonctionnant grâce à des algorithmes spécifiques [Giambasi & al 83, Lebahar 85, Quintrand 85].

Il faut s'assurer que la communication des informations entre ceux-ci s'effectue correctement et automatiquement. De plus, toutes ces fonctions logicielles utilisent très souvent un nombre important de données communes. C'est pourquoi les parties communes, pour éviter des redondances logicielles, sont souvent regroupées . Elles sont alors accessibles soit par des programmes spécifiques à leur utilisation, soit partagées entre plusieurs parties logicielles les utilisant.

Enfin, l'interaction et la souplesse d'utilisation que doit satisfaire le système de CAO implique des activations et des désactivations nombreuses des logiciels internes.

C'est pourquoi l'architecture du logiciel est très complexe et ne peut plus s'exprimer dans les termes classiques de programmation : les arbres de description et les méthodes de programmation structurée sont insuffisants pour décrire leur structure. On aura intérêt à décrire le fonctionnement logiciel avec des graphes comportant des noeuds logiciels spécifiques à une fonctionnalité et munis de sous programmes "satellites" (Cf Fig I-5).

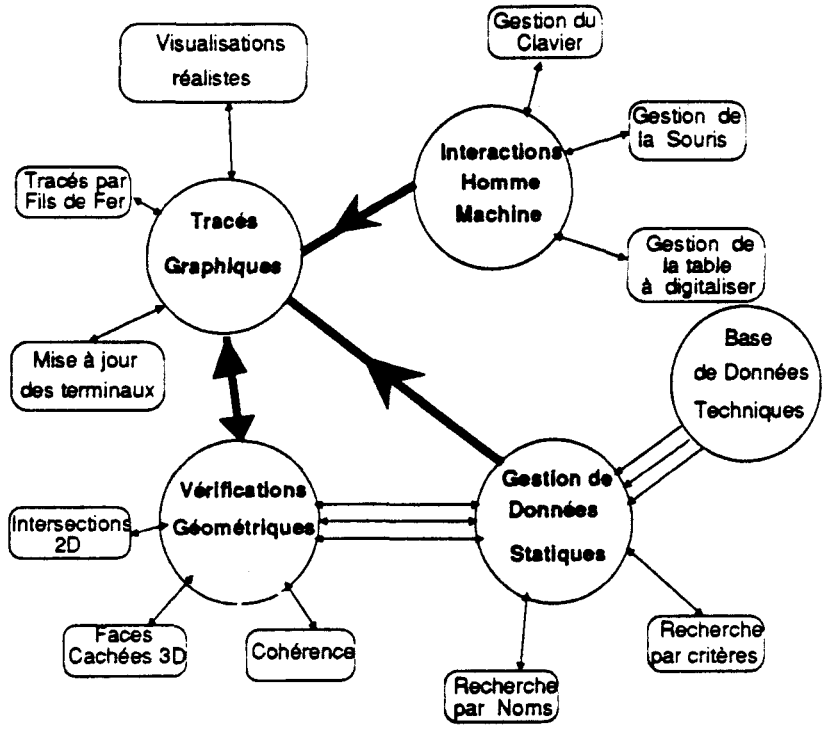


Fig I-5 *Décomposition en logiciels internes*

On voit sur le schéma précédent des noeuds de traitement spécialisé dans certaines tâches.

C'est le cas de la gestion élémentaire des matériels de communication (clavier, souris, tablette à digitaliser) : chacune de ces parties nécessite une portion de logiciel adaptée au matériel, mais cependant fait partie du sous-ensemble général qui décrit l'interaction graphique et qui permet de communiquer avec la machine.

De même, le noeud des tracés graphiques permet des fonctions générales comme l'affectation des attributs visuels des tracés, tandis que des sous programmes satellites se chargent des parties précises comme la remise à jour de l'image sur les écrans de visualisation, le traitement dédié au tracé de volume, la modélisation et l'affichage des objets en 3 dimensions[Gardan 85, Gardan & Lucas 83]...

Notons aussi que l'organisation interne est fortement dépendante des structures de données utilisées et des moyens de communication envisagés avec l'utilisateur.

Enfin, certaines commandes du système déclencheront des activations de sous-programmes internes en chaîne. Par exemple, lors de l'ajout d'un nouveau composant, l'utilisateur choisit sa commande, l'active et dispose d'un nouveau composant qu'il peut manipuler à volonté. Dans ce cas, il y a activation des modules spécialisés dans l'interaction homme-machine, de la partie de gestion des données statiques, de l'extracteur d'éléments dans la base de données technique, puis copie de l'élément référence dans la base de don-

nées projet, et visualisation graphique de l'élément en vue d'une manipulation ultérieure. Il est donc nécessaire, avant de concevoir le système de CAO, de connaître et d'étudier en détail les besoins des usagers pour choisir un modèle du système interactif de communications, et de bien planifier l'enchaînement des sous-programmes relatifs au déclenchement d'une commande.

Pour atteindre, ces objectifs, l'application de techniques modernes de programmation modulaire, de prototypages de logiciels, et des méthodologies "orientées objets" sera un atout majeur pour la réalisation de ces systèmes. La bibliographie en CAO atteste d'ailleurs de ces orientations prises par les concepteurs des nouveaux systèmes [Cholvy & Foisseau 83a, Chouraqui & Dugerdil 86, Dugerdil 85, Flemming & Coyne 90, Gardan 86b, Gardan & Zachari 91, Hanser 90, Haurat & al 90, Houbart 90, Lebahar 84, Marrin 84, Montalban 88, Petit 90, Smadja 86, Trousse 89 90a].

I.7.4. Logiciels généraux et utilitaires

Les systèmes les plus complets de CAO regroupent quantité de logiciels internes. Chacun d'entre eux est conçu et construit pour une tâche ou un ensemble de tâches spécifiques. Nous abordons ici, par catégorie de fonctionnalité, les logiciels les plus représentatifs systèmes :

- Langage de commande
- Menus
- Langages graphiques
- Langages de description
- Algorithmes spécialisés du domaine
- Base de données
- Liens avec les SCAO externes

Langages de commande

Les langages de commande permettent à l'utilisateur d'exprimer une action "valide" pour le système de CAO. Ils peuvent être de simples langages à format fixe [Gardan 86a 86b, Gardan & Lucas 83, Giambasi & al 83] ou de véritables langages modélisables par un automate d'états finis [Lebahar 85 Ch5]. Par exemple, pour un système de CAO en mécanique, faire une translation d'un objet élémentaire de type "*axe*" vers le haut se décrirait :

Translater Axe A1 vers Nord 10 pas

Remarquons que les commandes que l'on peut décrire avec ce genre de langage doivent couvrir la totalité des commandes du système. Les "phrases" de commande utilisent alors une grammaire simple et standard pour toutes les commandes [Lebahar 85 Ch2].

Il existe alors un interpréteur pour ces commandes, qui intègre aussi la communication nécessaire avec d'autres logiciels spécifiques (bases de données, algorithmes de calcul ou simulation, etc).

Jusqu'à présent ces logiciels de CAO dits de deuxième et troisième génération décrivaient des langages de commande assez simples, mais on constate une évolution de ceux-ci pour une manipulation plus naturelle et plus riche [Beech & al 86, Gardan 86a Ch4.3].

Les menus

Parmi les outils de communication, les systèmes à menus jouent un rôle important. Un menu se présente sous forme d'une liste de choix de commandes présentées de manière alphanumérique (nom de commande) [Berthelot 85, Gardan 86b] ou graphique (icone symbolisant l'action à réaliser). Ils peuvent être visibles de façon permanente sur le matériel (tablette, écran graphique), ou seulement au moment où ils sont utilisables (menus déroulants, structure arborescente des commandes), ou encore à l'appel d'une commande particulière. Ces outils sont le résultat des travaux de développement concernant la partie interface usager. Les langages de commandes sont vite apparus comme des utilitaires trop lourds pour réaliser des opérations standards et fréquentes. Les systèmes à menus offrent une bonne solution au problème de ces tâches répétitives et fastidieuses du point de vue de l'interaction.

Ils constituent un moyen efficace pour la communication, facile d'apprentissage et rapidement accessible. Cependant, des menus trop chargés provoqueront un sentiment de confusion [Giambasi & al 83, Quintrand & others 85]. Il est essentiel que les commandes ou actions qu'ils regroupent soient de nature identique de façon à ne pas rendre confuse l'utilisation du logiciel.

Langagesgraphiques

Ils font partie des langages de visualisation utilisés par l'ordinateur pour réaliser les tracés nécessaires à la visualisation de l'état du projet [Genoud & Grabowiecki 86].

Ces logiciels doivent être capables de décrire de manière symbolique les objets graphiques représentés et de provoquer le dessin d'un élément quand cela est nécessaire [Newman & Sproull 81]. Tout ceci doit se passer de façon automatique. Ces "langages", qui ne sont parfois que des bibliothèques de programmes graphiques, permettent les tracés élémentaires (vecteurs, arc de cercle) et de beaucoup plus haut niveau (modélisation de surfaces ou de volumes, description d'un objet en sous-éléments, segmentation graphique...). Ils peuvent être indépendants du matériel (GKS), ce qui permet d'améliorer la portabilité (concepts de station graphique) et d'accroître la puissance d'expression (segmentation GKS et structures PHIGS) [PHIGS 88] du système de visualisation; ou bien ils seront plus liés au matériel, ce qui permet de disposer d'outils adaptés à un matériel particulier et du coup plus efficace pour une sortie graphiques (VDI, CGI), [GKS 85, CGI 88].

Pour un niveau d'adéquation optimal aux projets de CAO, où la complexité des projets nécessite à la fois richesse et puissance d'expression, des standards de très haut niveau sont nécessaires (PHIGS), en particulier en ce qui concerne la description symbolique en mémoire de l'image représentant l'état du projet ou de ses parties (structures PHIGS) [Ducrot 86, Krzewina 86].

Langages de description

Ils permettent à l'utilisateur de décrire le modèle des objets en cours de conception. Ils peuvent se trouver sous les formes suivantes : documents alphanumérique (liste de composants, description de circuits), ou graphiques. Bien qu'ils restent à un stade plus ou moins expérimental, les systèmes de reconnaissance vocale peuvent être envisagés.

Algorithmes spécialisés du domaine

Ce sont des logiciels appliqués au domaine visé par le SCAO (placement, routage de circuits imprimés, calculs de structures...). Ils constituent le noyau de la partie "experte" du système. Grâce à eux, l'utilisateur peut vérifier la validité de sa création, et peut tester à volonté son modèle [Masseboeuf 86] (de façon plus ou moins commode ...). Un bon SCAO devrait permettre, à n'importe quel état d'avancement du projet, une conception manuelle pour les cas non répertoriés ou non accessibles par les algorithmes spécialisés, et une conception automatique pour des cas classiques ou simples.

Par conception manuelle, nous entendons que le concepteur réalise le projet "à la main", seulement aidé par les outils élémentaires du système (éditeurs de schémas, contrôle de validité simple, ...). Au contraire, la conception automatique concerne l'entière réalisation par la machine d'une partie essentielle du projet (routage des pistes en CAO électronique, calculs et réalisation automatique de structure en mécanique et en architecture, ...). Notons qu'en général ces logiciels, s'ils sont capables de résoudre une bonne partie des problèmes relevant de leurs spécialité sont très souvent incapables de résoudre de façon complète le problème posé [Carre 85b, Quintrand & others 85]. Finalement, l'approche la plus riche sera celle qui retient l'approche manuelle et automatique en même temps [Genoud & Grabowiecki 86, Smadja 86].

Bases de données

Les logiciels spécialisés pour les bases de données (accès, gestion) jouent un rôle essentiel. Le système de gestion de bases de données pour la CAO (S.G.B.D. CAO), comme tout système informatique ayant une forte interaction homme-machine, doit permettre à l'utilisateur et aux programmes spécialisés du SCAO d'interagir (en mémorisant, manipulant, extrayant, détruisant... ces données) avec la(les) base(s) des données existantes [Giambasi & al 83, Delobel & Adiba 84, Cholvy & Foisseau 83b].

Les structures que doivent gérer ces systèmes sont dynamiques [Gardan 86ba], car l'objet à concevoir est en perpétuelle évolution. Le système devra permettre d'informer aussi bien sur les données que sur la façon dont elles sont structurées, car l'objet initial est vide et croît en

structures et fonctions de façon dynamique; d'où la nécessité (en particulier pour les programmes internes) d'accéder facilement aux structures, au moins pour obtenir leur description.

Jusqu'à présent, peu de systèmes de gestion de bases de données étaient adaptés à la CAO [Cholvy & Foisseau 85, Valette & Foisseau 83]. Actuellement, les voies d'investigation pour la CAO vont vers une évolution de systèmes relationnels [Cholvy & Foisseau 85, Valduriez 87], et les systèmes orientés objets [Abiteboul & Grumbach 87, Gardarin 88, Parent & Spaccapietra 88].

Les spécialistes des SCAO classent les bases de données CAO en trois types (un de plus que le nombre de catégories citées au I.7.2), car ils séparent celles qui concerne les projets en deux types. Voici donc ces trois types [Gardan 86b, Giambasi & al 83] :

- La(les) base(s) de données techniques qui regroupe les informations techniques concernant les objets qui font partie du domaine d'application du système. Elles décrivent des données de nature essentiellement statique concernant le domaine d'application du SCAO (nomenclatures standards, composants élémentaires, ...).
- La base de données du projet en cours qui contient toutes les informations (structurelles, fonctionnelles, graphiques...) de l'objet en cours de conception. Les données sont dans ce cas de nature dynamique (l'objet est en perpétuelle évolution).
- La base de données pour les projets antérieurs qui contient les représentations (modèles) d'objets conçus auparavant. Son contenu est de nature statique car on suppose que les projets antérieurs sont figés (ils représentent en quelque sorte le "savoir-faire" de la société).

Nous n'abordons pas ici la structure des modèles existant [Delobel & Adiba 84], rappelons simplement que des trois modèles les plus communs (hiérarchique, réseau, relationnel), le dernier modèle semble le plus adapté aux caractères dynamiques et très imbriqués (beaucoup de liens entre aspects fonctionnels et structurels) des problèmes, bien que souffrant de défauts non négligeables . Ce modèle est aujourd'hui beaucoup mieux défini que les nouveaux modèles déductifs ou orientés objets, qui eux sont bien adaptés aux problèmes rencontrés en CAO, et qui font l'objet d'intenses recherches [Abiteboul & Grumbach 87, Cholvy & Foisseau 85, Gardarin 88, Parent & Spaccapietra 88, Rieu 86].

Liens avec les systèmes de CAO externes

En étroite relation avec les systèmes de gestion de base de données spécialisés, il existe une partie logicielle des SCAO, qui joue un rôle de plus en plus important : ce sont les logiciels de transfert et de communication de données sur un projet, entre systèmes de CAO.

Au fur et à mesure de la conception et de la réalisation de nouveaux SCAO, qui parfois sont devenus des standards de fait, et en vue d'améliorer la description informatique du projet, ou de trouver des descriptions satisfaisantes à la fois pour les techniques de

programmation algorithmique et les techniques logicielles spécifiques de l'intelligence artificielle, les réalisateurs de SCAO fournissent, aux utilisateurs de leur système, des passerelles vers d'autres SCAO concurrents. Ceci est, entre autres, une manière de leur dire : "vous pouvez adopter en toute confiance notre logiciel, puisque vous pouvez récupérer votre précédent travail".

Une autre raison de l'existence de ces passerelles, est l'apparition de SCAO fonctionnant en réseaux, c'est à dire où les utilisateurs-concepteurs font un travail local et spécialisé sur le projet, ce qui nécessite que les données, manipulées par des parties logicielles spécialisées dans des domaines indépendants, puissent être traduites d'une partie à l'autre. Dans ce cas, il s'agira d'intégrer des données étrangères par un logiciel spécialisé.

Enfin, pour uniformiser et industrialiser la description des projets et des bases de données techniques, certains standards, par exemple EDIF (EDIF), tentent de décrire formellement toutes les données nécessaires au SCAO : techniques, graphiques, fabrication ... En fait, on désire que tous les SCAO "parlent le même langage de description", quelle que soit leur provenance.

C'est pourquoi cette liaison inter-SCAO est importante. Il s'agit de formats de description et de spécification des données techniques du domaine. Il existe alors des interpréteurs, ou des traducteurs, en entrée et en sortie de chaque SCAO, qui transforment les données du SCAO dans le format de description prévu. Il s'agit de normes (IGES, EDIF), ou de standards de fait (DXF, SET). Plus les projets sont complexes et volumineux, plus il est important de disposer de telles passerelles.

I.8. Schéma conceptuel adopté dans ce document

Tout au long du texte, pour aborder des parties spécifiques aux systèmes de CAO, nous ferons référence au schéma suivant, celui-ci étant une description symbolique des relations entre les logiciels du système.

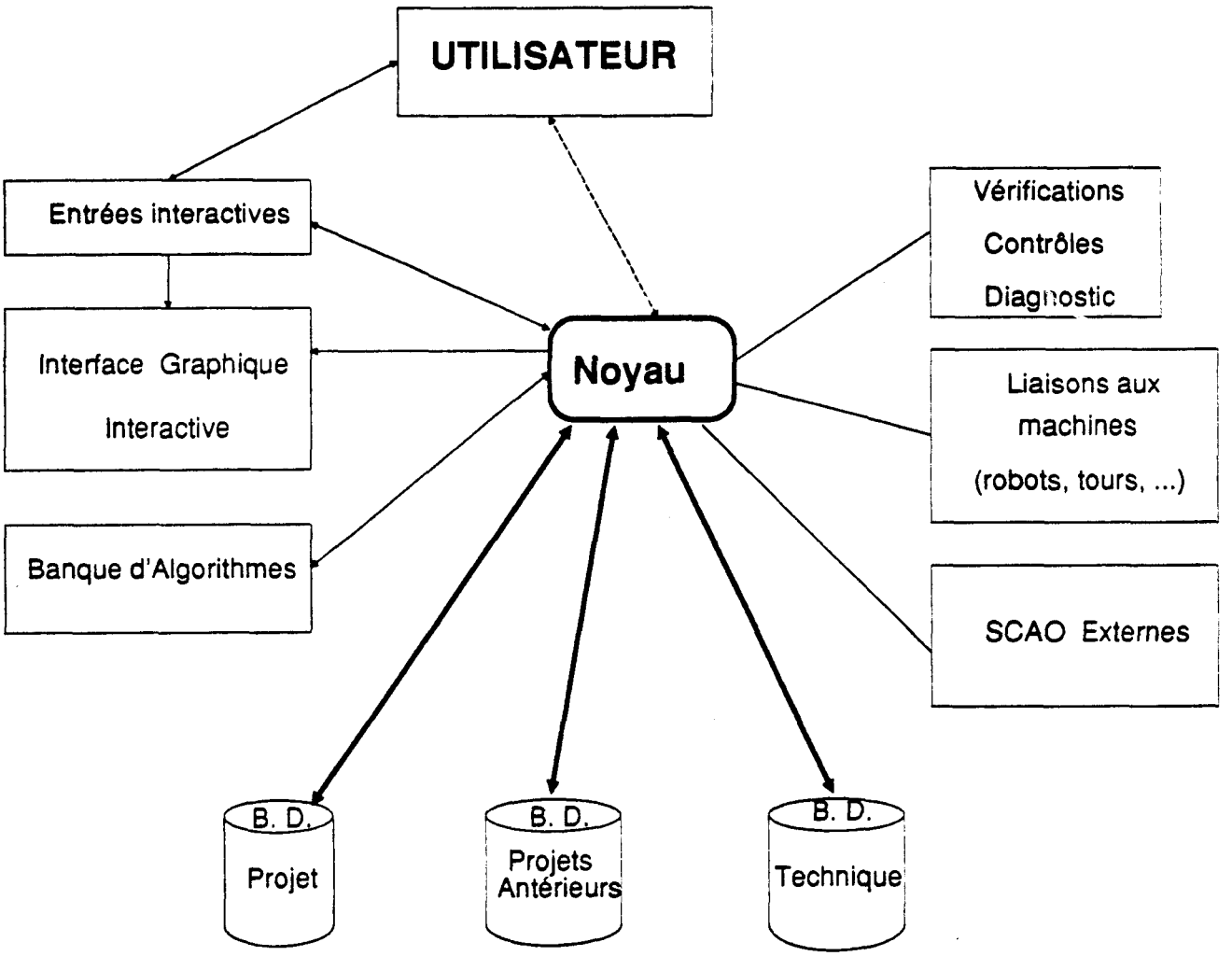


Fig I-6 *Modèle conceptuel de SCAO adopté*

Commentaires sur ce schéma :

La partie **"utilisateur"** est en fait le concepteur qui utilise l'ensemble matériel et logiciel "système CAO".

La partie **"entrées interactives"** constitue l'ensemble des outils de communication directement accessibles à l'utilisateur. Elle comprend : l'interpréteur des commandes, la gestion des menus, le langage de description. Elle est liée à la fois à la partie "interface graphique *in-*

teractive" pour détecter dans les menus l'action demandée, et au noyau pour l'interprétation des commandes directes ou pour réaliser les liens nécessaires avec les autres parties du système.

L'**interface graphique interactive** est l'ensemble des logiciels permettant la visualisation des objets, menus, icônes ..., tout ce qui demande à être visualisé sur un écran graphique, ainsi que la gestion des liens entre les appareils physiques d'entrée (tablette, clavier) et la partie entrées interactives. Le schéma habituel d'une interaction avec une entrée graphique est le suivant :

- a) Manipulation de l'appareil physique pour désigner un point, une case de menu, un *objet graphique élémentaire à l'écran*...
- b) Détermination d'une valeur associée à cette saisie (coordonnés, valeur numérique, choix, objet...)
- c) Détermination d'un lien entre cette valeur et une action, commande ou autre particularité du système de CAO, avec le paramétrage de cette commande.
- d) Activation de l'action correspondante par le noyau.

Banque d'algorithmes : cette partie concerne tous les logiciels spécialisés du système, visant à résoudre directement un problème : algorithmes de placement-routage, de simulation électrique, de calcul de structure en mécanique, d'intersection de surfaces ou de volumes, de simulations de fonctionnement... En bref, ce sont des algorithmes liés au domaine d'application visé, ou aux parties spécifiques mais classiques des SCAO, comme la modélisation d'objets en trois dimensions par exemple.

Les trois types de **base de données** ont déjà été précisés auparavant.

La partie **vérification, contrôle, diagnostic**, concerne les aides apportées à l'utilisateur pour évaluer sa réalisation (détecter les incohérences, dysfonctionnements prévisibles...).

La partie **"systèmes de CAO externes"** concerne la partie communication avec d'autres systèmes de CAO, en particulier l'interfaçage, c'est à dire l'adaptation des résultats d'un système pour l'utilisation dans un autre système [Claude 86, Smith 86].

Ce schéma ne décrit que les aspects généraux des fonctionnalités du logiciel. Nous le compléterons dans le chapitre III pour y indiquer les zones où nous avons étudié spécifiquement l'implantation de parties logicielles usant de techniques d'intelligence artificielle.

I.9. Quelques problèmes des systèmes de CAO

Ce chapitre introductif à ce qu'est la CAO en général ne met pas en relief les problèmes actuels ou les défauts qui commencent aujourd'hui à sembler de plus en plus rédhibitoires pour pouvoir réellement aider le concepteur dans sa tâche.

En effet, les choix et les modèles initiaux de développement qui ont été motivés par certains besoins élémentaires sont maintenant bien intégrés aux systèmes de CAO. Du fait même que la demande initiale est satisfaite par ces logiciels, les modèles montrent aujourd'hui leurs faiblesses et leurs limites. De nombreux travaux ont été réalisés pour analyser leurs déficiences et tenter d'apporter des solutions [Favard 89, Trousse 89 90a 90b, Latombe 85, Orchampt 87]. Néanmoins, certains manques demeurent et ne peuvent plus être négligés.

Nous rappellerons plus en détail ces problèmes dans les chapitres suivants, mais nous pouvons déjà citer quelques lacunes importantes auxquelles il faudrait aujourd'hui remédier :

- Evolution des SCAO : initialement ces systèmes avaient pour but de mettre à profit la puissance de calcul des ordinateurs, pour faciliter des travaux routiniers simples dans leurs concepts, mais fastidieux ou complexes dans leur réalisation (calcul de structure, de résistance de matériaux, simulation de fonctionnement, ...). On pouvait alors les considérer un peu à la manière d'une calculette électronique qui fournit un outil pratique de calcul. De ce fait, le développement naturel des recherches attachées aux SCAO s'est porté essentiellement sur l'amélioration des fonctions de calcul ou de modélisation numérique. Les outils actuels sont riches et puissants en ce domaine, mais montrent actuellement leurs limites.
- Outils interactifs et graphiques : un axe important de développement a été la mise à disposition d'outils interactifs graphiques adaptés aux problèmes de représentation visuelle rencontrés en CAO (modélisation en trois dimensions, interface usager basée complètement sur le graphisme, ...). Malheureusement ces outils ne fournissent qu'une aide à la visualisation et à la représentation virtuelle des concepts et entités manipulés. Ils ne fournissent aucune aide quant à la sémantique des objets et à l'interprétation de résultats [Houbart 90]. Il faudrait maintenant fournir une aide plus active au concepteur; une aide qui insiste sur le caractère dynamique et incertain de l'activité de conception.
- Le stockage des données a donné lieu à beaucoup de travaux pour améliorer les modèles de description des données utilisées, mais encore une fois il s'agissait plus de disposer de systèmes de gestion de base de données efficace que de fournir une représentation adaptée aux problèmes de la CAO. En général, peu de sémantique concernant le domaine d'application n'est incorporé de manière préméditée dans les SGBDexistant.

- Faiblesses spécifiques des modèles utilisés. Vu les aspects géométriques importants et la complexité des entités à utiliser, les modèles proposés aujourd'hui fournissent de quoi simuler géométriquement ou structurellement les objets manipulés dans les SCAO. Ils n'incorporent cependant pas les aspects fonctionnels des entités ainsi que les autres facettes sémantiques qui leur sont attachées [Houbart 90, Trousse 90a].

- Faiblesses des solutions algorithmiques au niveau global : les algorithmes spécifiques fournis dans les SCAO permettent de résoudre des problèmes locaux particuliers et ils le font souvent assez bien. Cependant, la complexité inhérente de la tâche de conception ne peut être totalement maîtrisée par un ensemble d'algorithmes [Favard 89]. De nouvelles solutions doivent être apportées pour disposer d'outils permettant de travailler aisément dans toutes les activités de conception.

- L'entrée des données reste une tâche laborieuse quand il s'agit de décrire des objets tridimensionnels ou complexes. C'est une phase coûteuse en temps et propice aux erreurs, car elle nécessite de nombreuses interactions avec l'utilisateur [Trousse 89].

- Modélisation des objets manipulés : les divers aspects géométriques, fonctionnels, ou sémantiques (qui peuvent concerner eux-même des caractères géométriques ou fonctionnels) ne sont actuellement pris en compte que partiellement (cas des pièces géométriques de structure complexe), ou pas du tout (aspects et concepts sémantiques des objets). Ce problème existe pour chacun des aspects pris séparément ou ensembles. Il faudrait d'une part améliorer la capacité des SCAO en ce domaine et d'autre part intégrer ces divers aspects dans une structure commune [Carre 89, Dugerdil 85, Trousse 89].

- Gestion de la cohérence : la cohérence des objets conçus est un de leur propriété à vérifier en permanence pour détecter immédiatement certaines erreurs de conception. Or les logiciels de CAO actuels ne savent souvent en tenir compte que d'un point de vue statique. Par exemple, un utilisateur doit lancer des programmes globaux de test de cohérence après avoir complété toutes les modifications d'un objet. Ces tests devraient être fait dynamiquement et en permanence, sans que l'utilisateur n'ait besoin de les activer.

- Richesse de la méthodologie de conception. En général les outils actuels supportent une méthode de conception et d'assemblage ascendante. De plus, les entités ne peuvent être manipulées et traitées par le système que si elles sont décrites complètement. Il faudrait proposer également une méthodologie de conception descendante (Cf I.4) qui soit capable de prendre en compte des informations incomplètes.

- Les outils d'évaluation restent très pauvres et ne sont en général que partiellement réalisés à l'aide d'algorithmes spécialisés (de simulation par exemple) dont l'utilisateur doit interpréter les résultats sans aucune aide.

- La gestion évoluée des versions de maquettes d'un produit est très rare [Trousse 89 90a]. En général les logiciels de CAO ne peuvent traiter et mémoriser (grâce à la base de données projets) que des projets réalisés complètement antérieurement. Il existe

très peu de SCAO capables de gérer correctement les évolutions successives d'un objet, ce qui permet réellement de travailler par "*essais-erreurs*" et d'explorer de façon pratique diverses solutions possibles.

Les recherches actuelles en CAO visent essentiellement à offrir une modélisation plus riche et performante des entités CAO manipulées et à trouver des remèdes aux manques des solutions algorithmiques en intégrant des techniques d'intelligence artificielle adaptées aux problèmes de la CAO :

- prise en compte des aspects sémantiques des objets [Gardan & Zachari 91, Houbart 90, Trousse 89].
- intégration de modèles géométriques "*sémantiques*" et d'expertise dans un SCAO [Petit 90, Trousse 89].
- intégration du concept d'objet en tant que tel dans les logiciels de CAO [Mohan & Keyshap 88, Petit 90, Trousse 88].
- représentation d'objets à base de *frames* pour introduire à priori de bons modèles de représentation des connaissances en vue d'intégrer des solutions de type I.A [El Dashman & Barthes 90, Dugerdil 85, Trousse 89, Vogel 85].
- structure d'accueil plus riche pour mieux prendre en compte la diversité des activités de conception et offrir de meilleurs services à l'utilisateur [Trousse 89].
- prise en compte d'aspects concernant la sémantique géométrique des objets [Orchampt 87, Trousse 89].
- divers systèmes experts d'aide à la conception [Dejesus & Callan 85, Druais 87, Jabri & Skellern 89, Jonckers 85, Joseph 85, Keen & Seviora 87, Odawara & al 90, Rychener 85, Sriram 85, Thoraval & al 90, Trousse 87, Tsuchida & al 89, Voirol & Piguet 89, Yoshimura & al 90].
- systèmes de résolution de problèmes à base de contraintes [Duverdier & Tsang 91, Hanser 90, Makino & Ishizuka 90, Verroust 90].
- raisonnement spatial [Andre 86, Retz Schmidt 88, Walker & al 88].
- coopération entre outils IA et SCAO [Gardan & Zachari 91, Neveu & al 90, Trousse 89].

Le lecteur trouvera une bonne documentation, ainsi qu'une bibliographie assez complète en ces domaines dans [Actes de MICAD 85 à 91, Gardan & Zachari 91, Favard 89, Orchampt 87, Trousse 89]. Notons en particulier que les systèmes de CAO sont un terrain privilégié pour l'étude de systèmes à base de contraintes, l'activité de conception d'objets complexes et surtout l'ordonnancement et l'assemblage de ceux-ci se modélisant naturellement par l'expression de contraintes à satisfaire [Duverdier & Tsang 91, Hanser 90, Orchampt 87, Verroust 90].

Ce paragraphe situe succinctement les défauts actuels des SCAO et les travaux actuels en vue d'une évolution résolvant ces déficiences. Nous introduirons brièvement dans les chapitres suivants l'intérêt de l'insertion de techniques d'intelligence artificielle dans de tels logiciels.

I.10. Conclusion

Nous avons présenté ici de façon rapide les différents aspects des systèmes de CAO. Nous n'y avons pas inclus d'aspects très techniques car ce sont plutôt les différentes parties fonctionnelles du logiciel qui nous intéressent.

Dans la partie suivante, en vue d'intégrer des fonctionnalités de systèmes de CAO utilisant des techniques d'I.A., nous évaluons les possibilités de connexion entre les langages d'intelligence artificielle classiques (LISP, PROLOG) et des langages algorithmiques couramment utilisés pour la conception de SCAO (PASCAL,FORTRAN,C,...), pour des parties relevant de logiciels spécialisés en graphique (plus spécialement GKS).

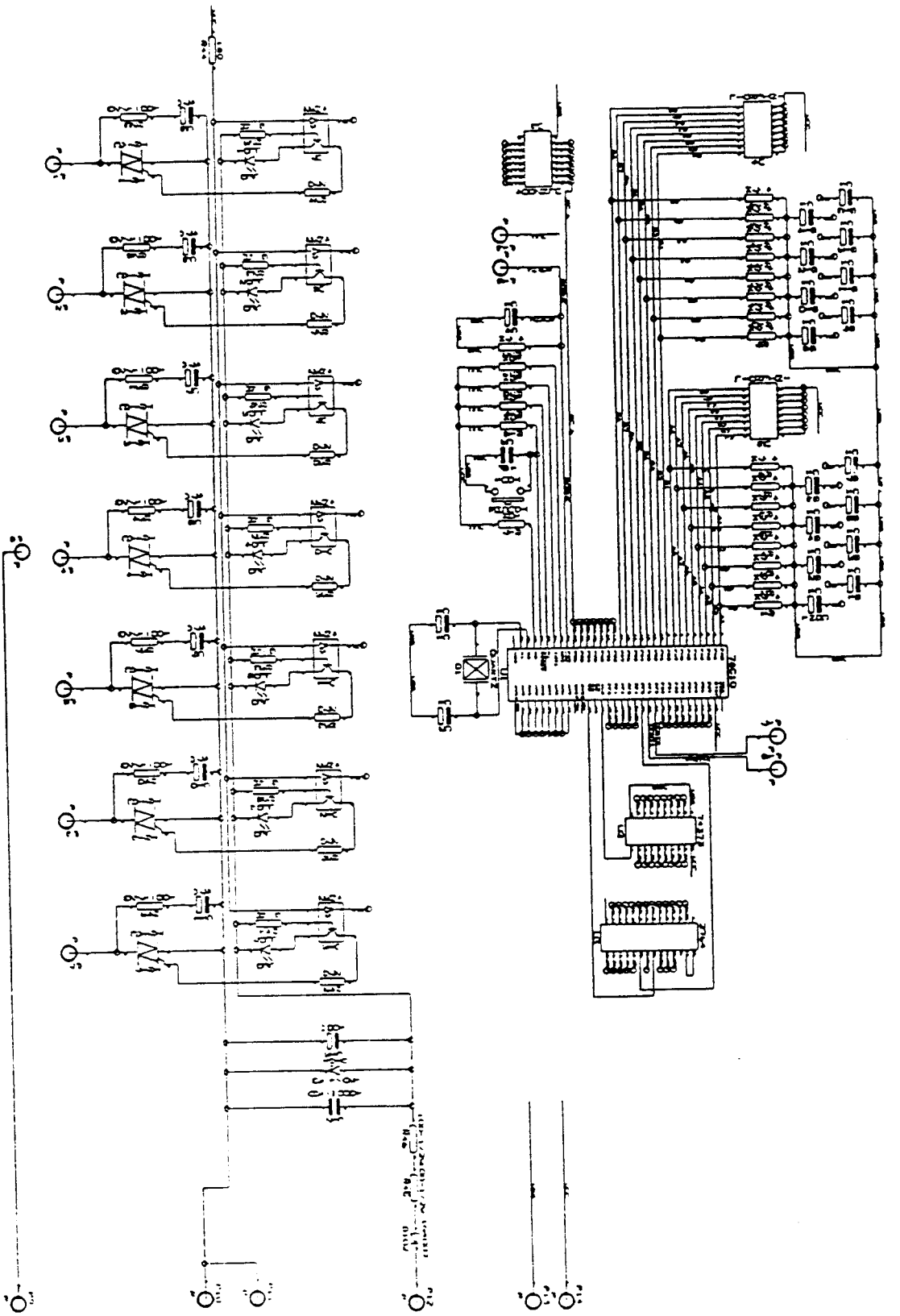


Fig I-7 *Schéma logique de circuit imprimé*

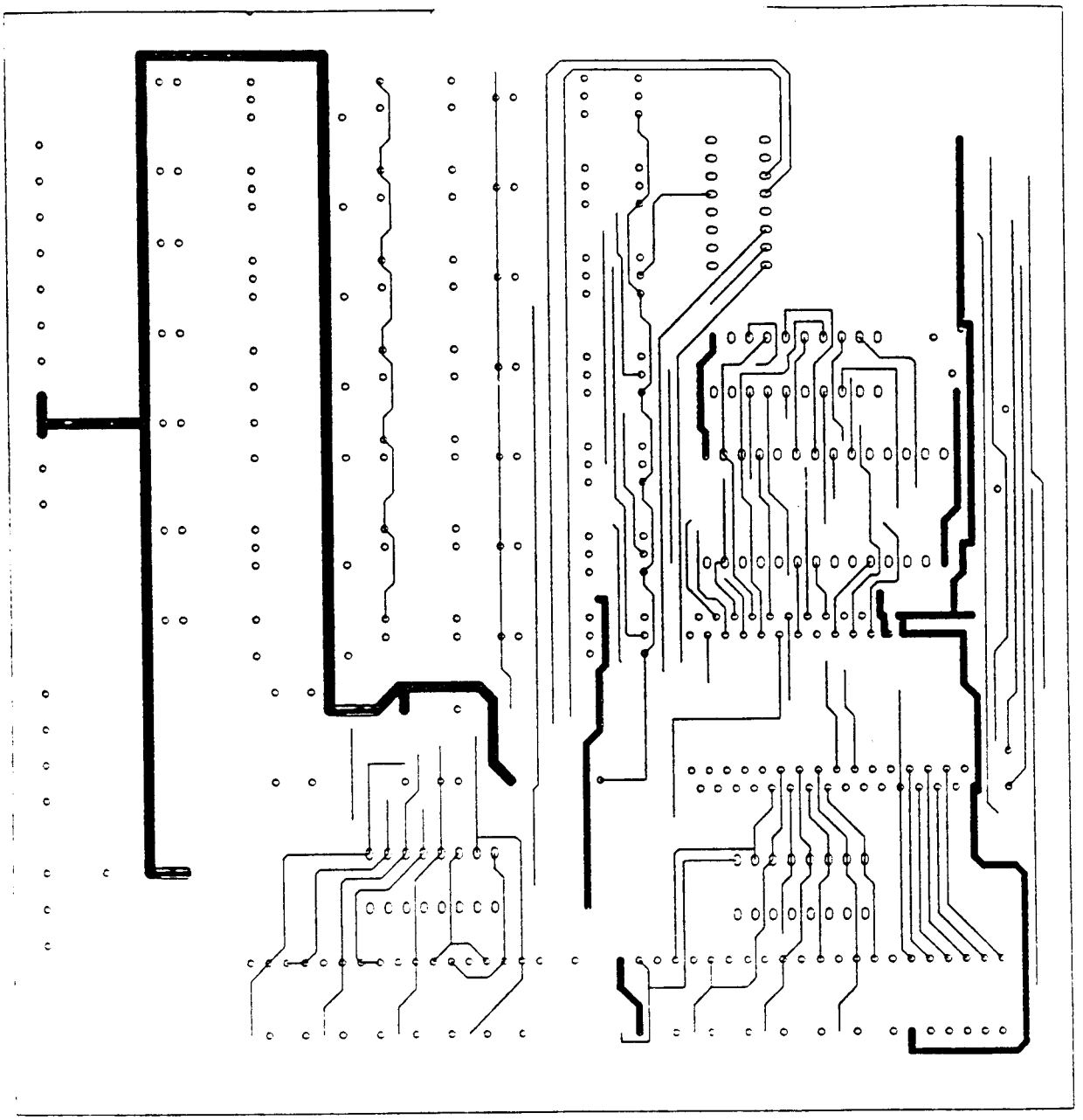


Fig I-8 *Routage de circuit imprimé (Face 1)*

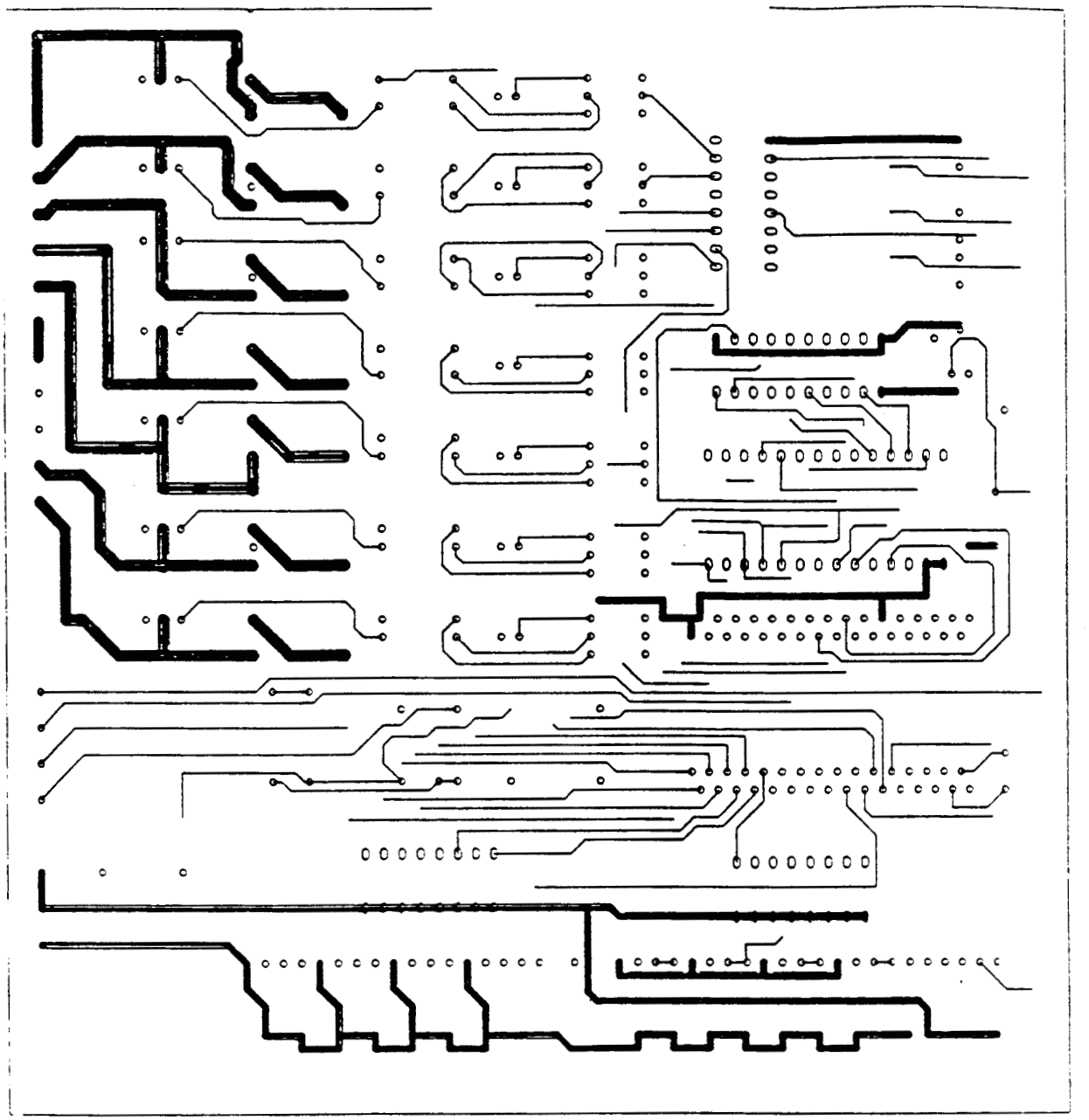


Fig I-9 *Routage de circuit imprimé (Face 2)*

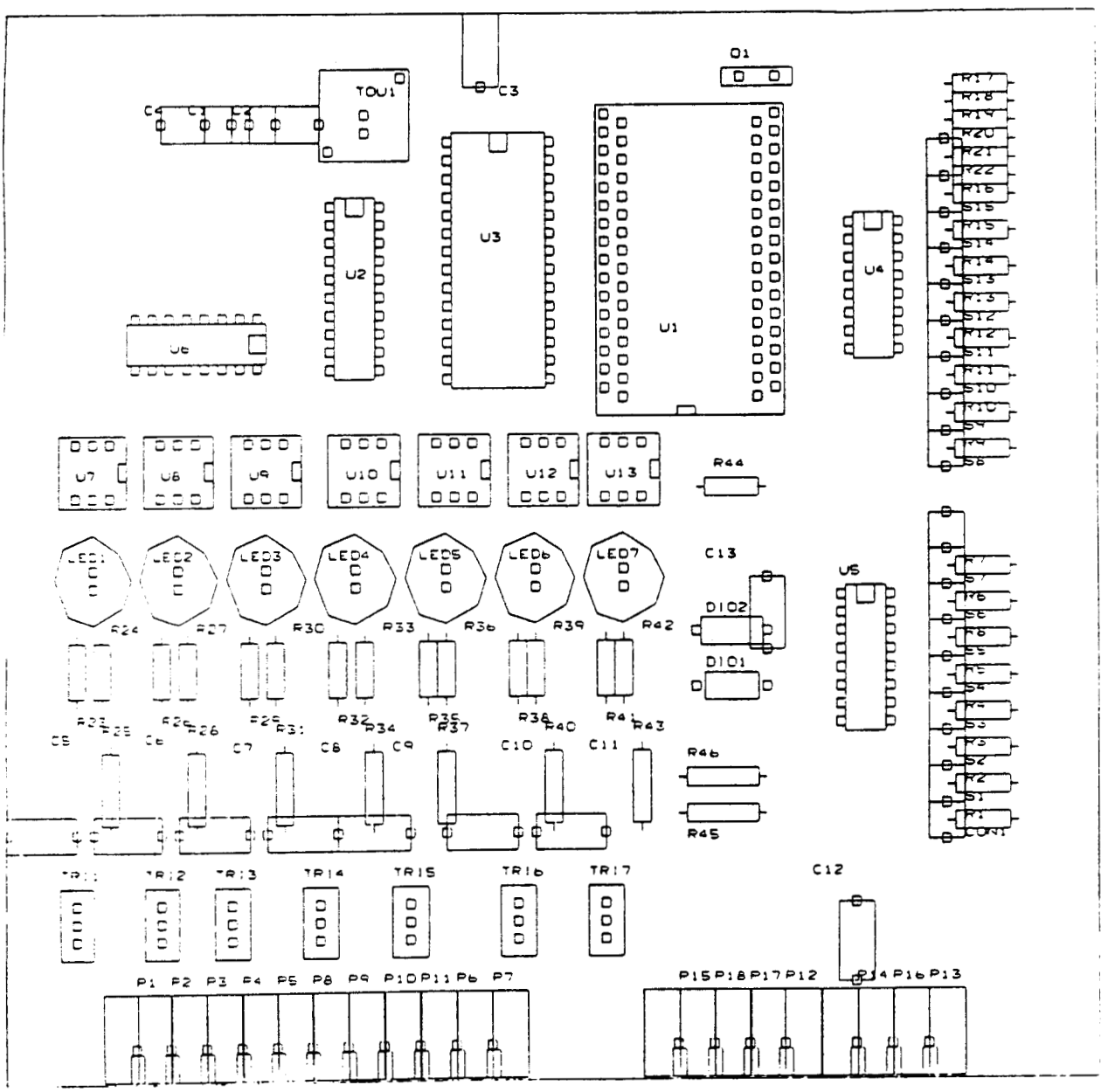


Fig I-10 *Placement de composants d'un C.I*

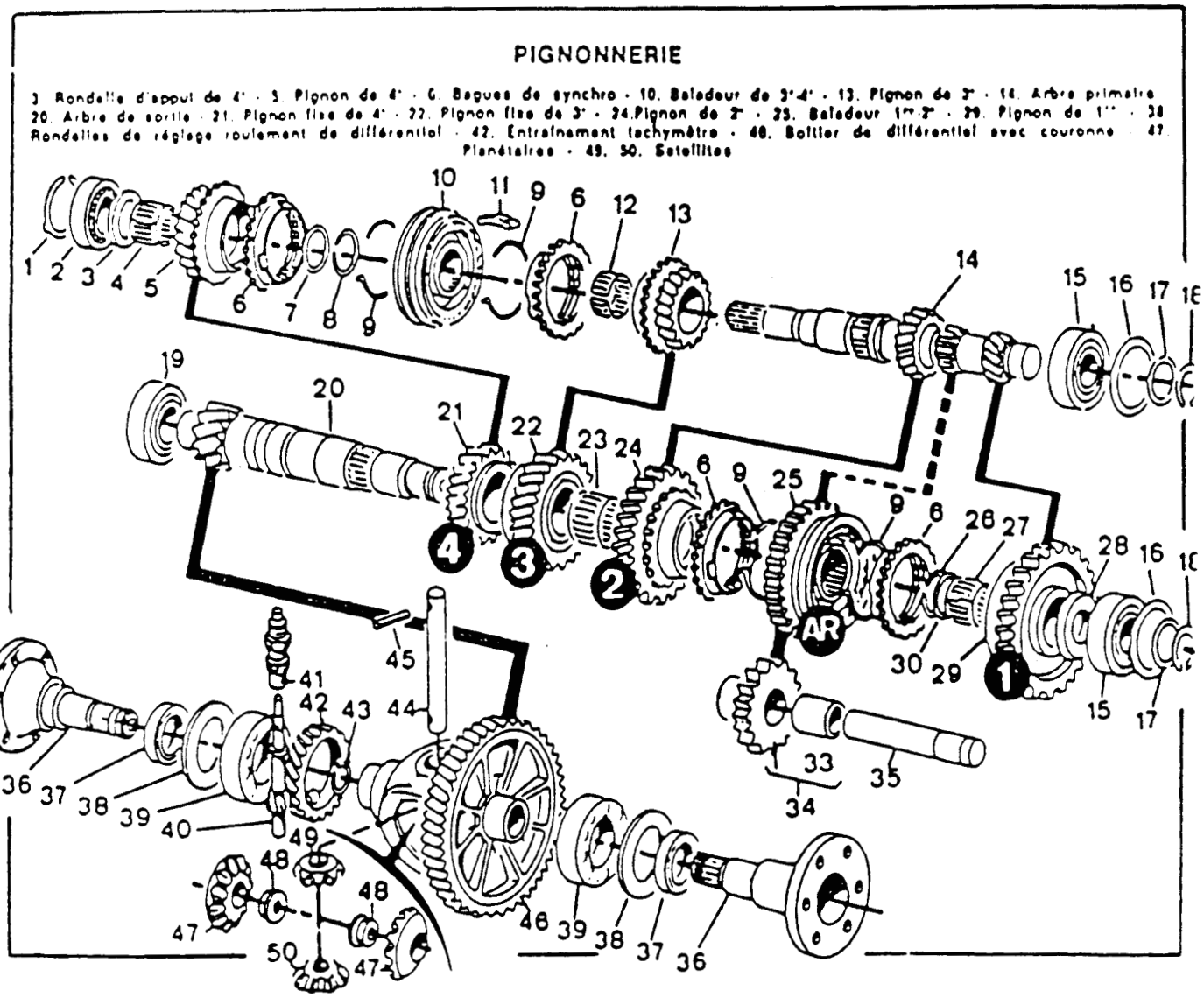


Fig I-11 *Décomposition d'une pièce en composants*

"Gabarit de façade" habillée manuellement de "Zip" par l'utilisateur

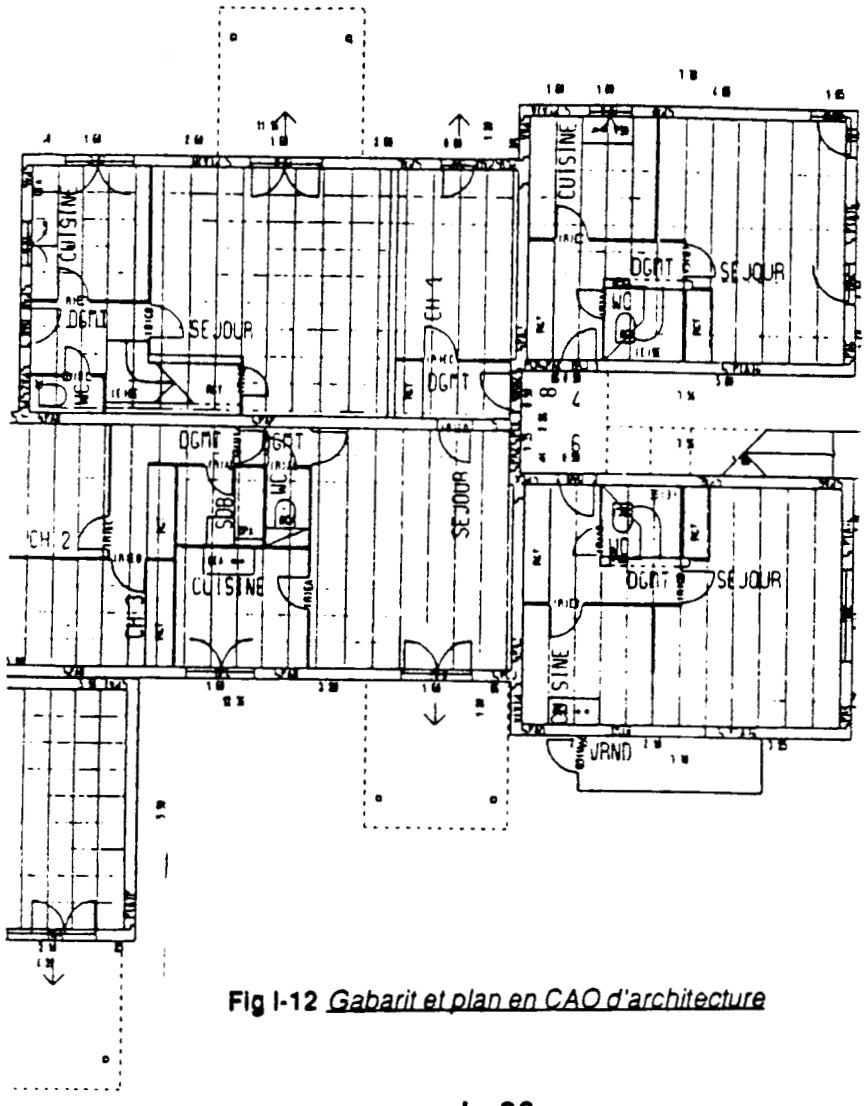
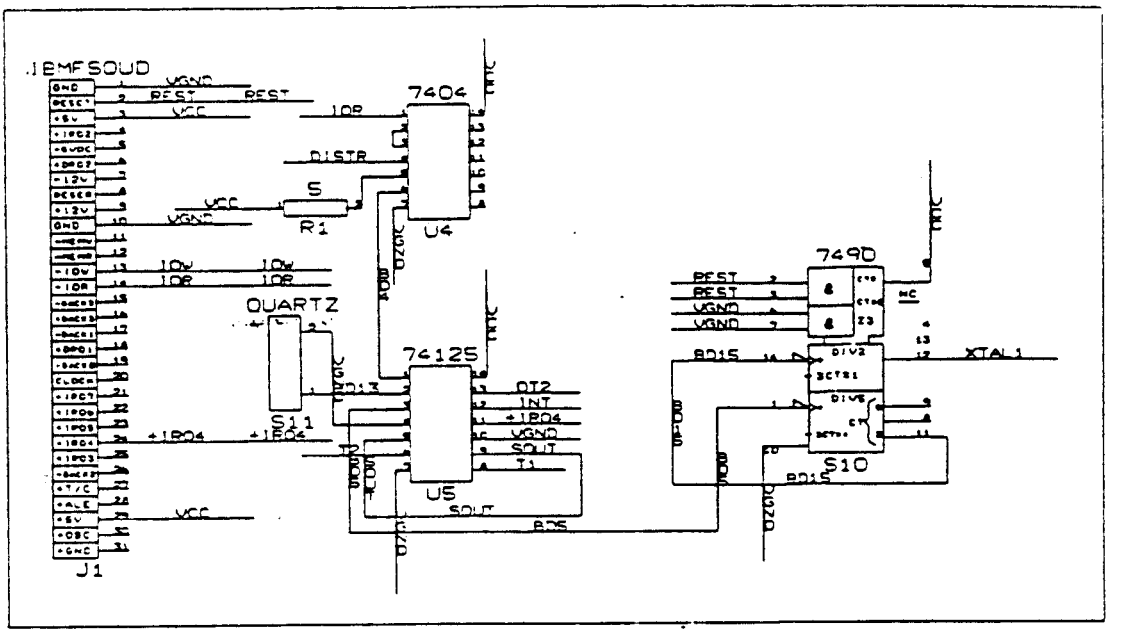


Fig I-12 *Gabarit et plan en CAO d'architecture*



METADESIGN CAE

PROJECT

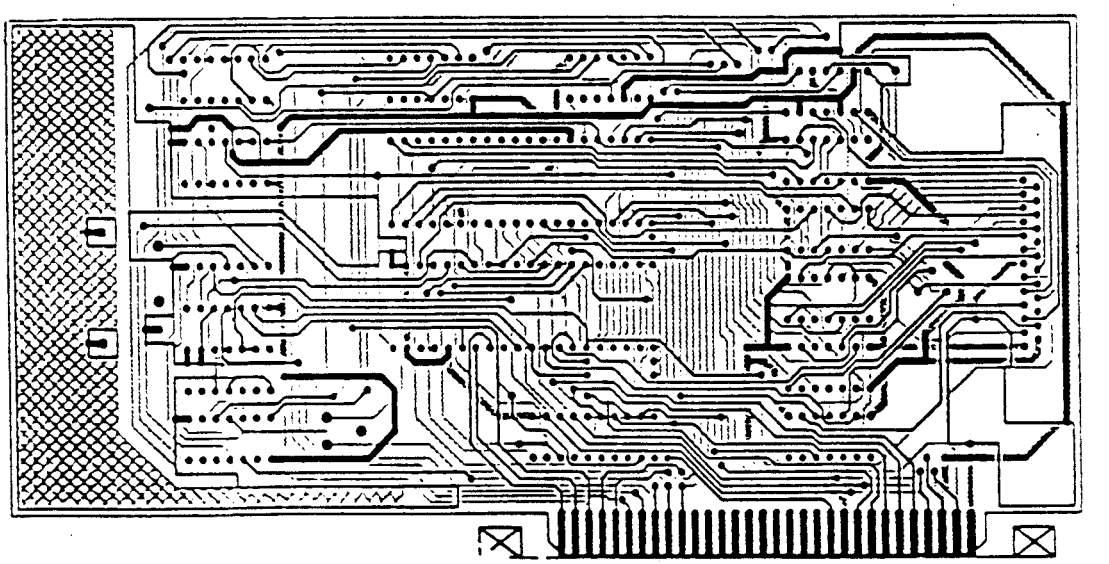


Fig I-13 *Projet de CAO électronique (en-tête)*

SCHEMATICS

Three programms will help you to obtain the drawing of the schematic diagramm and parts-/wirelist reports:

- *Component Data Base*

By installing your CAE programms, you will find already 1500 components in our data base.

But, more important, a special programm will help you to complete the data base by your own components.

- *Schematics entry*

In our schematics editor, the data base will send you the symbolic representation of components. You will indicate interconnections by drawing lines.

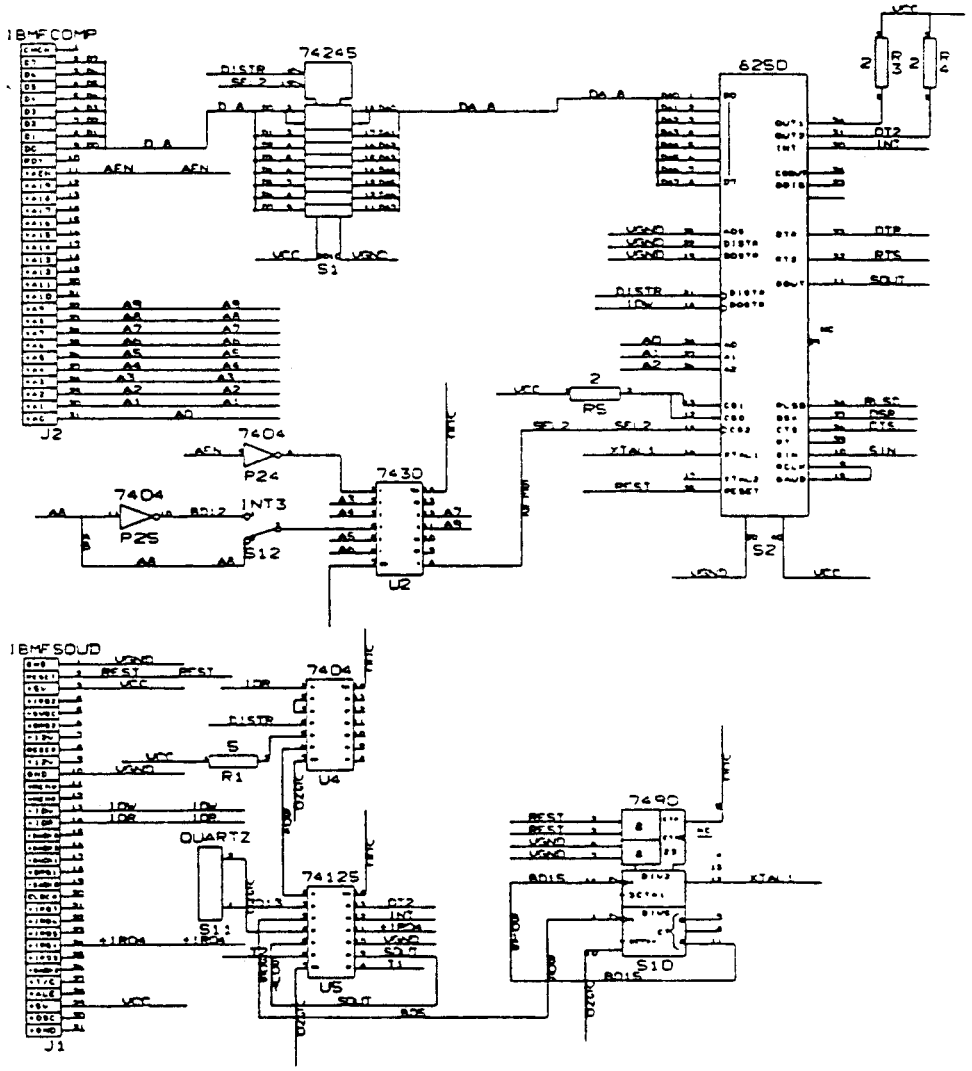


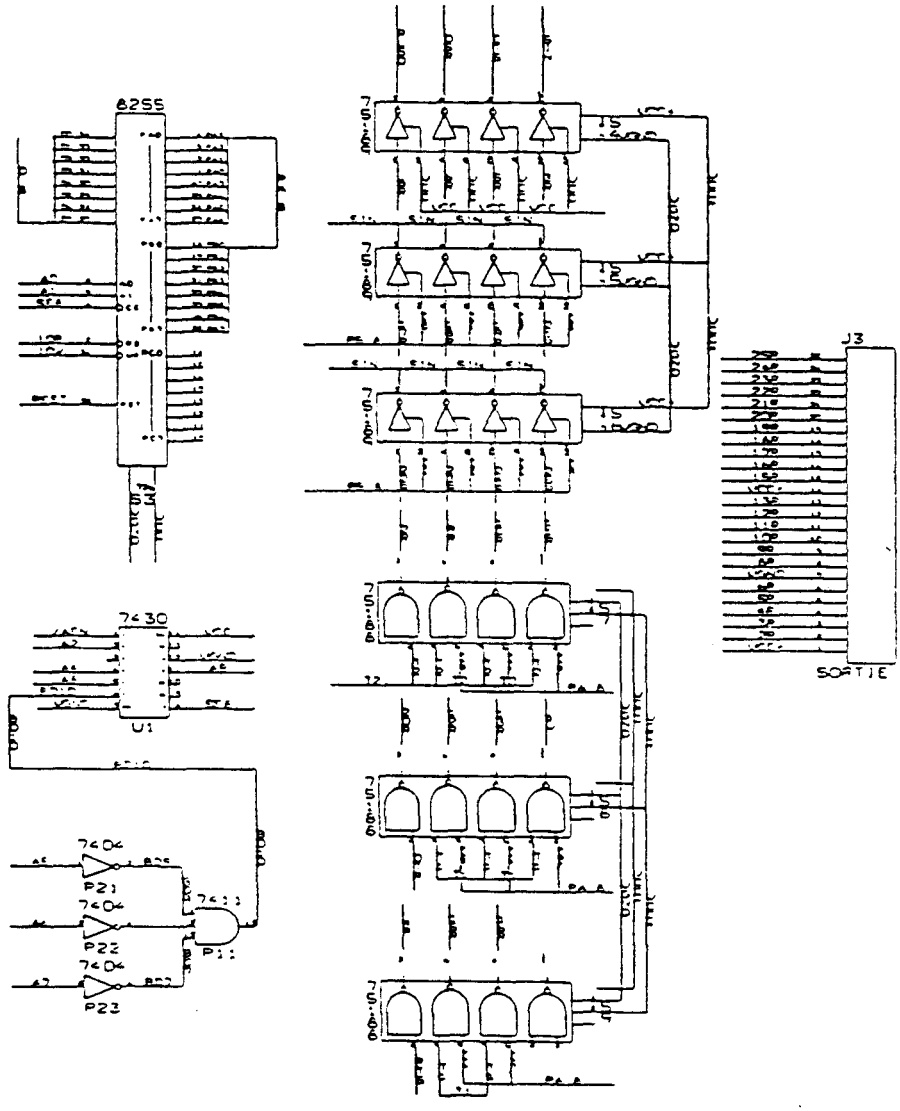
Fig I-14 *CAE : premier schéma*

- *Netlist reports:*

Once schematic entry is done, this programm interpretates the drawing and edits partslist, wirelist and warnings in case of basic errors (short-circuits, entries but no output pins, ...).

- *Option:*

Transfer programms will edit files to be sent to other electronics programms such as logic simulators or main frame routers.



DATE :	MODIF . :
RESP :	MODIF . :

Fig I-15 *CAE : deuxième schéma*

LIST- REPORTS

LIST OF CONNECTIONS / COMPONANTS

```
TYPE                    CONNECTOR IBM SIDE B (SOLDER)
J 1
| name pat. / connex num | nom pat. / connex num | nom pat. / connex num |
ent |  + IRQ2 /   N.C. 4 |  + DIRQ2 /   N.C. 6 | RESERV /   N.C. 8 |
|  -MEMW /   N.C. 11 |  -MEMR /   N.C. 12 |  CLOCK /   N.C. 20 |
|  + T/C /   N.C. 27 |                |                |
sor |  RESET /   REST 2 |  IOW /   IOW 13 |  IOR /   IOR 14 |
|  -DACK3 /   N.C. 15 |  + DRQ3 /   N.C. 16 |  -DACK1 /   N.C. 17 |
|  + DRQ1 /   N.C. 18 |  -DACK0 /   N.C. 19 |  + IRQ7 /   N.C. 21 |
|  + IRQ6 /   N.C. 22 |  + IRQ5 /   N.C. 23 |  + IRQ4 /   + IRQ4 24 |
|  + IRQ3 /   N.C. 25 |  -DACK2 /   N.C. 26 |  + ALE /   N.C. 28 |
|  + OSC /   N.C. 30 |                |                |
+/- |  GND /   VGND 1 |  VCC /   VCC 3 |  -5VDC /   N.C. 5 |
|  -12V /   N.C. 7 |  + 12V /   N.C. 9 |  GND /   VGND 10 |
|  + 5V /   VCC 29 |  + GND /   N.C. 31 |                |
TYPE                    CNCTS-CONNECTEUR IBM PC FACE A
J 2  — —
```

WARNINGS

```
OUTPUT PINS INCOMPATIBLE : + IRQ4
NO OUTPUT ON CONNECTION      : PA5
NO OUTPUT ON CONNECTION      : PA6
NO OUTPUT ON CONNECTION      : PA7
SHORT CIRCUIT SIGNAL/POWER  : 12R
SHORT CIRCUIT SIGNAL/POWER  : 13R
NO TARGET PIN CONNECTED     : 1V
OUTPUT PINS INCOMPATIBLE    : 2R
SHORT CIRCUIT ON CONNECTION  : 9R
SHORT CIRCUIT ON CONNECTION  : 10R
SHORT CIRCUIT ON CONNECTION  : 11R
```

Fig I-16 *CAE : nomenclature des connections*

The METADESIGN output list reports are edited in ASCII format. They resume informations about parts, wires, placement coordinates, non-connected pins, drill holes etc.

WIRE LIST

```
VGN     J1.1   J1.10  S11.2  S1.10  IC5.4  IC5.7
        IC5.10 IC4.7  IC2.7  S10.6  S10.7  S10.10
VGN     S2.19  S2.20  S2.22  S2.25  S3.7   IC1.7
        IC1.12 S9.7   S8.7   S7.7   S6.7
VGN     S5.7   S4.7   J3.7
REST    J1.2   S10.2  S10.3  S2.35  S3.35
VCC     J1.3   J1.29  R1.1   S1.20  IC5.14 IC4.14
        IC2.14 R5.1   S10.5  S2.40  IC21.14 R3.2
VCC     R4.2   IC11.14 S3.26 IC1.14 S4.2   S4.5
        S4.9   S4.12
IOW     J1.13  S2.18  S3.36
```

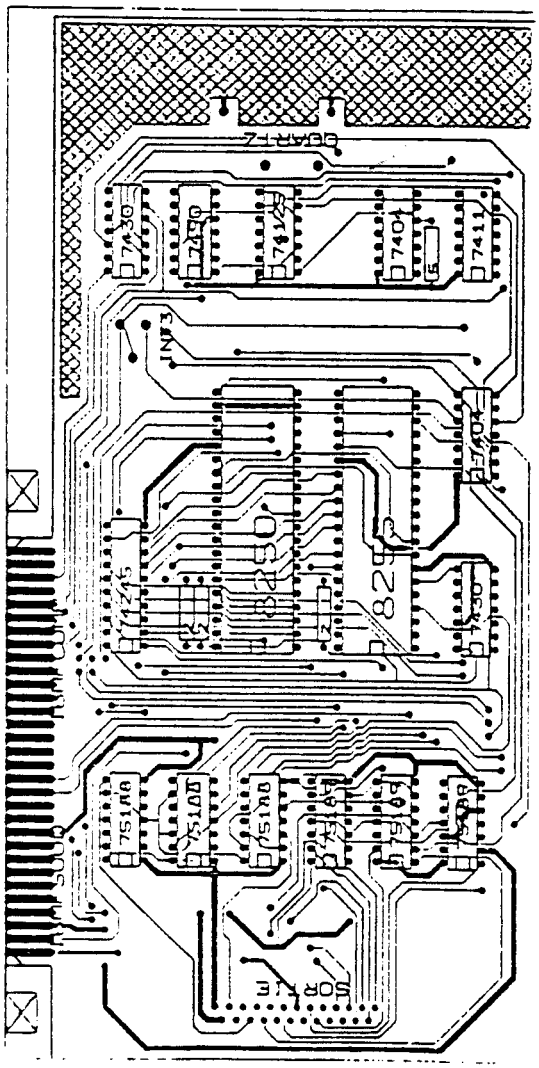
PLACEMENT COORDINATES

BOARD AURORE1 High : 100 mm Length : 200 mm
horizontal placement of ICs.
values are in 1/2000 inch refering to pin one .
origin (0,0) is the lower left corner.

log.No	Name	TYP	No pins	X1	Y1
K1	IBMFSOUD	CONNECT	31	1600	7500
K2	IBMFCOMP	CONNECT	31	1600	7700
P21	7404	C.I.	14	8800	600
S11	QUARTZ	SPECIAUX	2	13400	3300
S12	INT3	SPECIAUX	3	11000	5800
S1	74245	C.I.	20	6200	5800
U5	74125	C.I.	14	11800	3600

Fig I-17 *CAE : liste des connections (2)*

PLACEMENT



You don't need to retype any information about component outlines, number of pins or connexions to start placement and routing.

All these informations will be transfered from your schematics entry.

You determinate the rectangular or non rectangular shape of the board and the position of the components.

At any time the programm can indicate, by direct lines, the interconnexions of one component with already placed elements.

Fig I-18 *CAE : placement des composants (1)*

of the BOARD

A board corresponding to one schematic entry can be placed in 10 different ways in order to run the automatic-router for testing quickly the best results.

Once a board is placed and routed, the placement programm will draw the silk-screen mask with components names or logical numbers in user-definable position.

Aditionally, the programm edits placement coordinates and wrapping files.

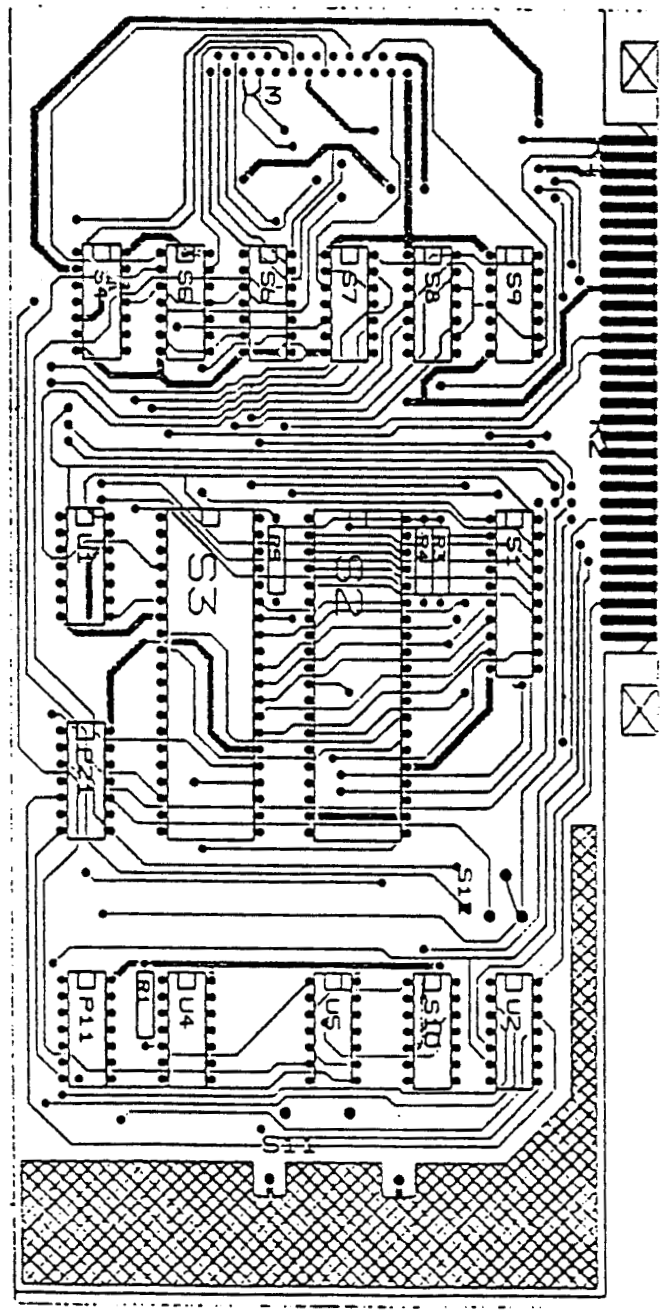
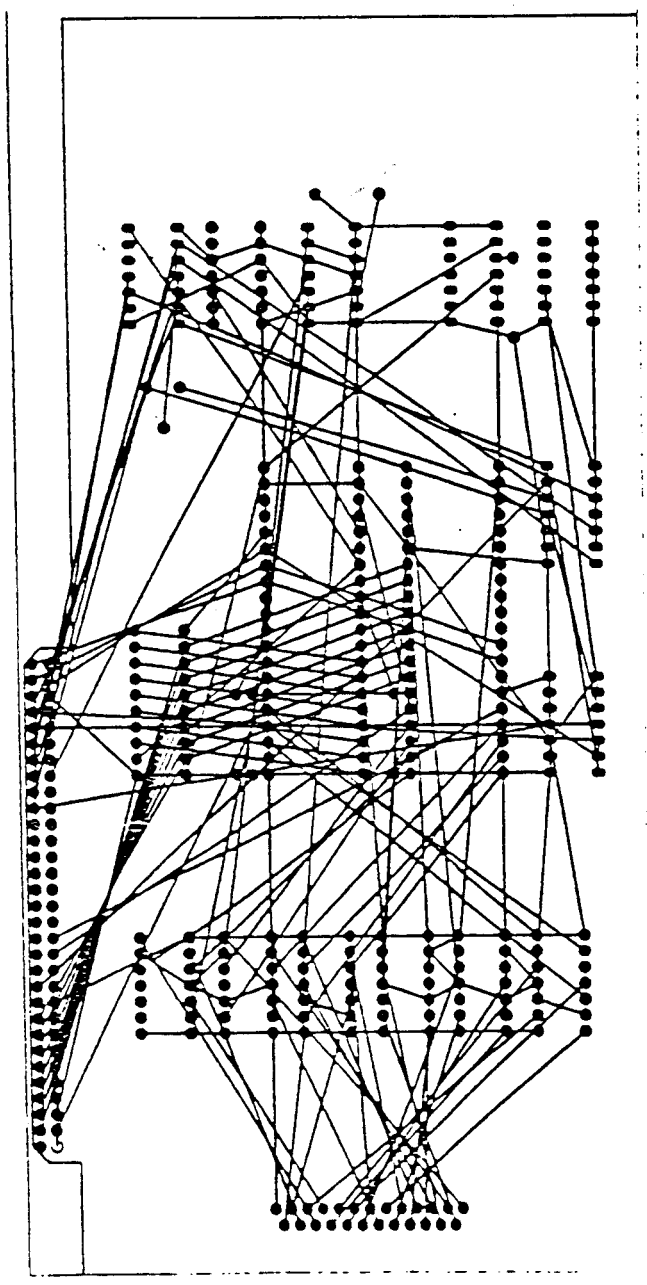


Fig I-19 *CAE : placement des composants (2)*

INTERACTIVEor



Routing can be done by interactif or automatic programs.

In fact, a mixed form appears always to be the best way to obtain industrial results.

You can first route interconnexions by yourself, start then auto-routing by specifying the connexions to be done, the number of vias allowed, clearance gap, route time limit etc , stop the router to control and modify some results, restart again ...

Fig I-20 *CAE : routage interactif (araignée)*

AUTOMATIC ROUTING

The pads used for routing can be defined corresponding to the needs of fabrication . In interactive routing, up to 250 different pads may be used simultaneously.

Routing can be done on a 4-layer-board. Each layer may have a different set of pads (economizing place on component or solder side, working with hidden vias or SMDs).

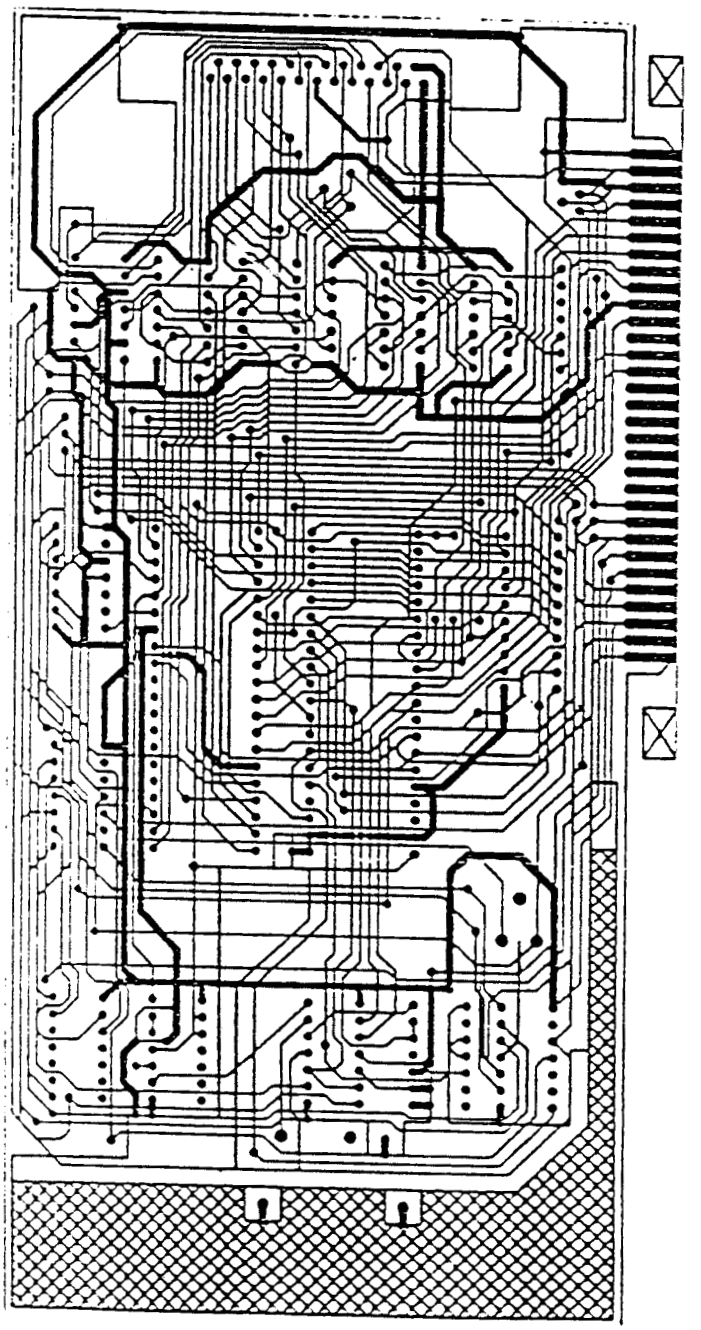
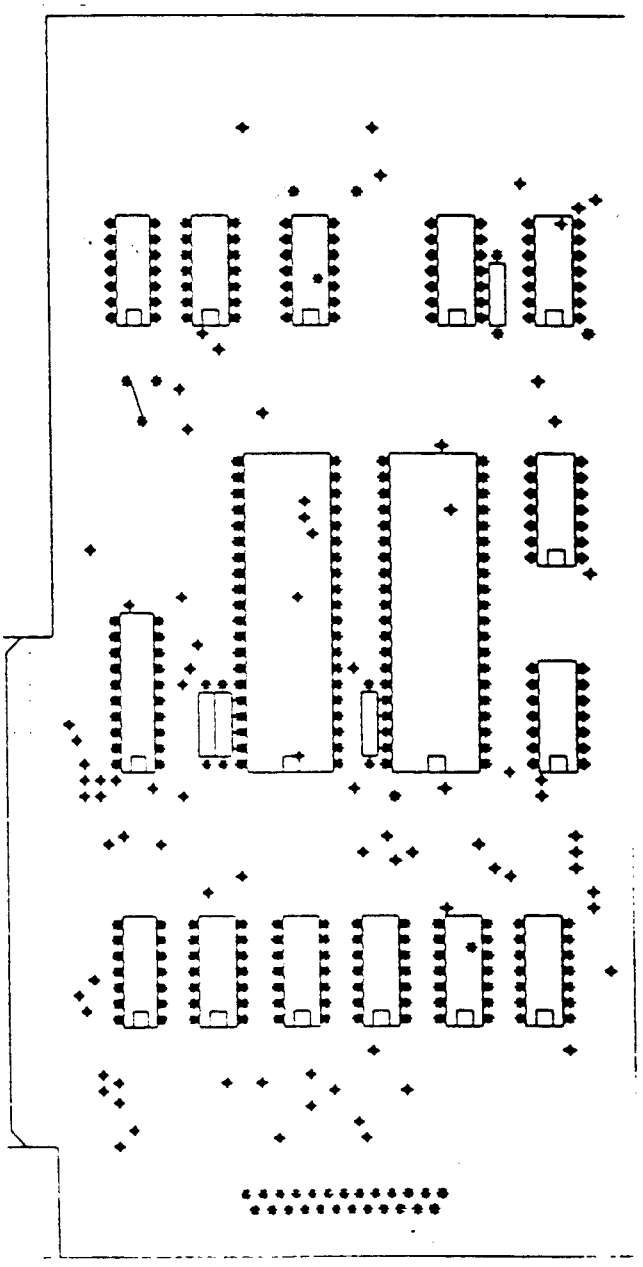


Fig I-21 *CAE : routage automatique (résultat)*

TOWARDS ..



Once routing is done, the programm will give you a set of facrication documents.

The film of each layer is drawn by

- photoplotters

- or pen-plotters at various scales.

Fig I-22 *CAE : document de fabrication1 (film)*

FABRICATION

Using various pad-data bases, you obtain automatically spare mask and drill mask.

ASCII files will give you a report about position and diameter of drill holes.

A complete annotated documentation about schematics and fabrication (text and illustrations) can be done by desktop-publishing (VENTURA) as you see in the present document.

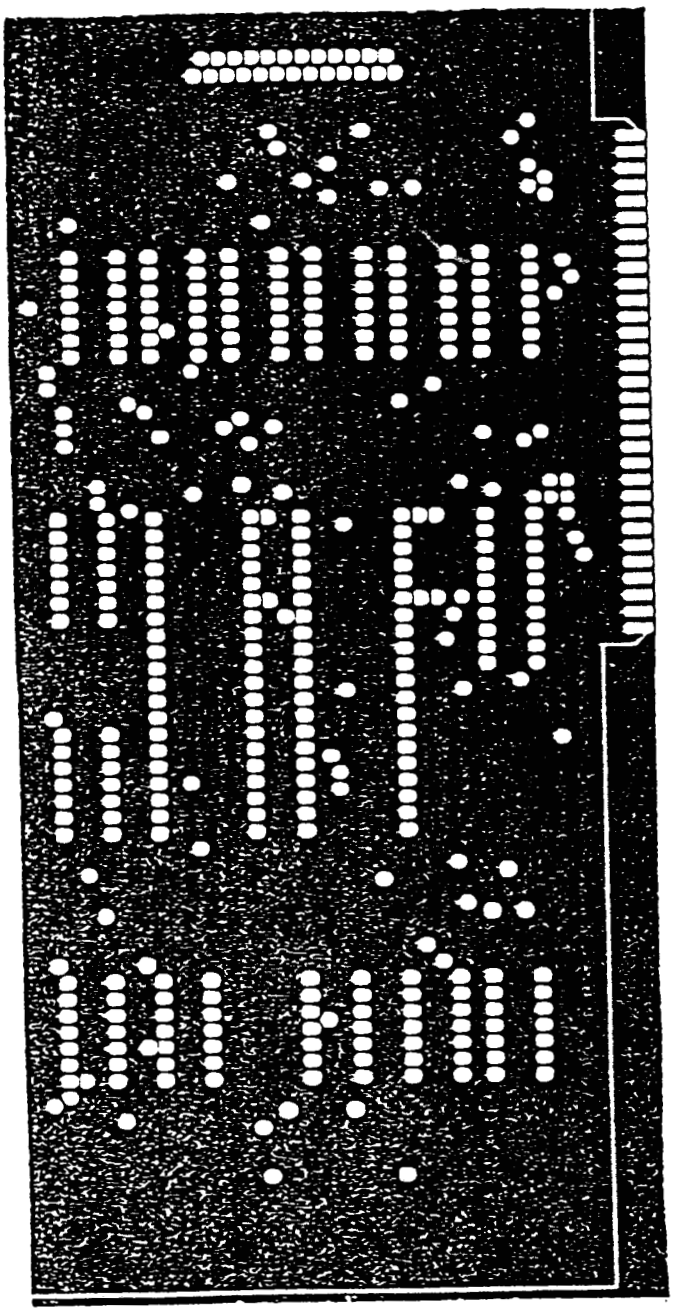


Fig I-23 CAE : document de fabrication (negatif)

Introduction au chapitre II

Les travaux que nous présentons ici concernent l'interfaçage entre les langages utilisés en intelligence artificielle, tels que LISP et PROLOG, et les langages algorithmique tels que C ou ADA. Il s'agit de l'étude de couplages forts entre ces langages.

Cette étude à été réalisée pendant l'année 85-86, à une époque où les langages IA commençaient à être diffusés sur le marché industriel, et où les recherches concernant l'intégration de logiciels débutaient. Si certains principes fondamentaux énoncés ici restent toujours d'actualité, le problème de l'intégration de logiciels n'est actuellement pas abordé par le biais d'un couplage fort entre logiciels, mais plutôt par l'intermédiaire de systèmes de communications dédiées permettant de conserver une indépendance assez forte entre les logiciels [ESF 89, Trousse 89]. Ceci a été rendu possible par l'avénement de réseaux efficaces et sûrs.

Cependant les résultats que nous avons obtenus sont toujours valables lorsqu'une communication à fort débit, ou lorsqu'un couplage fort entre les logiciels sont des paramètres importants de l'intégration.

Dans cette deuxième partie, nous abordons donc les problèmes logiciels, dûs à l'introduction de parties écrites avec un langage spécialisé en IA dans un logiciel écrit en langage procédural, comme c'est le cas des logiciels de CAO. Ceux-ci ont cependant des caractéristiques précises, en particulier au niveau graphique, et ces particularités nous ont servi de base pour l'interfaçage logiciel entre les deux types de langages.

La motivation initiale de cette étude est l'insertion de parties "intelligentes" dans un secteur fonctionnel du SCAO (interface homme-machine, bases de données, modélisation des éléments manipulés...). Quelle que soit la partie SCAO où est faite une telle insertion, on sera toujours confronté au problème suivant : comment communiquer des données entre la partie spécialisée en IA et la (les) partie(s) algorithmique(s), et réciproquement.

Si l'utilisation d'une machine particulière, d'un système d'exploitation précis, et d'implantations spécifiques de langage peuvent parfois permettre d'éviter ces problèmes de logiciel de base, il est hélas bien rare que les outils logiciels que l'on utilise, et que l'on a choisis pour leurs caractéristiques techniques, puissent coéxister au sein d'un même programme. Il faut généralement prévoir et construire une interface logicielle entre les deux parties.

On est alors amené à parler de "l'**interfaçage logiciel**". C'est ce que nous étudions dans ce chapitre, composé d'une étude générale théorique (II.1 à II.4), puis pratique (II.5), des problèmes et des formes de l'interfaçage, et d'une étude applicative à des implantations

industrielles de langages et de systèmes d'exploitation (II.6), cette dernière partie visant également à justifier les modèles présentés avant. Après quoi nous nous interrogeons sur le fait que les communications se font toujours en sens unique, du langage IA vers le langage procédural, et essayons d'évaluer l'opportunité d'implanter des communications dans l'autre sens, voire la coopération des deux parties (II.7).

Voyons maintenant plus précisément le contenu de ce deuxième chapitre.

Après explication de ce qui a motivé ce travail (II.1), nous introduisons le contexte technique de base de l'insertion de parties IA en CAO : algorithmique et historique des SCAO (II.2.1), intérêts de l'insertion de l'IA (II.2.2), problématique de l'intégration de l'IA dans un SCAO (II.2.3), caractéristiques essentielles des langages d'IA (II.2.4).

Quelques exemples précis de cas où l'interfaçage entre un langage d'IA et un langage impératif s'avère intéressant, et même parfois nécessaire, nous permettent ensuite d'introduire nos choix personnels d'études d'interfaçages (II.3.1, II.3.2), et de préciser le contexte logiciel généralétudié(II.3.3).

L'interfaçage logiciel entre les langages d'IA et les langages algorithmiques n'est pas un domaine réservé des SCAO; en fait, pour toute application algorithmique dans laquelle on veut insérer, ou créer, des parties spécialisées en IA, on rencontrera deux problèmes d'interfaçage : celui que nous avons appelé de "haut niveau", car il fait intervenir une description plus symbolique et déclarative des données et actions (par exemple la modélisation d'objets graphiques avec un langage d'IA), et un interfaçage de "bas niveau", que nous avons étudié ici principalement, et pour lequel on s'attache à résoudre les problèmes de système d'exploitation et de logiciel de base, problèmes qui surviennent quand les deux types de langage existent dans un même ensemble logiciel.

Nous nous sommes donc essentiellement intéressés à ce deuxième cas (II.4), que nous avons appelé **"interfaçage logiciel système"**, où nous décrivons le concept (II.4.1), introduisons un modèle descriptif permettant de classifier les interfaces (II.4.2), et citons finalement les problèmes rencontrés usuellement dans les implantations de langage procéduraux (II.4.3).

Les deux parties qui suivent donnent des exemples de telles interfaces. La première décrit certains de nos travaux d'interfaçage au sein de METADESIGN, pour des contextes algorithmiques et avec le système MS-DOS (II.5...). La seconde récapitule nos travaux d'études, pour différents langages d'IA et différents systèmes d'exploitation(II.6); chaque cas de langage est abordé suivant le même schéma: rapide description des particularités du langage et des données qu'on y manipule, principe d'interfaçage de base, application aux implantations de langage existantes. Un paragraphe dédié aborde succinctement les particularités des langages hybrides comportant une couche objet bâtie au dessus d'un langage algorithmique, comme par exemple C++.

Il était clair que nous ne pouvions voir tous les cas d'interfaçage logiciel entre milieu IA et de CAO. Néanmoins, nous n'avons pas voulu passer sous silence des problèmes plus théoriques d'une telle interface (II.7). Pour chaque sens de communication des données, nous essayons de mettre en évidence les problèmes que l'on rencontrera à priori : dans le premier cas pour la communication IA vers algorithmique (II.7.1), puis pour la communication en sens inverse (II.7.2), et enfin lorsque les deux procédés coopèrent (II.7.3). Cette dernière partie est une étude initiale, mais elle montre cependant les problèmes conceptuels pour de telles interfaces et plus généralement pour une intégration de base entre logiciels IA et logiciels procéduraux.

Nous concluons (II.8) en faisant une synthèse du chapitre, et sur les possibilités d'interfaçage ultérieures, qui ne manqueront pas d'intérêt pour l'utilisation des techniques IA dans les SCAO et les logiciels impératifs, un tel interfaçage ou mieux une telle intégration étant à notre avis une évolution indispensable des logiciels CAO du futur, si l'on veut fournir des services plus évolués ou sophistiqués que ceux d'aujourd'hui.

II. INTERFACAGE IA-SCAO

Nous étudions dans ce chapitre les possibilités de coexistence de parties logicielles intelligentes, écrites dans un langage d'Intelligence Artificielle (I.A.) tel que PROLOG, avec des parties logicielles utilisant des techniques algorithmiques classiques telles qu'on en trouve dans les SCAO, écrites avec un langage procédural tel que C, FORTRAN, ... La partie intelligente pourrait être par exemple un système expert en conception et la partie algorithmique un ensemble logiciel de modélisation et de conception CAO.

Insistons sur le fait que nous nous intéressons ici seulement à la partie communication de base, sans regarder l'influence cette communication sur la modélisation des données dans la partie algorithmique CAO et sur la représentation des connaissances dans la partie IA. Ces aspects important pour l'intégration des environnements IA et CAO ont fait récemment l'objet de quelques études [Favard 89, Gardan & Zachari 91, Trousse 88 et 89] qui montrent l'intérêt et l'importance d'une modélisation correcte et adaptée des objets à manipuler en CAO.

La cohabitation des deux contextes logiciels, IA et procédural, peut prendre trois formes, suivant que l'on considère le sens des communications d'un langage vers l'autre :

- les programmes d'application IA utilisent, ou gèrent, les fonctions des programmes procéduraux. Le programme procédural est alors l' "esclave" du programme IA [Jonckers 85].
- Les programmes d'application procéduraux utilisent, ou gèrent, des programmes IA spécialisés dans le domaine de l'application. Le programme IA est alors l' "esclave" du programme procédural.
- les programmes IA et procéduraux cohabitent dans un même ensemble logiciel, et coopèrent réellement dans l'ensemble logiciel final.

Les possibilités précitées forment à elles trois un sujet d'étude très vaste. Cependant la première possibilité est celle dont les intérêts sont les plus immédiats et notre travail concerne essentiellement ce premier cas, en s'attachant à résoudre les problèmes de logiciel de base entre les deux contextes. Nous étudions de façon succincte les deux derniers cas à la fin du chapitre.

II.1. Les raisons de cette étude

Ce travail a été réalisé dans la société METADESIGN située à LILLE, spécialisée dans le développement de logiciels de base infographiques , METADESIGN-GKS [GKS4.0 88], et d'applications graphiques, plus précisément en CAO, METADESIGN-CAE [CAE 4.0 88].

Après qu'un prototype du logiciel de CAO électronique, METADESIGN-CAE, ait été conçu et réalisé, l'équipe technique, soucieuse de rester à l'avant-garde en matière de CAO, désirait pallier à certains manques classiques des SCAO :

- insuffisance de l'aide apportée au concepteur-utilisateur du SCAO [Dejesus & Callan 85, Favard 89],
- défaillance des logiciels réalisant seuls des tâches de conception, par exemple les algorithmes de placement-routage [Carre 85a],
- inadéquation des interfaces homme-machines aux exigences des utilisateurs (difficultés d'apprentissage, inconsistances dans le paramétrage des commandes suivant les parties des logiciels...).

Une première étude fut donc entreprise, concernant le placement et le routage de composants sur les cartes de circuits imprimés en CAO. électronique [Carre 85b], pour essayer d'utiliser les techniques classiques d'intelligence artificielle (systèmes experts avec bases de connaissances), afin d'automatiser partiellement ces opérations de conception, d'une manière plus fiable que par des méthodes algorithmiques classiques.

Cependant, il n'était pas question, de modifier l'architecture logicielle interne du SCAO ; il s'agissait en fait de greffer sur cette structure des modules "intelligents". On désirait ensuite réaliser une panoplie de tels modules, intégrés au SCAO, chacun d'eux réalisant automatiquement des tâches spécialisées en conception électronique. Enfin, on désirait voir si une généralisation à tous les SCAO était possible.

Une approche immédiate pour réaliser un système expert dans un domaine donnée de conception, consiste à décomposer le travail de l'expert placeur-routeur en tâches de nature simple et relativement indépendantes (placement et routage des mémoires, cas des alimentations, des multiplexeurs...). On espère ainsi trouver une décomposition en sous-tâches suffisamment simples pour être réalisées par des algorithmes spécialisés. Le système expert est alors considéré comme un "bon utilisateur" de ces parties impératives. Ce principe a déjà été abordé dans [Jonckers 85].

Il est alors clair qu'il faut interfacer les langages courants d'intelligence artificielle avec lesquels sont réalisés les modules "intelligents" (LISP, PROLOG, pour ne citer que les plus connus), avec des langages procéduraux classiques (C, PASCAL, FORTRAN, ADA ...), avec lesquels sont développées et réalisées les parties algorithmiques du SCAO.

Il s'agit en fait de concevoir une "passerelle" ou "connexion" logicielle entre ces deux types de langages. Cela est rarement prévu explicitement par les concepteurs des langages, du moins pour des implantations de langage de provenances différentes. D'autre part, une telle réalisation permet de connaître précisément les problèmes importants que l'on doit résoudre quand on crée ce genre d'interface : paramètres et données passées par le module expert inadaptées et incompatibles avec la partie algorithmique, nature dynamique des données en IA.

Mais ce genre d'interface est à sens unique ; la communication inverse est-elle envisageable, est-elle viable ? Ce sont des questions auxquelles nous avons tenté de répondre ; du moins partiellement, car l'essentiel de notre étude a porté sur l'utilisation de parties impératives par des parties IA

Nous étudions le premier cas d'interfaçage que nous avons cité, c'est à dire une partie IA qui utilise une ressource procédurale, sur quelques exemples concrets, nous donnons la correspondance entre chaque cas rencontré et notre classification de modèles d'interface logicielle, puis nous apportons des éléments de réponse pour la liaison inverse : partie algorithmique utilisant des parties IA.

Cette étude est donc un travail de type logiciel de base, pour l'intégration de modules experts dans les SCAO.

II.2. Aspects de l'interfaçage IA-SCAO

Nous décrivons ici le contexte technique dans lequel s'inscrit cette étude et étudions l'intérêt de l'interfaçage envisagé.

II.2.1. Historique du contexte algorithmique des SCAO

On a coutume [Giambasi & al 83] de classer les différents stades d'évolution des SCAO, des premiers à ceux d'aujourd'hui, en trois phases :

- Les SCAO dits de première génération, où le système est constitué d'un ensemble de logiciels indépendants qu'on utilise l'un après l'autre, dans un ordre établi par le constructeur du SCAO, sur le projet qu'on aura préalablement décrit en parties (par des fichiers, des listes de données...). La description est manuelle et sans interaction avec le système. Ce sont des outils très lourds à manipuler, qui ne sont guère utiles que pour réaliser des tâches très spécialisées, et que l'on sait décrire formellement (calcul de structure, représentation graphique...).
- La deuxième génération, caractérisée par l'insertion du traitement par lots, autorise l'utilisateur à se servir à son gré des ensembles de programmes adaptés à une partie spécifique de la conception dans le domaine. Il établit lui même la suite de traitements qui seront effectués sur le projet, en décrivant l'enchaînement des programmes à exécuter. Les programmes concernés sont par exemple des programmes de simulation de fonctionnement, de contrôle technique, de simulation électrique, de calculs de structures ... Par rapport à la génération précédente, il y a donc essentiellement un accroissement dans la souplesse d'utilisation de ces logiciels.
- Enfin les logiciels modernes, dits de troisième génération, intègrent tous ces programmes et parties logicielles au sein d'un seul logiciel, et surtout apportent l'interactivité généralisée entre l'homme et la machine, ainsi qu'une vision élaborée des résultats (par exemple une trace graphique au lieu d'importantes descriptions alphanumériques sur papier...).

Mais dans tous ces systèmes, l'élément qui prédomine est le programme ou algorithme spécialisé dans une tâche particulière, comportant souvent une part importante de calcul. Qu'il s'agissent de SCAO appliqués à l'électronique, incluant des programmes de placement, de routage, de simulation logique ou analogique, à l'architecture et au bâtiment (programmes de modélisation volumique, calculs de cotation) ou encore à la mécanique (programme de résistance des matériaux, éléments finis, de modélisation volumique) tous sont de gros consommateurs de calcul, et la programmation impérative y prend une place essentielle.

Notons aussi que les SCAO actuels ne réalisent actuellement que des fonctions et des traitements qui ont être modéliser sous forme d'algorithme, telle que la représentation graphique des éléments par exemple. En d'autre termes, ils n'apportent pas, ou très peu, et de manière indirecte, une aide réelle à l'utilisateur, car ils ne traitent pas l'information symbolique.

II.2.2. Intérêts de l'insertion de l'IA dans les SCAO

Introduire des techniques d'Intelligence Artificielle dans des systèmes de CAO n'est pas une idée nouvelle [Descottes 81, Latombe 77, Schramel 86], et beaucoup de travaux de recherche récents portent sur l'intégration et la coexistence de système IA dans des logiciels de CAO [Favard 89, Orchampt 87, Trousse 87,]. Nous faisons dans cette section un tour d'horizon des avantages cités couramment à l'introduction de systèmes ou de techniques IA dans un SCAO:

- Les logiciels de CAO modernes deviennent des outils de plus en plus sophistiqués et de plus en plus complexes. De plus, ils ne fournissent plus seulement un ensemble de logiciels épars, mais intègrent de plus en plus de sous-systèmes logiciel sophistiqués. Enfin, ils doivent également devenir des outils flexibles. Cette évolution n'est pas assimilable par les techniques informatiques traditionnelles [Latombe 85]. Les SCAO doivent non seulement présenter à l'utilisateur les informations d'une manière plus élégante, facilement assimilable, ou fournir des outils d'analyse de ces informations, mais également assister réellement le concepteur dans sa tâche, en résolvant certains problèmes très spécifiques de conception, ou en fournissant des systèmes d'aide à la conception [Latombe 77, Latombe 85, Orchampt 87]. Ces aides seront plus facilement réalisées en employant des techniques IA.
- Nous avons vu qu'il existe principalement deux méthodologies de conception : la méthode ascendante et la méthode descendante. Si du point de vue théorique ces deux approches sont bien connues et utilisées naturellement par les concepteurs, les logiciels actuels ne fournissent en général qu'une seule de ces deux approches de conception. Etre capable de traiter les deux approches nécessite de prendre en compte l'aspect incomplet et incertain d'une tâche de conception. Les techniques IA permettent d'aborder plus facilement ces caractéristiques d'incomplétude et d'incertitude [Moulin 90, Neveu & al 90, Trousse 89]. Parallèlement à ces traitements, intervient la gestion de version d'artefacts incomplets; des techniques de

programmation et de représentation des connaissances orientées objets ou à base de frames permettent de mieux gérer ces objets incomplètement définis [Carre 89, Dugerdil 85, Trousse 89].

- L'une des activités les plus fréquentes dans le domaine général de la conception est l'assemblage de composants en vue de fabriquer un objet complexe. Mais ces éléments sont des objets qui obéissent à certaines lois de nature physique ou fonctionnelle. Chacun des éléments de l'ensemble doit donc satisfaire certaines contraintes dépendantes de lui-même et de l'ensemble dans lequel il se trouve. Ces contraintes peuvent être de nature spatiale (pour le placement-routage de composants sur une carte de circuit imprimé par exemple) ou plus fonctionnelles. Pour aider le concepteur à concevoir son produit dans ces cas spécifiques, il serait utile d'intégrer des capacités de raisonnement et de résolution à base de contraintes[Du Verdier & Tsang 91, hanser 90, Verroust 90].

- Les techniques algorithmiques classiques fournissent des solutions sur des problèmes précis et bien définis. Or il n'existe pas en conception de méthode générale pour obtenir des solutions, et bien souvent le problème à résoudre est incomplètement défini. La conception est une activité créatrice qui se fait par tâtonnements et essais successifs. D'autre part la diversité des solutions possibles pour un problème donné conduit très vite à une explosion combinatoire que ne peut prendre en compte l'algorithmique classique. Les techniques IA permettent justement de pallier à ces problèmes. Il semble donc normal de penser que ces techniques peuvent fournir des outils pour concevoir des logiciels de CAO de haut niveau intégrant un partie de l'activité de conception.

- Jusqu'à présent les SCAO existant ne fournissent des aides que sur leur utilisation ou fonctionnement interne; ils ne savent pas conseiller l'utilisateur sur le choix d'options techniques particulières. Pour pouvoir fournir ce genre d'aide, il faut que le système soit capable de raisonner sur l'objet en cours de conception à partir de connaissances spécifiques d'un domaine. C'est pourquoi les divers systèmes de représentation de connaissances issus de l'IA peuvent apporter beaucoup à la réalisation de système d'aide intelligent.

- Les modèles existant en CAO ont été conçus pour donner une représentation fonctionnelle ou visuelle réaliste des objets. Cependant ils sont incapables de prendre en compte les relations spatiales ou fonctionnelles existant entre les objets [Trousse 89]. Toujours dans un but de perfectionnement et d'aide à l'utilisateur, il faut fournir des modèles capables de prendre en compte ces relations. L'IA peut fournir justement des modèles satisfaisant sur ce point.

- On manipule et travaille souvent en CAO sur des images ou des schémas. La sémantique visuelle des images joue un rôle capital dans l'activité de conception. Les informations contenues dans l'image elle même permettent au concepteur de mieux appréhender les points à résoudre et donc de trouver plus facilement une solution. Le vieil adage "*Un bon dessin vaut mieux que mille explications*" est particulièrement bien adapté à la CAO. Si les systèmes actuels permettent une représentation graphique

agréable et sophistiquée, ce qui est déjà un point positif par rapport à une représentation uniquement textuelle, il n'existe en fait aucune information sémantique attachée à la représentation d'un objet dans le SCAO. Si l'on veut fournir à l'utilisateur un véritable outil de conception, plutôt qu'un éditeur graphique dédié puissant, il faut en fait que l'outil prenne également en compte le rôle des éléments manipulés, aussi bien fonctionnel que géométriques ou topologique. Cette simple capacité permettrait par exemple de pouvoir tester en permanence et facilement la cohérence de l'état courant de l'objet conçu. Encore une fois les techniques IA permettent de représenter plus facilement un grand nombre de relations possibles entre objets et sont donc bien adaptées à cette capacité. Parallèlement à cet aspect de prise en compte de la sémantique des objets manipulés, les concepteurs considèrent les objets qu'ils manipulent sous différents aspects; des modèles multi-représentation des connaissances produiront des représentations informatiques en bon accord avec les divers modèles mentaux utilisés par les concepteurs, et pourront donc être traités d'une manière cohérente vis à vis de leur utilisation par les spécialistes.

- Certaines activités de conception consistent à résoudre des problèmes particuliers, tester le fonctionnement du produit, ou encore en vérifier la cohérence. Jusqu'à présent les SCAO fournissaient parfois des logiciels algorithmiques dédiés à ce genre de problème. On doit hélas constater que bien souvent les solutions apportées par de tels algorithmes généraux restent incomplètes et que bien souvent le concepteur doit remettre en cause une partie non négligeable du travail effectué par ces algorithmes spécialisés. En tenant compte du domaine d'application et de l'expertise de concepteurs, on peut penser que des systèmes experts résoudraient plus efficacement certains travaux de conception ou de contrôle que des programmes purement algorithmique, tout en conservant l'aspect "automatisation" de certains travaux considérés comme routiniers et répétitifs par les concepteurs.

- Les SCAO actuels, conçus et construits à partir de techniques et langages procéduraux, sont incapables [Gardan 86a, Lebahar 85] de traiter les informations du point de vue symbolique, soit parce que l'on ne leur a pas donné cette capacité (choix à priori), soit parce que d'autres problèmes (modélisation géométrique, bases de données, simulation, environnements intégrés) furent considérés plus importants ou urgents à résoudre. Seules, les parties fortement liées à la technologie des bases de données, parce qu'elles forment un élément essentiel dans le SCAO, et parce que les nomenclatures techniques sont nombreuses et de formes diverses, et aussi parce que mal adaptées à la CAO [Cholvy & Foisseau 83b, Cholvy & Foisseau 85], en particulier la base de donnée projet et les bases de données techniques, ainsi que la partie graphique interactive, pour la manipulation d'objets graphiques complexes, ont été améliorées en ce sens. Cependant, cela reste un domaine peu exploré pour les parties de contrôle, les machines et robots externes, les liaisons avec les SCAO externes, et l'interface homme-machine. L'intérêt de l'apport des techniques IA dans les SCAO à ce sujet est évident : introduire un minimum de capacités de traitement symbolique dans les SCAO.

- L'interface homme-machine des SCAO modernes est axée sur la manipulation graphique interactive. Néanmoins, certaines parties interactives utilisent toujours des moyens d'entrées purement textuels, souvent par l'intermédiaire d'un langage dédié ou d'une interaction nécessitant de nombreuses communications entre l'utilisateur et la machine. La saisie de données et d'informations nécessaires au système s'avère alors longue et sujette à beaucoup d'erreurs. Une solution éventuelle à ce genre de problème peut être l'intégration de modules de communication en langue naturel, ou de nouveaux modèles de description des données (comme une vue graphique plus adaptée à la spécification de caractéristiques d'objets). Là encore, la puissance et la richesse des modèles développés dans le cadre des études en IA peut apporter beaucoup. La communication en langage naturel [Pitrat 85], utilisée à bon escient, peut devenir aujourd'hui une réalité, de même que le conseil à l'utilisateur. Certains aspects de l'interface homme-machine peuvent être simplifiés (moins de structures figées des commandes, menus s'adaptant automatiquement à la manière d'utiliser le système, généralisation des icônes, commandes interconnectées). Au cours de la conception, grâce aux techniques IA, les erreurs peuvent être détectées interactivement et une solution peut alors être proposée immédiatement à l'utilisateur en vue de leur correction.

II.2.3. Problématique de l'intégration de l'IA en CAO

Nous traitons ici des aspects techniques relatifs à l'intégration et à la coexistence d'une partie logicielle classique de CAO avec une partie logicielle construite à partir de techniques IA

II.2.3.1. Communication de base entre les outils intégrés

On peut voir le problème de l'intégration sous deux aspects différents : grâce à un nouveau modèle ou grâce à l'intégration d'outils différents.

Dans le premier cas, l'intégration sera faite sur la base d'un nouveau modèle pour lequel on reconstruira l'ensemble logiciel avec de nouvelles normes. Ceci revient en fait à respécifier et reconstruire différemment le SCAO en prévoyant l'intégration des techniques IA et algorithmiques dans un même logiciel.

Dans le deuxième cas, il s'agira de réussir à faire communiquer deux parties logicielles indépendantes. On ne modifie pas profondément les concepts fondamentaux des deux parties, mais on les adapte pour qu'elles utilisent les services des autres.

Dans les deux cas, les parties algorithmiques et IA devront se communiquer des informations qui seront à priori de natures différentes, ne serait ce que parce que les langages utilisés ou les modèles de représentation des données dans les deux parties logicielles sont différents.

Prenons par exemple le cas de l'utilisation d'une liste dynamique de données dans chacune des deux parties: l'une écrite en LISP et l'autre en C. Le langage C ne propose aucun système de gestion de liste automatique et le programmeur C doit spécifier une nouvelle structure de donnée par l'intermédiaire d'une spécification de nouveau type. Le langage LISP est quant à lui basé sur la manipulation de liste et rend transparent au programmeur la gestion interne des liste manipulées. De ce fait, le programmeur LISP ne connaît pas la structure de donnée des listes manipulées en LISP. On ne peut donc pas assurer à priori que la structure de donnée liste spécifié par le programmeur C soit compatible avec la liste gérée de manière automatique par LISP. Il y a donc lieu de penser qu'il faille convertir les données C en données compatibles LISP et inversement. Ceci est généralisable à tous les types de données manipulées dans les deux parties, mais sera d'autant plus important que les données à communiquer sont complexes.

D'autre part, chaque compilateur ou interpréteur de langage utilise son propre modèle pour l'appel d'un sous-programme. Un langage pourra par exemple empiler les paramètres d'appel d'un sous-programme dans l'ordre inverse de leur apparition à l'appel (cas de C), étant ainsi en contradiction avec l'ordre d'empilage de l'autre langage. On devra donc parfois traiter également ce genre de conflit entre langages.

Ce qui vient d'être dit est surtout valable dans le cas où chaque partie existe au sein d'un même programme. Dans le cas où les deux parties logicielles existent de manière indépendante, il faut trouver et fabriquer un mécanisme de communication entre les deux parties; ce système devant gérer la conversion des structures de données si nécessaire. Cette technique a le défaut de nécessiter un traitement et une conversion systématique des données transmises entre les parties IA et procédurales, mais ce traitement peut être réalisé de manière automatique par une couche logicielle spécialisée.

Nous retiendrons que la transmission de données de nature différente et le type des langages différents est un premier problème à résoudre lors de l'intégration de logiciels d'IA et de CAO.

II.2.3.2. Correspondance des modèles

Dans les systèmes de CAO actuels, le modèle le plus répandu et le plus important pour manipuler numériquement et visuellement les objets conçus en CAO est le modèle descriptif (vu surtout du point de vue géométrique). Les modèles filaires, surfaciques ou solides permettent une représentation visuelle adaptée au type de problème traité. Cependant ces modèles présentent certaines limitations: la prise en compte de modèles géométriques complexes est encore difficile (composition d'objets en particulier), ils ne tiennent pas compte de l'information structurelle et sémantique, ils ne savent pas traiter certains concepts géométriques essentiels (notion de tangence, position relative des objets). En bref, ils sont limités parce qu'ils ne contiennent aucune information sémantique.

Au contraire, le raisonnement apporté par des techniques IA implique la prise en compte d'aspects sémantiques concernant la topologie de l'objet, sa géométrie, ses caractéristiques technologiques et fonctionnelles, et ses relations avec les autres objets composant le produit final. Le modèle de représentation des connaissances choisi par le système intègre donc la prise en compte de toutes ces notions.

Pour ces raisons, l'intégration de l'IA et de la CAO dans un même ensemble logiciel nécessite un traitement pour la correspondance des données au niveau des modèles utilisés. Ici se pose le problème du choix d'intégration : on pourra choisir de redéfinir un modèle capable d'intégrer les données sémantiques, comme par exemple MCAO1 et MCAO2 [Gardan et Zachari 91], ou de faire communiquer des SCAO existant avec des outils IA adaptés, ceci visant à produire un environnement complet pour les problèmes de CAO [Trousse 89 90b, Neveu et al 90].

II.2.3.3. Choix d'architecture de l'ensemble intégré

Au niveau de l'architecture d'un système intégrant des logiciels de CAO avec des techniques IA, il existe deux critères important à prendre en compte : d'une part la modélisation des données ou la représentation des connaissances et d'autre part la technique de communication utilisée. Le choix de ces paramètres influencera l'architecture finale du système.

Si l'on choisit d'utiliser des logiciels déjà existant sans les modifier profondément, on aura intérêt à choisir une architecture où les parties IA et CAO sont indépendantes (séparation forte) et fabriquer des systèmes de communications spécifiques et adaptés aux domaine de la CAO, comme c'est le cas pour *ANAXAGORE* [Trousse 89]. Dans ce cas, le système se présente comme un ensemble d'outils spécifiques communiquant et contribuant ensemble à la conception du produit.

Par contre, si l'on choisit de modifier les systèmes de CAO existant pour fournir un modèle plus riche prenant en compte des aspects plus sémantiques des objets manipulés, on peut avoir intérêt à coupler plus fortement les parties IA et de CAO en les englobant dans un seul logiciel. Néanmoins, ce dernier choix ne contraint pas définitivement l'architecture du système et l'on pourra également envisager une architecture distribuée, comme c'est le cas pour MCAO2 [Gardan & Zachari 91].

D'autre part, la technique de communication des données choisie influencera plus fortement l'architecture du système. Si l'on ne veut pas envisager de mécanisme de communication particulier, il faudra se baser sur les possibilités d'interface entre les langages utilisés et leurs communications de base implémentées dans les compilateurs ou interpréteurs, et de ce fait choisir une architecture où les ensembles IA et CAO ne sont plus vraiment indépendants. Enfin, si l'on veut garder l'indépendance de chaque partie, il faudra élaborer un

mécanisme de communication indépendant des langages utilisés, et l'on aboutira à une architecture du système où les différents logiciels sont indépendants et se transmettent des informations suivant le même schéma.

II.2.3.4. Synthèse et conclusion

L'intégration de systèmes IA avec des systèmes CAO dépend de trois paramètres essentiels : la communication de base (données élémentaires et sous-programmes) entre langages, la différence entre les modèles manipulés en CAO et ceux manipulés en IA, et l'architecture choisie pour le système final.

L'aspect modèle de données ou représentation des connaissances est un paramètre très important pour obtenir une bonne intégration, mais n'interdit pas d'envisager divers types d'architectures. L'aspect communication de base, moins important du point de vue fonctionnel du système, influence cependant plus fortement l'architecture final du système et donc son implémentation. Un système de communication dédié indépendant des langages utilisés permettra d'envisager un plus grand nombre de possibilité pour l'architecture finale.

II.2.4. Quelques caractéristiques des langages spécialisés en IA

Ce qui distingue essentiellement les langages IA des langages algorithmiques, associés étroitement au modèle d'architecture de machine Von Neumann, c'est la possibilité de programmer de façon non impérative.

Nous signifions par là que le programmeur n'indique plus à l'ordinateur comment il doit faire le travail demandé, mais fournit plutôt une description du problème. Dans certains cas, PROLOG par exemple, un programme spécialisé (l'interpréteur ou le moteur d'inférence) utilise cette description et la traite selon son fonctionnement (logique du premier ordre). On peut aussi faire une distinction entre aspects macroscopiques, nous entendons par là les capacités à décrire formellement les problèmes (langages IA), et microscopiques, c'est à dire des capacités à décrire des informations de structure simple ou complexe (langages procéduraux).

On classifie couramment ces langages en trois familles [Slimani 86] :

- langages applicatifs ou fonctionnels,
- langages déclaratifs,
- langages orientés objets.

Les représentants les plus célèbres pour chaque famille, et aussi ceux auxquels nous nous intéresserons ici (du point de vue implantation du langage), sont :

- applicatifs ou fonctionnels : LISP (ici LE-LISP, VAXLISP, ML).
- déclaratifs : PROLOG (ici PROLOG/CNET, D-PROLOG, Turbo-PROLOG, PROLOG/P).
- orientés objets : SMALLTALK (ici SMALLTALK80, SMALLTALK/V).

quelques avantages des langages IA :

- adaptés à la manipulation symbolique et formelle.
- permettent une approche formelle des problèmes, donc proche de la description de ceux-ci (langages fonctionnels et applicatifs).
- capables de décrire des situations d'un niveau d'abstraction très élevé.
- souvent interprétés (facilite le prototypage).
- adaptés au prototypage.

Inconvénients classiquement cités :

- difficulté de lecture des programmes ; à cause de la manipulation mathématique et symbolique, ou de leurs caractéristiques syntaxiques et sémantiques.
- inadaptés pour de gros logiciels ; en fait un problème dû surtout aux machines cibles et aux implantations spécifiques.
- faiblesse en structures de données ; argument faux, Cf. les classes Collection, Stream, Magnitude de SMALLTALK, la richesse et la souplesse des listes LISP pour décrire n'importe quelle structure, les listes, arbres et foncteurs de PROLOG; car ces langages se chargent eux-même de la gestion de leurs structures de données.
- gros consommateurs de temps machine et d'espace mémoire. Ceci est de moins en moins vérifié, car la technologie des compilateurs et interpréteurs, pour ces langages, est maintenant au point, et les machines cibles sont de plus en plus performantes.
- objets de laboratoire peu adaptés à l'industrie ; c'est un problème de transfert de technologie, qui tend à disparaître vu le nombre sans cesse croissant d'implantations de ces langages, surtout pour le langage PROLOG.
- jeunesse de ces langages, et donc de la technologie de programmation adoptée ; argument discutable : LISP existe depuis presque 30 ans, SMALLTALK existe depuis 1972, PROLOG depuis 1971.

Au lecteur désireux de s'informer, nous conseillons pour ces langages la bibliographie suivante, sans prétention d'exhaustivité :

- LISP : [ueinnec 84a 84b, Winston & Horn 84, Steele Jr 84, Chailloux 85b]
- PROLOG : [Condillac 86, Delahaye 86, Giannesini & al 85, Mellish & Clocksin 81] PROLOG III : [Colmerauer 90].
- SMALLTALK : [Cox 86, Goldberg & Robson 83, SMALLTALK/V 87].

Nous pensons cependant que ces langages souffrent de deux défauts en ce qui concerne leur utilisation dans la réalisation de SCAO : ils ne sont pas adaptés à l'écriture d'algorithmes, cela vient de leur nature, et ils sont non typés (sauf les langages objets), afin de pouvoir décrire et utiliser dynamiquement l'information symbolique.

II.2.4.1. Déficiences pour l'écriture d'algorithmes

Depuis que l'ordinateur existe, les scientifiques et les industriels ont produit une grande quantité d'algorithmes spécialisés, codés en des langages procéduraux, sous-programmes ou procédures utilisés dans beaucoup de logiciels. Citons l'exemple des bibliothèques de programmes de calculs numériques ou statistiques existant pour le langage FORTRAN.

En fait, la programmation procédurale est bien adaptée à la statistique, aux calculs divers (numérique, matriciel). Si LISP, par le rajout d'instructions de programmation structurée au langage, ou SMALLTALK, par le contenu potentiellement algorithmique des méthodes associées aux messages, permettent de programmer d'une façon un peu impérative, PROLOG interdit très fortement ce type de programmation.

Or, les SCAO sont de gros consommateurs de calculs et de programmes impératifs, vu la nature des traitements à effectuer sur les projets. Il est donc dommage, de ce point de vue, que les SCAO ne puissent profiter de la puissance symbolique des langages IA ; la difficulté d'écriture d'algorithmes est donc un défaut, qui rend problématique l'utilisation de ces langages dans un SCAO, ou dans tout autre logiciel utilisant des techniques de programmation impérative.

II.2.4.2. Langages non typés

La puissance des langages IA est aussi une de leurs faiblesses, si l'on veut les utiliser de façon procédurale, ou pouvoir utiliser des parties procédurales externes. Les données existantes lors du déroulement des programmes utilisant ces langages sont très souvent dynamiques, c'est à dire fabriquées à l'exécution. D'autre part, leur capacité à traiter des informations à haut niveau d'abstraction et de façon formelle implique l'inexistence des types de données tels qu'on les rencontre dans les langages procéduraux.

Or, pour pouvoir écrire des algorithmes, il faut déterminer d'avance les données que l'on manipulera, ceci d'une façon statique. D'autre part , si l'on veut utiliser des composants logiciels écrits de façon procédurale, il faut savoir décrire les données de la même manière que dans les composants.

Certains ont tenté de faire coexister dans un seul langage des aspects IA et des aspects procéduraux. En particulier des fonctionnalités des langages objets, plus proches de la programmation procédurale que les langages fonctionnels et déclaratifs, ont été introduites dans des langages comme C, pour donner ObjectiveC [Cox 86] ou C++ [Stroustrup 86]. Le langage Turbo-PROLOG [TB-PROLOG 86], de la société BORLAND, est typé un peu à la manière PASCAL, mais d'une façon parfois décriée par les spécialistes.

Finalement, on voit que les langages utilisés dans les programmes d'intelligence artificielle ne sont pas prêts à être utilisés dans les ensembles de programmation procédurale que sont les SCAO. Revoir complètement la conception de ces systèmes, pour qu'ils soient réalisés principalement avec des langages IA, serait peut être une bonne solution. Mais, quelle que soit la décision prise par l'implanteur d'un nouveau type de SCAO, nous pensons que certains traitements de la CAO imposent la programmation impérative et qu'il serait dommage de perdre la technologie existante.

Il semble donc que la communication langage IA - langage procédural soit indispensable à l'évolution des SCAO vers une intégration des techniques d'intelligence artificielle.

C'est ce que cette étude tente d'éclaircir et de formaliser, à partir de l'existant des langages dans l'industrie logicielle, sans se préoccuper des problèmes de cohabitation entre techniques IA et procédurales.

II.2.5. Conclusion

L'interfaçage entre les langages IA et les langages procéduraux a donc plusieurs intérêts essentiels pour les SCAO :

- enrichir à moyen terme les SCAO, qui sont dénués de capacités de raisonnement et d'aide réelle à l'utilisateur, par des parties spécialisées en IA, par exemple pour : l'interface homme-machine, les contrôles de validité du projet, la conception automatique dans des cas simples. Ceci n'empêche pas pour autant de disposer d'environnements intégrés.
- permettre de réaliser des parties internes aux SCAO, capables de raisonnement, et qui utilisent le contexte algorithmique du SCAO.
- faciliter l'évolution des SCAO vers des systèmes capables d'aider le concepteur et dotés d'interfaces homme-machines évoluées.

L'intégration IA - SCAO pose néanmoins des problèmes de modélisation des données et de communication de base entre les diverses parties logicielles. Elle peut être vue comme une redéfinition ou adaptation des modèles de base en CAO, conduisant à une modification importante des logiciels de CAO, ou comme la coexistence de logiciels spécialisés capables de communiquer entre eux grâce à des mécanismes de communication indépendant de l'implantation des logiciels

Nous avons dans ce chapitre étudié plus particulièrement les mécanismes de communication de base fournis de manière standard par les langages procéduraux et les langages IA., et également étudié plusieurs modèles de communication de base entre langages.

II.3. Communication entre partie IA et partie algorithmique

Nous discutons ici des choix possibles, au sein du SCAO, de communications de données entre une partie "intelligente" et une partie algorithmique.

II.3.1. Quelques exemples de communication nécessaire

Voici quelques cas où l'introduction d'une partie écrite en langage IA nécessite une communication avec une partie algorithmique.

1 - Supposons que l'on ait réalisé une partie du type système expert en conception dans le SCAO. Au cours d'une prise de décision, ou du déroulement du raisonnement, cette partie peut avoir besoin, de la part de l'utilisateur, pour éviter un choix au hasard, ou parce qu'il lui manque des informations, d'une valeur représentant une position dans l'espace. Cette position, couple (x, y) ou triplet (x, y, z) de coordonnées, peut être fournie de manière alphanumérique au clavier. L'utilisateur devra réfléchir aux valeurs à fournir, avec une marge d'erreur certaine, alors que cette position peut être précisée graphiquement par un moyen interactif rapide et aisé de saisie de position à l'écran. La solution utilisant ce dernier moyen pour entrer cette information est plus élégante, plus fiable (référence visuelle), et plus naturelle pour l'utilisateur.

Il faut qu'au moment où le processus de raisonnement décide de demander une position au concepteur, on communique avec l'interface graphique interactive : on déclenche alors une saisie interactive de position et celle-ci est renvoyée au système IA

2 - Certains aspects d'un domaine de CAO particulier, dans des cas simples, voire élémentaires, permettent une conception automatique et déterministe de sous-ensembles de l'objet conçu [Begg 84, Giambasi & al 83, Lebahar 84].

Une partie "intelligente" peut utiliser avantageusement ces potentialités, écrites en langage procédural. Il faut alors établir un protocole de communication entre la partie IA, qui adopte un format particulier de données (base de connaissances par exemple), gérées dynamiquement, et la partie algorithmique manipulant des structures bien connues, et de façon principalement statique. C'est l'un des aspects les plus importants et difficile de l'interfaçage.

3 - Supposons enfin que nous ayons réalisé une interface homme-machine permettant le dialogue en langage naturel, grâce aux techniques usuelles de ce domaine [Pitrat 85]. Chaque phrase comprise par cette interface va déclencher, comme pour un langage de commande usuel, une ou plusieurs "actions" standards du système. Il faudra donc, à un moment donné de l'exécution des programmes, communiquer avec la partie algorithmique existante pour cette action ou cette commande du SCAO.

Notons dès à présent que les exemples 1 et 3 font intervenir des aspects "système", au niveau microscopique des langages, et sont de ce fait relativement simples à réaliser, tandis que l'exemple 2 montre que la communication peut être moins simple dans certains cas, vu l'aspect dynamique des données en IA et leur représentation interne différente des représentations utilisées en algorithmique. Nous reviendrons sur ce point par la suite.

II.3.2. Quelles parties faire communiquer

On peut imaginer faire des interfaces avec beaucoup de parties pré-existantes dans le SCAO. Plutôt que de tenter une énumération exhaustive, nous regroupons ici les parties procédurales à interfacer pour deux catégories type des techniques d'intelligence artificielle.

II.3.2.1. Cas de résolutions de problèmes

Ce qui nous vient à l'esprit quand nous voulons insérer l'IA dans un SCAO, c'est qu'il faut utiliser la connaissance du domaine des experts. On peut prétendre que cette connaissance existe, mais de manière peu adaptée à l'IA dans les bases de données spécialisées du projet en cours, des informations techniques du domaine, des projets antérieurs. Cette prise de position entraînera des interfaçages complexes.

Nous pensons néanmoins qu'il vaut mieux revoir la conception de ces bases pour qu'elles soient adaptées à la fois à la CAO et l'IA [Cholvy & Foisseau 85].

Une autre tentative d'insertion peut être l'utilisation des algorithmes et/ou sous-programmes du SCAO pour réaliser la conception automatiquement [Jonckers 85]. On peut alors concevoir un système expert de manipulation de ces sous-ensembles procéduraux dédiés au domaine d'application du SCAO. On devra dans ce cas réaliser des interfaces avec les banques d'algorithmes ou autres sous-programmes.

Si enfin nous voulons construire des systèmes de vérification et de contrôle de la conception, il faudra interfacer avec les parties spécialisées du domaine (technologie, fabrication, simulation), qui elles sont écrites de façon impérative.

II.3.2.2. Cas de l'interface homme-machine

Nous pouvons avoir besoin de communiquer avec la partie graphique interactive, par exemple pour :

- Déclencher des actions du système (interface avec les logiciels concernant le langage de commande).
- Créer une aide interactive, qui contrôle et corrige les erreurs minimes (communication avec l'ensemble du système, ou les parties concernant le contrôle).
- Aider un utilisateur débutant en guidant ses actions par des propositions adaptées à la situation d'utilisation, issues de l'expérience d'un utilisateur expérimenté. Il faudra ici encore interfacer avec l'ensemble des logiciels du système.

L'interface homme-machine des SCAO est sans doute l'un des domaines les plus prometteurs, pour l'insertion des techniques IA. Après une évolution vers des interfaces totalement intégrées, on assiste maintenant à l'enrichissement par des aides à l'utilisation, par des descriptions schématiques et par diagrammes de l'évolution des projets; comme c'est déjà le cas pour les ateliers de génie logiciel interactifs. L'insertion de modules IA dans les interfaces permettra de créer des logiciels conviviaux, qui possèdent une certaine "connaissance" de leur contexte immédiat d'utilisation, et qui décrivent mieux les liens existant entre les différentes parties du système.

II.3.3. Interface partie IA - partie graphique

Nous avons choisi comme partie spécialisée du SCAO à interfacer avec les langages IA toute la partie graphique et ce pour plusieurs raisons :

A - La spécialité de la société METADESIGN est le logiciel de base en graphique respectant la norme GKS [GKS 3.0 88, GKS 4.0 88].

Celle-ci est, depuis 1985, un standard international ISO, et METADESIGN, par ses implantations sur micro-ordinateurs de cette norme, voulait promouvoir sa diffusion dans le milieu des utilisateurs graphiques, et montrer ses avantages en matière de portabilité. La société a d'ailleurs développé un logiciel de conception assistée en électronique, "METADESIGN-CAE", à partir d'une de ses implantations de la norme GKS [GKS 3.0 88, CAE 4.0 88] pour ce qui concerne les parties graphiques de ce logiciel.

Il semblait donc naturel et dans l'esprit des activités de la société, pour offrir aux programmeurs en IA sur micro-ordinateurs des capacités graphiques interactives et portables, de réaliser des interfaces entre des langages IA et les bibliothèques GKS de METADESIGN. Notons que ces interfaces ne sont pas prévues par le comité de normalisation graphique ni ISO, ni AFNOR, car, comme nous le verrons plus loin, beaucoup de problèmes conceptuels se posent, et ceci reste encore une partie immature de la technique, où des travaux de recherche sont encore nécessaires.

B - Le second point que permet d'étudier l'interfaçage avec les parties graphiques est l'aspect purement système. Nous prendrons ici un exemple simple pour expliciter cet aspect fondamental, qui met en jeu des techniques de logiciel de base élémentaire.

Soit un programme écrit en LISP ou PROLOG. Dans un programme, manipulant essentiellement de l'information symbolique, nous avons besoin d'un procédé de calcul des solutions d'une équation du 3ème degré :

$ax3 + bx2 + cx + d = 0.$

Ni LISP ni PROLOG n'incorporent, dans leur implantation, de fonction ou de prédicat, réalisant cette résolution (sauf pour PROLOG III). Nous pouvons écrire une portion de code réalisant cela, mais nous pouvons aussi utiliser les innombrables bibliothèques FORTRAN existantes, dont nous supposerons ici qu'une, au moins, nous procurera une routine permettant de calculer cela.

On trouve une subroutine "DEGRE3", dans un bibliothèque commercialisée, faisant exactement ce que nous désirons : calculer les racines de l'équation. Cette procédure s'écrit :

DEGRE3(A,B,C,D,X1,X2,X3,TOUTES_REELLES).

où :

A, B, C, D, sont les coefficients en entrée.

X1, X2, X3, sont les racines trouvées si elles existent.

TOUTES_REELLES est une valeur booléenne indiquant si les racines ont pu être calculées dans l'ensemble des réels.

LISP fournit dans certaines implantations une solution pour appeler cette procédure : l'appel de fonction externe en LE-LISP [Chailloux 85b] ou étrangère ("alien") en COMMON-LISP [Steele Jr 84, VAXLISP 84]. Il suffit alors de décrire celle-ci (nom et arguments...) avant qu'elle ne soit effectivement utilisée.

De même, avec PROLOG, grâce à la technique des prédicats évaluables en PROLOG/CNET [Barbeyre & al 83], ou globaux et externes en Turbo-PROLOG [TB-PROLOG 86], ou encore étrangers ("alien") en D-PROLOG [D-PROLOG 85a] et en PROLOG/P [PROLOG/P 85], on peut utiliser des prédicats écrits dans un autre langage.

Par exemple, on peut déclarer en Turbo-PROLOG :

DEGRE3 (real, real, real, real, real, real, real, integer)- (i,i,i,i,o,o,o,o) language PASCAL

- les quatres premiers réels déclarés représentent A, B, C, D
- les trois réels suivants les solutions trouvées
- l'entier représente le booléen TOUTES_REELLES
- on précise le "mode de passage" des paramètres (flow pattern), 4 en entrée et 4 en sortie.
- on indique enfin le langage externe utilisé, ici PASCAL.

Alors commencent les problèmes, dont voici les plus communs :

- 1) existence effective d'une communication avec un langage externe
- 2) compatibilité du codage de données entre les deux langages
- 3) ordre de passage des paramètres compatible avec le langage externe

- 4) sauvegarde et restitution des contextes des 2 langages
- 5) non existence de connexion système entre les 2 langages

Nous étudierons en détail ces problèmes d'implantation de langage, au cours de la section II.4, dans laquelle nous décrivons les aspects système de l'interfaçage entre langages. Retenons pour le moment que les constructeurs et implanteurs de langages, s'ils prévoient des mécanismes de communication inter-langage, ou plus précisément incorporent la possibilité de faire de la programmation multi-langages (mixed language programming) [MS-C 87], ne s'attachent pas souvent à prévoir une bonne incorporation de ces mécanismes au niveau système d'exploitation et machine cible, à moins que le système d'exploitation visé soit très rigoureux au niveau de la communication inter processus ou inter programmes (cas de VMS, MULTICS).

Un système ouvert et peu rigoureux comme MS-DOS est caractéristique des problèmes que l'on rencontre dans l'interfaçage logiciel : pour réaliser cette opération entre langages de constructeurs différents, il faut fabriquer complètement la voie de communication, ce qui implique quasiment une modification du système d'exploitation.

Les bibliothèques graphiques, par exemple GKS [GKS 85], ont l'avantage de mettre en évidence ces problèmes :

Multiplicité des types de données (compatibilité de codage) :

- entiers(SET_POLYLINE_INDEX(I)).
- énumérés
  (SET_FILL_AREA_HATCH_STYLE(hollow,hatch,solid,pattern).
- réels(SET_LINEWIDTH_SCALE_FACTOR(ep)).
- chaînes de caractères (TEXT ("texte")).
- typesstructurés(SET_CHARACTER_UP_VECTOR(point)).
- structures complexes
  (ESCAPE(no, enreg_entrée, enreg_sortie)).
  (POLYLINE (nb-point, liste_point)).

Primitives très paramétrées (ordre de passage des paramètres) :

- SET_POLYLINE_REPRESENTATION
  (poste_de_travail, indice, type_de_trait, coefficient_épaisseur, indice_couleur).

Si les paramètres ne sont pas passés dans le bon ordre pour la bibliothèque graphique, la primitive ne fonctionnera pas (car les codages des types énumérés, entier, réel, dépendent du langage et de la machine cible).

Communication à double sens (codage des données, ordre de passage des paramètres, sauvegarde et restitution de contexte) :

```
REQUEST_LOCATOR
    (poste_travail,numero_releveur                    --en entrée
    état_de_retour, transformation_choisie, position  --en sortie)
```

Bien sûr, d'autres parties logicielles des SCAO mettent en évidence ces problèmes, mais la partie graphique interactive, de par le nombre de primitives nécessaires, de par ses fonctionnalités et de par l'interaction avec l'utilisateur, fait surgir les problèmes très vite, si l'on doit en rencontrer.

C - La partie graphique interactive joue un rôle essentiel dans les SCAO. Toutes les communications entre l'utilisateur et la machine qu'elles soient directes (commandes, saisies) ou indirectes (lecture de schémas, visualisation) se font grâce à cette partie. La simulation de l'objet conçu par sa représentation à l'écran, par l'animation de ses constituants (en mécanique), et toutes les représentations de fonctionnement confortables (simulation thermique, contraintes de structure, simulation analogique) se font plus naturellement de façon graphique.

La partie graphique du SCAO, au niveau de base (bibliothèque de primitives), ou au niveau de l'interaction (menus, commandes...) est le meilleur élément de compréhension et de manipulation de la conception [Begg 84, Lebahar 84]. Il est donc normal de penser qu'un interfaçage avec ces parties sera nécessaire.

A un niveau de programmation plus descriptif de l'information (ou plus abstrait), par exemple en ce qui concerne la mémorisation des images CAO en mémoire, par des techniques de segmentation [Newman & Sproull 81], de structures [PHIGS 88], la partie graphique permet d'aborder un aspect plus important et ardu de l'interfaçage : la communication et la manipulation d'objets créés dynamiquement par les programmes.

En effet, on devra sans doute transmettre des données, dont on ne connaît pas la structure, au langage impératif, et il faut mettre en place un mécanisme capable de "décrypter" la structure de l'objet et de la modifier, pour qu'elle soit adaptée, par exemple en morceaux déterminés [Rueher & al 85] au langage procédural récepteur. C'est dans ce cas que l'interfaçage logiciel devient un problème beaucoup plus ardu, car, en plus des problèmes de logiciel de base, on doit concevoir des mécanismes de transfert de données beaucoup plus sophistiqués.

Enfin, une dernière raison, qui n'est pas des moindres, est que dans un logiciel utilisant des technique IA en CAO, par exemple un système expert, la modélisation de l'objet dans l'espace, et sa représentation à l'écran, constituent une part importante de l'expertise du

concepteur. Les objets conçus par ordinateur sont simulés visuellement et l'expert se réfère directement aux dessins et schémas comme image mentale. Etre capable de faire "converser 2 langages entre eux", le langage IA avec la partie graphique écrite en langage procédural, au sujet d'objets géométriques et de leurs structures informatiques associées, c'est déjà savoir transmettre l'image mentale brute de l'objet conçu [Begg 84].

Nous avons choisi d'étudier l'aspect communication de base entre langages IA et langages CAO. Des travaux ont déjà été réalisés, dans un autre contexte, qui visent l'interfaçage soit "système" [Hubner & Markov 86], soit "symbolique" [Rueher & al 85, Kanada & Kawai 91], ce dernier devant de toute façon régler les problèmes de l'interfaçage système, pour communiquer avec logiciel graphique.

Nous avons inclus en annexe une description succincte du système graphique GKS. Les parties qui suivent ne sont pas dépendantes de la compréhension de GKS.

II.3.4. Conclusion

De nombreuses raisons incitent à réaliser une interface entre les langages IA et les langages procéduraux. En CAO, les causes principales poussant à réaliser cet interfaçage sont les suivantes :

- les déficiences des langages IA, par leur jeunesse, ou leur inadaptation aux problèmes algorithmiques, pour créer certaines parties nécessaires aux SCAO (graphique interactif, bases de données CAO, algorithmes spécialisés...), ce qui provoque la création d'interfaçages en majorité incapables de manipuler l'information symbolique. C'est le cas d'une interface système PROLOG-GKS, comme par exemple [Hubner & Markov 86].
  Plus simplement, pour ne pas refaire ce qui a déjà été fait pour les SCAO avec des langages non conçus pour cela, rendre capable les langages IA et procéduraux de communiquer des données entre eux, et de les traiter.
- le fait que si l'on veut créer, au sein du SCAO, des modules intelligents écrits avec des langages IA, il faudra faire communiquer ces modules avec les autres parties du système, ceci d'une manière contrôlée et avec des capacités de traitement symbolique. Dans ce cas, on sera amené à faire des interfaces beaucoup plus sophistiquées qui devront être capables de manipuler l'information symbolique du contexte IA pour en extraire l'information statique et impérative, et transmettre celle-ci d'une façon correcte aux parties algorithmiques. Inversement, dans certains cas, ce genre d'interface devra préparer l'information impérative de manière à ce qu'elle puisse être utilisée par la partie IA. C'est par exemple le cas d'interfaces comme PATK2D [Rueher & al 85].

Toutes ces remarques sont valables pour les logiciels basés sur des techniques de programmation impérative, logiciels auxquels on voudrait rajouter des parties dites "intelligentes", ou simplement qui utilisent des langages spécialisés en IA. Le premier problème à résoudre

est le problème de la communication entre le langage IA et le langage procédural, à la fois du point de vue logiciel de base, et aussi pour gérer correctement les informations, quand on passe d'un contexte à l'autre.

On peut en tirer le schéma de communication simplifié suivant entre un langage spécialisé en intelligence artificielle (LIA) et un langage algorithmique (LA) :

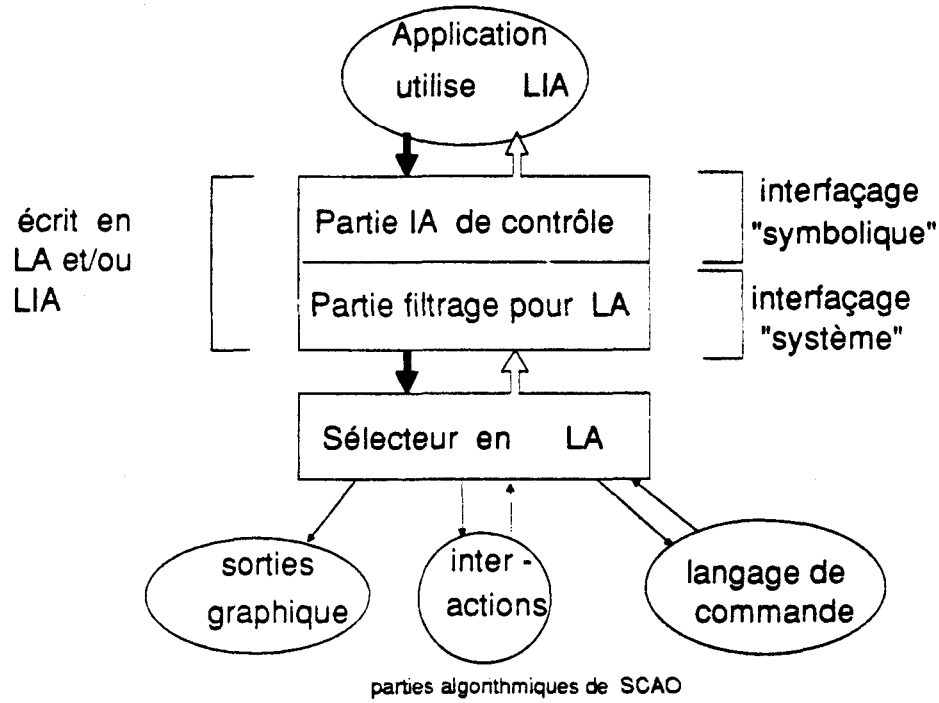


Schéma de communication simplifié entre LIA et LA pour un SCAO

Note : l'objectif de la société étant d'incorporer des parties IA aux SCAO, par exemple en fabriquant un système expert ou une interface homme-machine, nous nous sommes dans un premier temps attachés à étudier l'interfaçage "système". Nous parlerons cependant de l'interfaçage dit "symbolique", de façon partielle. Nous pensons que chaque interface, du point de vue de sa structure et de son implantation, va dépendre de l'application dans laquelle elle sera insérée ; pour aborder ces problèmes de manière approfondie, il faut réaliser des études spécifiques.

Dans les paragraphes qui suivent, nous abordons de façon technique l'interfaçage logiciel, en tant que communication inter-langage, et traitons de nos travaux en matière d'interfaçage LIA-LA de bas niveau. Nous avons essayé un interfaçage avec des bibliothèques graphiques interactives, celles-ci étant d'une richesse algorithmique suffisante pour mettre rapidement en évidence les problèmes éventuels.

II.4. Interfaçage logiciel système

Dans cette partie, nous décrivons les éléments, conceptuels et logiciels, du problème de l'interfaçage logiciel, du point de vue logiciel de base. Abordant la question en dehors de tout contexte de langage, ou de système d'exploitation particuliers, nous introduisons les concepts et les problèmes couramment rencontrés. Puis nous introduisons une classification, que nous prendrons comme référence, pour chacun des cas que nous avons étudiés. Enfin, l'interfaçage logiciel étant couramment rencontré dans les programmes impératifs, nous en donnons deux exemples sous système d'exploitation MS-DOS, au sujet de "binding" GKS (l'interface langage pour utiliser une bibliothèque GKS, Cf Annexe II-0).

II.4.1. Description générale de l'interfaçage logiciel

II.4.1.1. Introduction

Le but d'une interface logicielle entre deux parties programmées séparément, est de permettre, de façon générale, la communication entre ces deux parties. Dans le cas que nous étudions ici, interfaçage que nous avons appelé "**système**", il s'agit pour l'un des modules de transférer des données, simples ou complexes, de nature statique, c'est à dire de structure figée -- ex : entier, réel, chaîne de caractères, tableau, enregistrements sans champ variable...-- vers l'autre module, qui les traitera selon ses fonctionnalités, et retournera un résultat éventuel au module appelant. On parle aussi souvent de "connectique logicielle".

Ceci existe de manière classique dans les langages algorithmiques de haut niveau, mais pour des modules écrits avec le même langage ; grâce à la notion de *module* avec Modula2 [Wirth 82], de *paquetage* en ADA [Barnes 88, Le Verrand 82], de *unit* de certains PASCAL [MS-PASCAL 85a], qui permettent d'écrire deux parties séparées, et de les faire communiquer au travers de leur interface. Grâce à ce principe, on peut créer des parties spécialisées regroupant plusieurs modules (Cf I.7), et les parties concernées peuvent être implantées, après description des interfaces, par des équipes indépendantes. Notons que la transmission à double sens, i.e.. module A vers module B et inversement, est nécessaire pour réaliser des SCAO, à cause de l'architecture logicielle en graphe et imbriquée de ceux-ci.

Ce n'est pas la seule raison d'être des langages modulaires précités, loin s'en faut, mais c'est une capacité importante que de vieux langages comme FORTRAN IV permettent, d'une façon un peu détournée, par exemple par les subroutines externes à plusieurs entrées [Lamoitier 78].

Malheureusement, certaines parties n'utilisent pas le même langage, pour des raisons d'inadaptation du langage à la spécialité de cette partie, par exemple l'écriture d'un SGBD en FORTRAN, ou par le fait que celle-ci est déjà disponible dans un autre langage que celui choisi pour le noyau du SCAO, par exemple un noyau graphique en FORTRAN et un noyau en C. Dans ce cas, il n'existe pas de moyen standard pour faire coexister les deux parties. Si une

telle communication est prévue par les deux implantations de langage sur le système d'exploitation cible, on peut parfois ne pas avoir à se préoccuper de cette question, sinon il faudra réaliser la partie communicante entre les deux langages : l'interface logicielle.

Voyons maintenant les éléments de ce genre d'interface.

II.4.1.2. Description

Nous faisons intervenir ici deux parties programmées et compilées séparément, P1 et P2, écrites respectivement en langage L1 et L2, et la partie communicante inter-langage I. La figure II.9 donne une idée du fonctionnement logiciel du traitement, et du parcours des données au cours d'une communication.

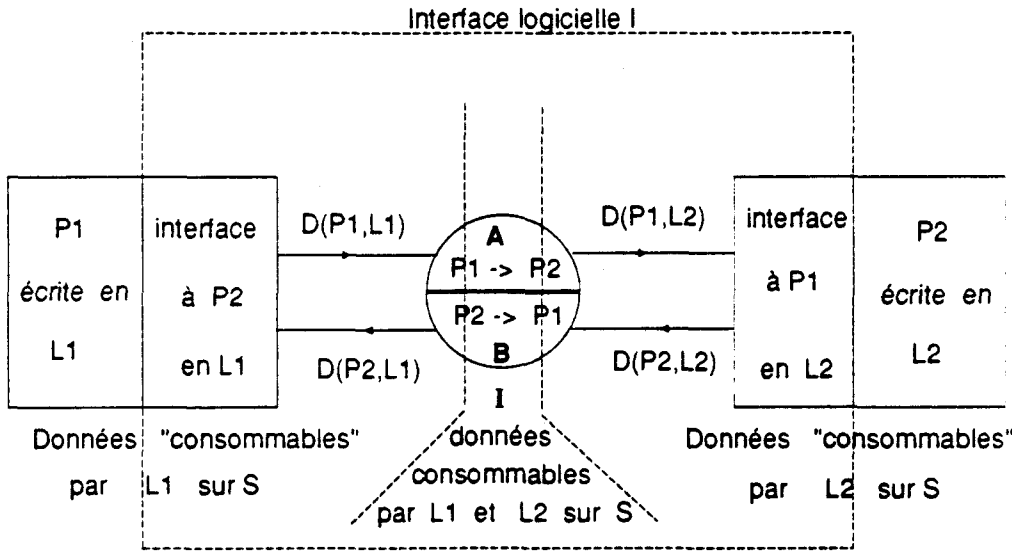


Schéma conceptuel de l'interfaçage logiciel

Fig II-2

Notons que pour l'interfaçage système, les parties A et B de I ne manipulent que des données de structure connue, et que les fonctionnalités de P2 appelées par P1, et réciproquement, sont connues de P1, P2, I, et sont décrites dans les interfaces de P1 et P2.

Certaines données, en cours de transformation et adaptation, ou quand elles sont de nature simple (caractère, entier, réel), ou de nature directement existante sur le système d'exploitation, transitent directement de P1 vers P2, c'est à dire sans que leur structure ou leur nature soit modifiées.

Si nous adaptons ce schéma à notre étude, nous obtenons un schéma similaire entre langages LIA et LA :

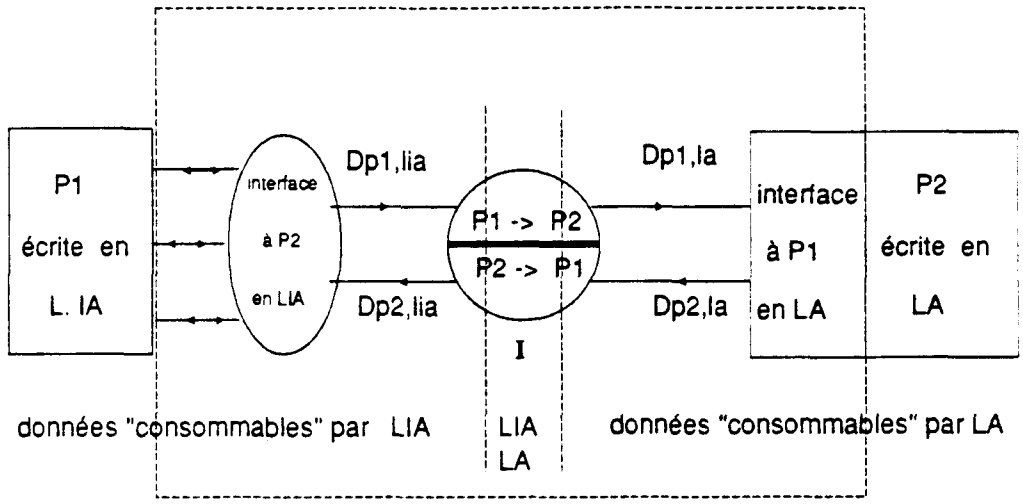


Schéma de l'interfaçage LIA : LA

Fig II-3

Dans ce cas, nous avons la présence d'une partie logicielle écrite dans le LIA et spécialisée dans la communication avec d'autres langages : c'est l'interface de P1. Citons quelques exemples de mécanismes de connexion, intégrés au langage d'IA:

- prédicats "évaluables" (PROLOG 2),
  prédicats globaux "externes" (Turbo PROLOG),
  prédicats "alien" (D-PROLOG, PROLOG/P) dans le cas de PROLOG.
- fonctions externes "CALL EXTERN" (LE-LISP),
  "Alien functions" (VAXLISP, COMMON-LISP) dans le cas de LISP.
- objets et méthodes "utilisateur" (SMALLTALK/V),
  "externe" (SMALLTALK-80) dans le cas de SMALLTALK.

Ce schéma ne préjuge en rien de la manière dont est implantée l'interface logicielle. Celle-ci peut être écrite avec un seul langage, ou être une combinaison de parties écrite avec plusieurs langages.

Notre exposé suppose aussi qu'il existe une implantation des deux langages sur le système d'exploitation cible. Voyons maintenant les architectures logicielles les plus courantes pour l'implantation d'une interface logicielle.

II.4.2. Architectures courantes des interfaces logicielles

Nous les avons classées en deux catégories :

- les interfaces dites "**séparées**", qui ne nécessitent pas une communication directe entre les deux parties
- les interfaces "**directes**", qui supposent l'existence et le fonctionnement des deux parties en même temps, grâce à une liaison disponible avec le système d'exploitation et/ou avec l'implantation des langages utilisée.

II.4.2.1. interfaces séparées

Elles sont constituées de trois parties (Cf Fig II- 4) :

- la partie IP2, de P1, qui contient l'interface aux ressources de P2 (des noms et des déclarations de ressources existantes en P2), et des appels à ces ressources. Il n'y a pas forcément coïncidence des noms de ressources, entre ceux déclarés dans l'interface à P2 et ceux utilisés par P1.
- La partie IP1, de P2, qui joue un rôle symétrique à IP2.
- l'interface I, constituée d'une mémoire de stockage des informations (noms des ressources appelées + données), divisée en deux parties : la zone Z1, d'écriture par P1, et de lecture par P2 ; la zone Z2, d'écriture par P2, et de lecture par P1. De plus, IP1 et IP2 contiennent l'implantation logicielle des routines utilisées par P1 pour écrire dans Z1, et de celles utilisées par P2 pour écrire dans Z2.
  Notons qu'ici I est partagée par P1 et P2, ou plutôt qu'une partie de I, dédiée à P1, existe en P2, et réciproquement pour P2.

Remarque : Ce sont les réalisateurs de P1 et P2 qui s'entendent sur le codage et la description des données en Z1 et Z2. Ainsi, ils peuvent réaliser, chacun de leur côté, une partie de l'interface. Ici, la description de Z1 et Z2, et la lecture-écriture de ces données en ces parties, estessentielle.

Le schéma initial est modifié en :

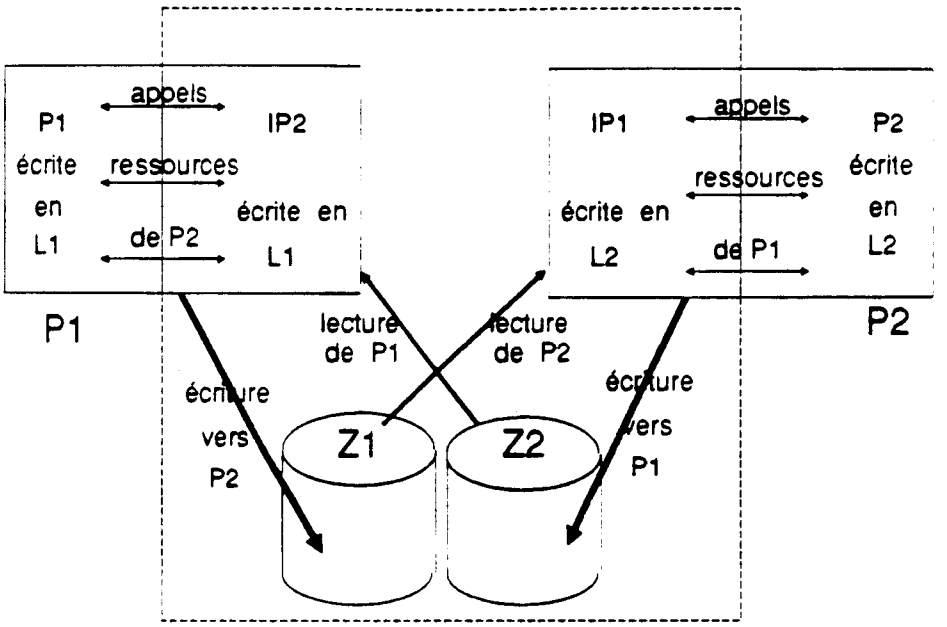


Schéma d'une interface logicielle séparée

Fig II-4

Il nous faut faire plusieurs remarques sur ce modèle :

- Si le système S est multitâches, alors P1 et P2 peuvent exister et fonctionner ensemble ou séparément.
  Dans le premier cas, il faudra prévoir un modèle classique de communication entre processus pour être sûr que P1 comme P2 lisent et consomment effectivement les données qui leurs sont adressées.
  Dans l'autre cas le déroulement de la communication est plus simple, mais pas toujours facile à mettre en oeuvre : écriture des données par P1 (P2) dans Z1 (Z2), consommation par P2 (P1) de ces données, traitement par P2 (P1), écriture éventuelle de résultats par P2 (P1) dans Z2 (Z1), et récupération de ces données par P1 (P2); le tout d'une façon strictement séquentielle. Dans ce cas, l'une des parties, ici P2 à priori, est utilisée comme "esclave" de l'autre.

- Dans certains cas, Z1 et Z2 ne sont pas séparées, ce qui complique la gestion de lecture-écriture entre P1 et P2 (surtout si P1 et P2 sont des processus indépendants sur S), ainsi que la description des routines et données dans ces zones.

Ce genre d'interface est utilisée en CAO, pour les standards d'échanges de données (SET, IGES, EDIF, DXF, CGM [Arnold & Bono]), ou pour utiliser des périphériques graphiques (langage HPGL ...), comme des traceurs.

avantages :

- on peut faire communiquer, au niveau "système", toutes les parties logicielles que l'on veut, même si les implantations de L1 et L2 sont incompatibles, à condition que la gestion de Z1 et Z2 par P1 et P2 soit rigoureuse et bien conçue.
- tous les types de mémoire sont utilisables pour Z1 et Z2, mémoire vive ou de masse.

Inconvénients :

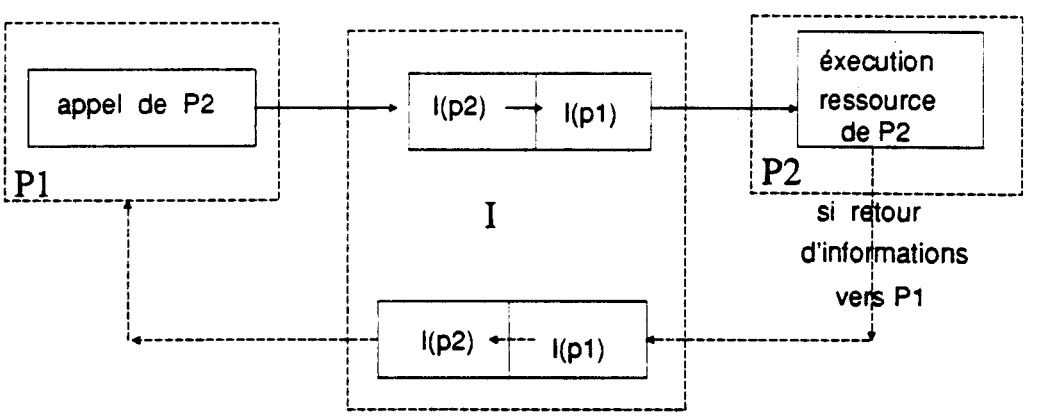
- lourdeur de la mise en place (rigidité et complexité de I),
- lenteur des communications (2 transferts mémoire Z, et 2 transferts langages)
- seulement adapté à une liaison P1 maître - P2 esclave, c'est à dire inadaptée à un modèle où des processus P1 et P2 coopèrent,
- aspect essentiellement séquentiel des communications.

C'est un **procédé de "bas de gamme" et de dernier recours**, surtout adapté au cas où les 2 parties logicielles sont complètement indépendantes, par exemple dans le cas de fichiers de tracés, écrits en langage HPGL; ou de données à communiquer entre systèmes de CAO.

II.4.2.2. Interfaces directes

Ce type d'interface caractérise celles qui utilisent les outils bruts, du genre commandes système, ou plus évoluées, du genre édition des liens, liens dynamiques..., du système d'exploitation cible.

Dans ce cas, les deux parties P1 et P2, et l'interface I, existent toutes trois dans un même ensemble logiciel (programme). Le parcours des informations, lors d'un appel de ressources de P2 par P1, se fera toujours selon un schéma similaire au suivant :



Chemin des données dans une interface directe

Fig II-5

P1 reprendra son déroulement quand P2 aura terminé celui de la ressource visée, et quand les parties de I auront aussi terminé leur transfert d'information.

Idéalement, P1, P2, et I, devraient faire partie du même programme, et d'ailleurs les constructeurs de software qui attachent de l'importance à la communication inter-langage, s'efforcent d'obtenir ce résultat (MICROSOFT, MERIDIAN, VAX). Mais nous nous sommes surtout intéressés au cas où les langages de P1 et P2 ne sont pas fournis par le même éditeur de logiciel. Nous avons pu, dès lors, faire un sous-classement des interfaces directes, en fonction de la capacité des procédés d'interfaçage à faire coexister simplement, dans un même programme, les trois parties précédentes. Voyons chacune des catégories ainsi mises en évidence.

Interfaces directes fausses

On utilise un mécanisme, fourni par le système d'exploitation cible, que l'on peut mettre en oeuvre "manuellement" (pour la création et la mise en fonction), pour faire communiquer P1 et P2. C'est par exemple le cas des "pipe" (ou tubes) d'Unix [Bourne 85, Rifflet 86], ou des Device Drivers de MS-DOS et Unix [Kernighan & Pike 86, Bourne 85, Rifflet 86, MS-DOS 84].

Les procédés de communication sont ici contrôlés par le système d'exploitation, et le programmeur doit faire appel aux ressources d'une manière protocolaire prévue par le système. Pour les pipes Unix par exemple (ou MS-DOS), on devra créer un fichier en sortie de P1 qui sera repris par P2.

Pour les "device drivers" (pilotes d'unité), il faudra d'abord les écrire (ceci donnera P2) d'une manière officielle (ex: Character et Block device driver en MS-DOS), puis y faire appel de la manière imposée par le système, qui n'est pas forcément très pratique. Notons aussi qu'avec les "tubes", sous MS-DOS, et sous UNIX en utilisant un fichier de commande, seul un lien unidirectionnel est envisageable, empêchant ainsi la communication P2 vers P1.

Finalement, pour pouvoir faire appel à ces fonctions, il faut que les implantations des langages (surtout pour la partie P1, car elle déclenche la communication), permettent d'utiliser ces ressources du système.

Un symptôme de l'inadéquation de ces procédés est, par exemple pour MS-DOS, la non utilisation, par les concepteurs de logiciels, du procédé fourni pour réaliser et utiliser des device drivers; les concepteurs préfèrent concevoir des programmes résidant en mémoire, et les déclencher par interruption système.

De plus, en y regardant de plus près, ces outils ne sont en fait que des interfaces séparées, mais obéissant à un protocole d'exploitation défini par le constructeur du système, ce qui rend les programmeurs encore moins libres, mais tout autant gênés, pour ce qui est de la conception et de la réalisation d'interfaces logicielles.

Voici le schéma d'interfaçage lié à ces outils :

- pour les PIPE (par fichier de commandes UNIX, et dans tous les cas MS-DOS) :

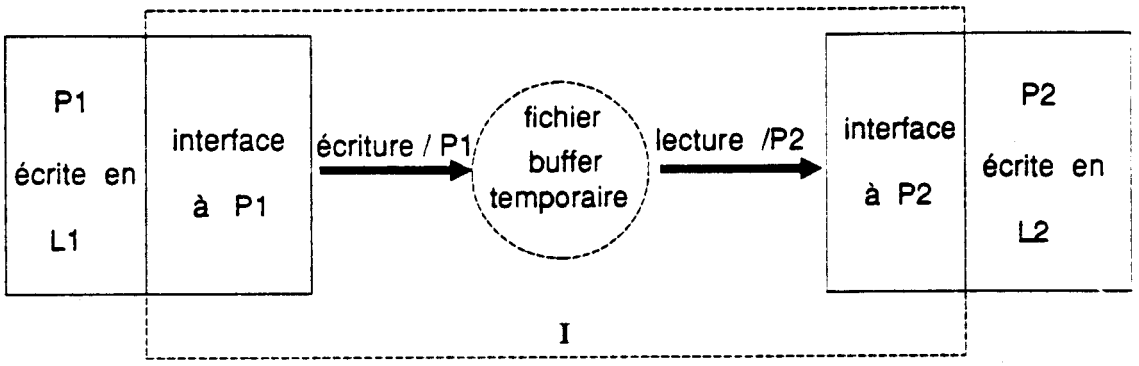


Fig II-6

- pour les device drivers :

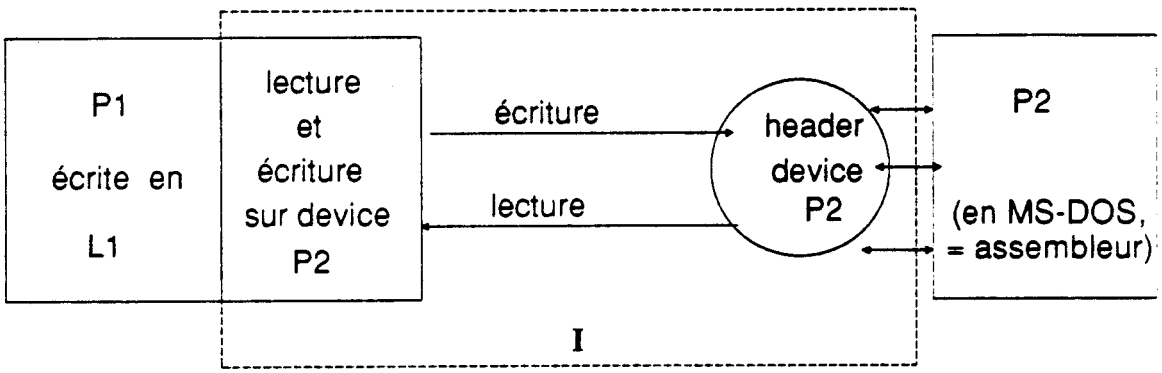


Fig II-7

Avantages :

- mécanismes intégrés au système
- indépendance de P1 et P2
- mêmes avantages que pour les interfaces séparées

Inconvénients :

- ceux des interfaces séparées

Interfaces directes intermédiaires

Il s'agit d'utiliser une possibilité offerte couramment par les systèmes d'exploitation : la lecture-écriture sur fichier par deux processus indépendants.

Les fichiers sont communément représentés en mémoire par des structures de données, dans lesquelles il existe un tampon contenant les données en cours de lecture ou d'écriture [Kernighan & Pike 86, MS-DOS 84].

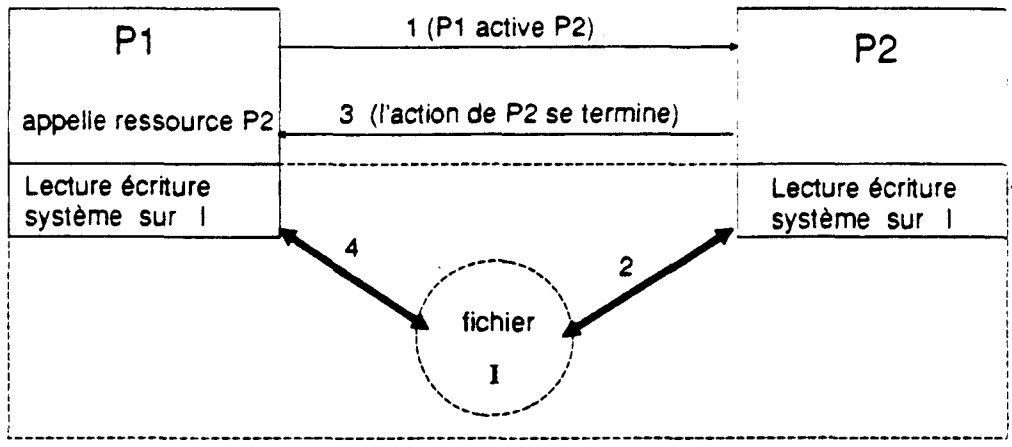
Ces structures, quand les fichiers sont utilisés, sont créées par le système d'exploitation et sont situées en mémoire vive. De ce fait, beaucoup de systèmes permettent de lire et d'écrire dans ces structures temporaires, comme s'il s'agissait d'un périphérique normal. Par extension, certains langages offrent aussi la possibilité de lire et d'écrire en mémoire, par utilisation de cette possibilité du système, dans une zone mémoire ou une structure de données quelconque (fonctions *read et write en FORTRAN, encode et decode en MS-PASCAL, read et write en C*).

Partant de cette hypothèse, si le système d'exploitation permet de déclencher, d'une manière quelconque, deux programmes (processus), en séquence, on peut réaliser le schéma de communication suivant :

On suppose que P2 fonctionne comme esclave de P1.

- P1 prépare son appel de ressource à P2 en écrivant des données dans le tampon
- P1 fait appel à P2 par un moyen quelconque
- P2 est activé, lit et décode le contenu du tampon
- P2 réalise l'action décrite par les données du tampon
- P2 écrit son résultat dans le tampon (action correctement réalisée, données en retour si nécessaire...)
- P2 se termine
- P1 redevient actif et lit le tampon en retour
- P1 continue son traitement.

Le schéma de principe de la communication est alors :



Communication dans une interface directe intermédiaire

Fig II-8

Le travail principal pour construire ce type d'interface consiste donc à :

- trouver le moyen de déclencher deux programmes ou processus en séquence, qui soient indépendants et chargés tous deux en mémoire. Par indépendants, nous entendons qui peuvent être créés indépendamment l'un de l'autre,
- vérifier que l'on peut lire ou écrire dans un fichier virtuel ou mémoire (file handle, stream handle),
- se mettre d'accord (les concepteurs et réalisateurs de P1 et P2) sur un langage de "description des appels de ressources et de données" commun à P1 et P2,
- s'accorder sur les ressources disponibles de P2.

avantages

- mécanisme de communication intégré au système, avec simplification, sûreté, contrôle automatique par le système.
- P1 et P2 sont indépendants, du point de vue langage comme du point de vue système.
- on peut utiliser un tampon en mémoire vive ou en mémoire de masse (fichier), il n'existera en fait qu'une différence d'efficacité.
- on peut transmettre de P1 à P2 n'importe quel type de donnée statique, voire dynamique, à condition que le langage de description le prévoit.

inconvénients

- il faudra, si les données transmises sont de structures complexes, prévoir dans P1 et P2 un codeur-décodeur de données intégré à chaque programme
- une lourdeur certaine de mise en place

- adapté seulement à une liaison P1 maître - P2 esclave
- aspect toujours séquentiel des communications (P2 termine sa tâche avant que P1 ne reprenne la sienne)
- nécessité d'un protocole commun de description des données.

On fera remarquer que ce type d'interface présente des similitudes avec le modèle séparé. Néanmoins, ce mécanisme est plus simple à mettre en place, et surtout utilise directement le système d'exploitation, grâce au fichier interface I, et aux appels disponibles dans les langages L1 et L2. C'est pour cela que nous l'avons classée parmi les interfaces directes. C'est ce type d'interface que nous avons mis en place pour les bindings de METADESIGN-GKS 4.0 (voir II.5)

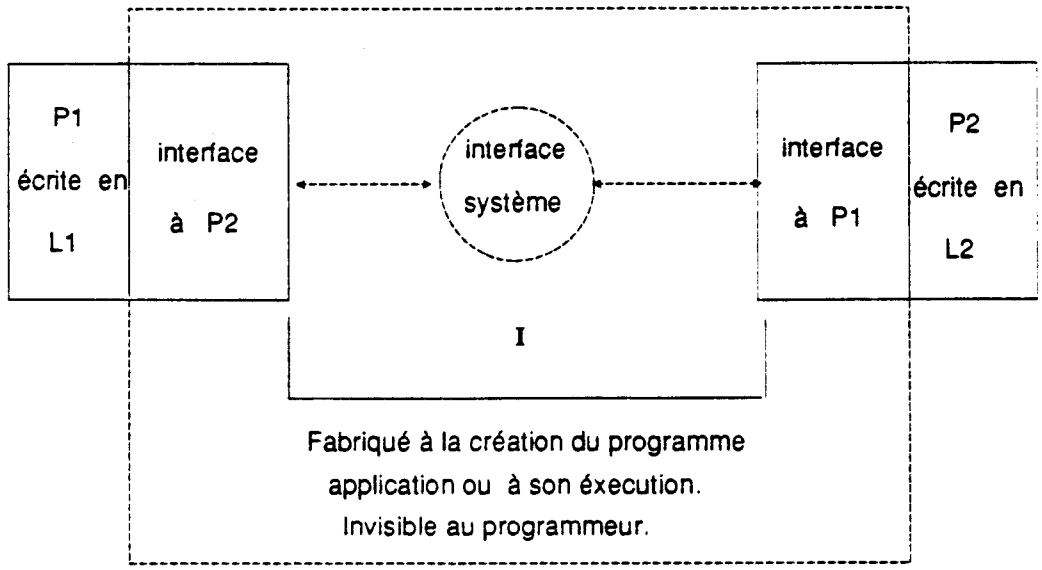
Interfaces directes vraies

Ce type d'interface constitue la voie "royale" de l'interfaçage, du moins à première vue. Les programmeurs de P1 et P2 n'ont pas à se préoccuper des communications, car cela est fait par les langages utilisés, et par le système d'exploitation, comme si les deux parties étaient écrites dans un même langage modulaire. Quel en est le principe :

- P1 est programmé indépendamment de P2, sauf pour ce qui concerne les appels de ressources à P2, qui sont réalisés de la manière imposée par l'interface à P2, incluse dans P1. Il en va de même pour P2.
- On effectue les compilations de P1 et P2 séparément. A ce moment, le créateur de l'application utilise un outil du système d'exploitation pour lier P1 et P2, par exemple un éditeur de liens, pendant leur déroulement.

C'est le principe privilégié par les constructeurs de systèmes et de langages pour que les concepteurs d'applications puissent créer des parties logicielles en langages différents : c'est le cas des langages MICROSOFT sous système MS-DOS, et des systèmes VAX VMS et MULTICS.

En voici le schéma de principe :



Communication dans une interface directe vraie

Fig II-9

Notons que dans ce cas le codage des données est essentiel : soit les compilateurs des langages L1 et L2 codent et transmettent ces données de la même manière et utilisent le même protocole d'appel de sous-programmes (langages sous VMS), soit on doit se livrer au jeu des correspondances entre types de données de P1 et de P2, et préciser le protocole d'appel utilisé (Turbo-PROLOG, langages MICROSOFT,) ; entre C et PASCAL par exemple, les paramètres de sous-programmes sont empilés en ordre inverse l'un de l'autre, en Turbo-PROLOG on doit dire quels paramètres seront utilisés en entrée, et lesquels en sortie.

Remarquons que si ce type d'interface existe communément pour les langages algorithmiques, il n'en est rien pour faire communiquer les langages IA avec les langages procéduraux, sauf pour les implantations récentes de ces langages, ou certaines implantations réalisées elles-mêmes avec les langages algorithmiques avec lesquels on veut communiquer (cas de DELPHIA PROLOG).

Avantages :

- communication intégrée au système et aux langages
- mise en place aisée (normalement)
- fonctionnement séquentiel ou partagé de P1 et P2 (suivant système d'exploitation et la réalisation des parties P1 et P2).
- liaison P1-P2, aussi bien maître-esclave que processus coopérants.

Inconvénients

- les créateurs de P1 et P2 doivent s'accorder sur les interfaces respectives à P1 et P2
- nécessité de permettre des déclarations inter-langages pour P1 et P2 (doit être fourni par les implanteurs des langages)
- correspondance des données pas toujours simple
- protocoles d'appels de sous programmes parfois incompatibles
- pas de contrôle réel sur les données transmises, il faut une correspondance exacte. Difficultés de mise au point à cause de ce problème

Même si ce type d'interface comporte certains inconvénients, elle est tout de même la plus prisée des programmeurs, car elle leur évite en principe de manipuler le système d'exploitation dans ses détails.

Dans le paragraphe suivant, nous précisons en détail quelques problèmes classiques des interfaces logicielles, surtout pour le cas des interfaces directes vraies.

II.4.3. Problèmes usuels de l'interfaçage logiciel système

En dehors des difficultés que l'on peut rencontrer quand on crée une interface logicielle du type séparé, direct faux ou direct intermédiaire, difficultés très souvent liées au système d'exploitation cible (par exemple en MS-DOS il faut tout créer à la main) et aux implantations des langages, nous essayons ici de préciser, de façon non exhaustive, les problèmes classiques que l'on rencontre lorsque l'on veut créer une interface directe vraie, au moment des appels de sous-programmes.

En général, ces problèmes interviennent lorsque les langages L1 et L2 ne proviennent pas du même constructeur. Certaines remarques seront valables pour les autres types d'interfaces, surtout en ce qui concerne le codage des données. Nous le préciserons à chaque fois que ce sera nécessaire.

Reprenons les cinq points problématiques cités au paragraphe II.2 .

II.4.3.1. Connexion logicielle inter-langages

Pour pouvoir espérer interfacer les deux langages, il faut qu'il existe, dans le cas d'une interface vraie, les possibilités d'une déclaration externe pour réaliser l'interface à P2 dans P1, et réciproquement si nécessaire. Par exemple, les prédicats externes, ou évaluables, en PROLOG, sont un bon exemple de cette connexion logicielle.

Nous avons cité les fonctions externes de LISP, et les prédicats externes ou évaluables de PROLOG, mais encore faut-il que ce mécanisme, ou quelque chose de similaire, existe pour l'implantation utilisée des langages interfacés.

II.4.3.2. Codage des données

Il faut que les deux langages se "comprennent" ; chacun d'eux code les entités de type élémentaire d'une façon qui lui est propre : par exemple, les réels peuvent être codés au format IEEE sur 4 bytes ou 8 bytes, ou encore sous un format particulier non standard, les entiers peuvent être codés sur 2 ou 4 bytes, voir plus suivant la machine-cible, les chaînes de caractères peuvent être du genre de celles codées par C (caractères ASCII suivis d'un caractère nul) ou de type ADA (longueur de la chaîne + caractères).

Si les mêmes données sont codées différemment en mémoire dans les deux langages, l'appel du sous programme externe conduira à des résultats imprévisibles, souvent un arrêt du système.

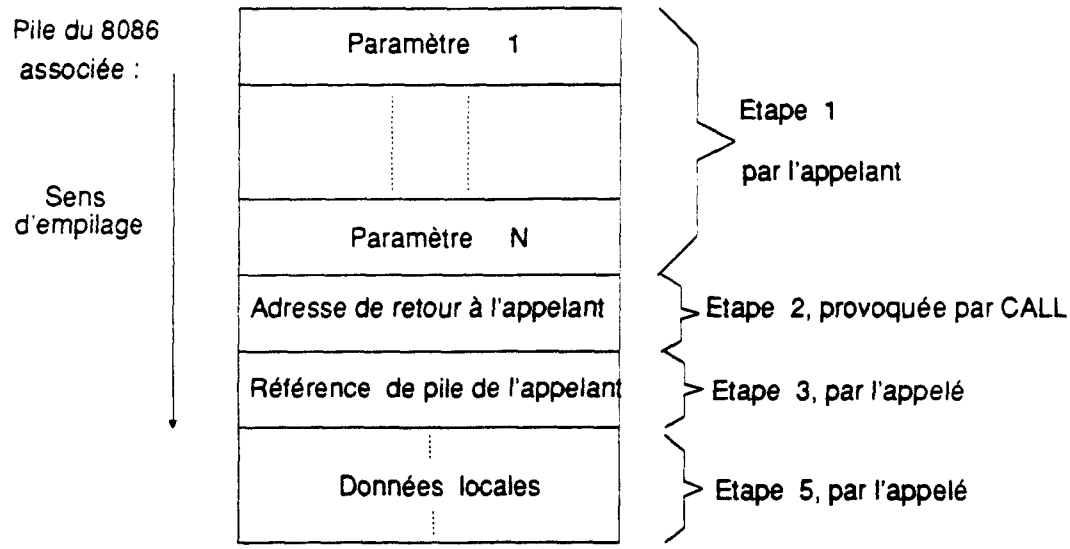
Ceci est valable pour des données de type simple, codées directement par les langages : entiers, réels, booléens, chaînes de caractères, adresses mémoires, pointeurs. Pour des données de structure plus complexe, par exemple des enregistrements ou des listes, il faudra s'assurer que les données, que chaque langage reçoit de l'autre, sont correctement codées, c'est à dire sont "assimilables" par le langage récepteur de l'information.

II.4.3.3. Passage et sauvegarde des contextes d'appel

Chaque appel de sous-programme avec paramètres provoque en général, avant exécution de celui-ci, les étapes suivantes :

- sauvegarde du contexte appelant (registres, piles, données)
- empilage des paramètres dans leur ordre d'apparition lors de la déclaration, ou dans l'ordre inverse
- empilage de l'adresse de retour à l'appelant
- préparation de l'accès au contexte de l'appelant
- passage au contexte de l'appelé

Voici un exemple de ces étapes en MICROSOFT C sous un environnement MS-DOS [MS-C 87, MS-QUICKC 87, MS-PASCAL 85a] :



Evolution de la pile du 8086, au cours d'un appel de sous-programme MS-DOS

Fig II-10

- 1 : l'appelant empile chacun des paramètres. Après quoi, le registre de pile SP pointe sur le dernier paramètre empilé.
- 2 : l'instruction CALL par l'appelant provoque l'empilage de l'adresse de retour. Après quoi, SP pointe sur cette adresse empilée.
- 3 : l'appelé empile l'ancienne valeur (celle associée à l'appelant) du pointeur de base de pile (BP) : ce pointeur permet d'accéder aux données locales de l'appelant. C'est une sauvegarde de contexte ; après quoi, SP pointe sur cette valeur de BP.
- 4 : l'appelé crée son pointeur de base (référence) de pile, en y rangeant la valeur actuelle de SP (BP = SP), décalée de 2 octets (1 mot machine). Ainsi, une indexation de type "BP + i" permet d'accéder aux données transmises par l'appelant, et celles de type "BP - i" permettent d'accéder aux données locales de l'appelé.
- 5 : l'appelé crée son contexte local (par empilage de ses variables locales) grâce à la pile. Le registre BP est ici une adresse fixe de la pile. le sous-programme appelé s'en sert pour référencer ses variables et les paramètres effectifs d'appel (cf point 4). Quoi qu'il arrive, le registre SP pointe toujours sur le sommet (bas) de la pile. une fois cette création de contexte local faite, le sous-programme se déroule réellement.
- 6 : dès que l'appelé a terminé son déroulement, il y a restitution de contexte de l'appelant, par destruction du contexte de l'appelé; ici on reprend la valeur de BP de l'appelant, celle qui avait été mémorisée à l'étape 3. Puis l'appelant reprend son déroulement à l'adresse de retour sauvegardée à l'étape 2.

De manière générale, les sauvegardes de contexte se font par l'appelant (avant l'appel) pour ses données, et par l'appelé (pendant l'appel) pour les références d'accès aux données de l'appelant (BP ci-dessus).

La figure précédante suppose un passage de paramètres dans l'ordre d'appel (du premier au dernier). Or certains langages ne respectent pas cet ordre de passage, c'est le cas de C (du dernier au premier); car ceci permet de disposer de sous-programmes à nombre de paramètres variable. Cette description du déroulement d'un appel concerne le résultat de la traduction en langage machine des programmes, par le compilateur.

Si donc nous supposons que le langage du sous-programme appelant empile les données de la première à la dernière, et que le langage de l'appelé fasse l'inverse, les paramètres n'auront plus de signification pour l'appelé. Bien d'autres problèmes peuvent survenir à cette étape , si les deux langages ne respectent pas le même protocole d'appel. Ces problèmes sont difficiles à catégoriser, car ils dépendent étroitement de l'implantation des deux langages interfacés.

II.4.3.4. Protocole d'appel

L'exemple ci-avant du déroulement de l'appel d'un sous-programme MS-DOS, permet aisément de comprendre le problème de la sauvegarde et de la restitution de contexte entre les deux langages. Pour que l'appel et le retour se passent bien, il faut que les deux langages suivent le même ordre et les mêmes procédés protocolaires, au cours de l'opération. Il faut que les compilateurs des 2 langages implantent de la même façon la séquence d'appels de sous-programme ; ceci peut être exigé au départ par l'implantation du système d'exploitation (cas de VMS), ou simplement pour que l'interfaçage logiciel soit possible (cas de MS-DOS).

II.4.3.5. Connexion système inter-langages

Il faut enfin qu'on puisse créer une connexion système entre les deux langages. Les points II.4.3.1, II.4.3.2., et II.4.3.4. sont à considérer uniquement dans le cas où les langages peuvent communiquer ensembles, dans leur implantation sur le système d'exploitation cible (cas des interfaces directes vraies).

Dans les autres cas, il faudra fabriquer la voie de communication système entre les 2 langages. C'est à dire implanter le mécanisme qui transmettra les données de P1 à P2. Dans un système comme MS-DOS, une telle contrainte implique l'abandon de l'interfaçage de type direct. Sous d'autres systèmes, une telle solution peut être envisagée en conservant le type direct (cas des pipes UNIX).

II.4.3.6. Conclusion

Les quelques problèmes que nous venons de décrire sont assez fréquents, mais ne constituent pas une liste exhaustive des déboires rencontrés lors de la création d'une interface logicielle. Chaque cas d'interfaçage amène de nouveaux problèmes inconnus, et particuliers au système d'exploitation utilisé, aux langages interfacés, et à leurs implantations respectives.

En général, les minis et gros systèmes d'exploitation sont suffisamment bien conçus, pour que le réalisateur de l'interface ne rencontre qu'un "minimum" de difficultés. Ces systèmes imposent en effet une manière rigide et protocolaire de créer et d'utiliser des programmes ou processus exécutables ; ce qui facilite grandement le travail à réaliser, à la fois pour les implanteurs de langage, et pour les concepteurs d'interface. Ce n'est pas le cas des petits systèmes d'exploitation, laissant à la charge du programmeur beaucoup d'aspects de système et de logiciel de base à l'intérieur de leur application, comme c'est le cas de MS-DOS.

Dans le paragraphe qui suit, nous donnons deux exemples d'interface logicielle, l'une étant de type direct vrai, l'autre de type direct intermédiaire, toutes deux implantées sous MS-DOS. Nous entrerons le moins possible dans les détails techniques, bien que nous y soyons parfois contraints. Ces deux exemples ont abouti à des applications internes à la société METADESIGN, l'une pour la standardisation de pilotes d'unité graphique, l'autre pour l'uniformité de structure des interfaces langages (binding) pour GKS.

II.5. Deux exemples d'interface logicielle

II.5.1. Introduction

Les deux exemples que nous présentons ici ont été réalisés et utilisés pour la société METADESIGN, afin de généraliser le modèle et les possibilités d'interface langage pour le noyau GKS (Cf Annexe II-0), c'est-à-dire offrir aux utilisateurs de METADESIGN-GKS4.0 un éventail de *bindings* très grand, qu'on puisse enrichir facilement avec de nouveaux langages, grâce à un modèle que l'on peut implanter simplement dans tous les langages disponibles sur MS-DOS. Le but de la première réalisation était d'avoir un accès très rapide et pratique, par simple lancement d'un programme spécialisé, à des configurations matérielles diverses de pilotes d'unité graphique utilisés par GKS ; ceci a abouti aux ressources résidantes, que nous décrivons dans cette partie.

Ces deux réalisations ont été faites sur le système MS-DOS, dans un contexte d'utilisation de noyau graphique GKS, soit pour interfacer avec un nouveau langage, cas de GKS40, soit pour interfacer avec des matériels divers, cas des ressources résidantes. La technique des ressources résidantes a été appliquée pour faire un essai d'interfaçage entre TURBO-PROLOG et GKS.

Ce ne sont certes pas des exemples d'interfaces entre langage IA et procéduraux, mais elles donnent une idée claire des distinctions que l'on doit faire entre les interfaces directes vraies et les autres.

Afin d'expliquer le fonctionnement de ces interfaces, nous devons décrire succinctement le système MS-DOS, sur lequel ont été réalisés ces logiciels, et quelques traits ou finesses caractéristiques de celui-ci.

II.5.2. Particularités du système MS-DOS

Les renseignements techniques que nous donnons ici sont issus de notre expérience du système MS-DOS, et de quatre références principales sur ce système d'exploitation [Duncan 86, Martin & Piette 87, MASM 85, MS-DOS 84].

II.5.2.1. Structure

Le système est divisé en plusieurs couches, qui servent à isoler le noyau logique et la perception du système par l'utilisateur, du matériel sur lequel il est implanté. Ces couches sont, de la partie la plus dépendante du matériel vers la plus indépendante :

Le BIOS :

Il est spécifique de la machine cible, et fourni par le constructeur de celle-ci. Il contient les pilotes d'unité, appelés **device driver** en anglais, ou même abusivement *driver*, dédiés à la machine, pour des appareils physiques d'un des types suivant :

- l'ensemble clavier-écran (driver dénommé CON),
- l'imprimante (driver PRN),
- une unité supplémentaire quelconque (driver AUX),
- la gestion de la date et de l'heure (driver CLOCK),
- l'appareil de démarrage à froid (boot system device), qui est situé sur disque (c'est un "block" device).

Les parties les plus dédiées au hardware des devices sont souvent situées en mémoire morte (ROM), pour pouvoir être aisément accessible par la majorité des programmes et commandes du système.

Le BIOS est chargé en mémoire vive à l'initialisation du système, à partir du fichier "IO.SYS", qui contient une partie du système : celle qui est la plus dépendante du matériel.

Le noyau

Il est fourni par la société MICROSOFT, MS-DOS étant une marque déposée de celle-ci. C'est un programme qui implante le système tel que le voit l'utilisateur. Il contient une série de primitives indépendantes du matériel, appelées "*fonctions système*", qui ont des fonctionnalités du type :

- manipulation de structures de fichiers,
- gestion de la mémoire,
- entrées-sorties sur des devices "caractères", l'un des deux types de devices existant en MS-DOS,
- chargement et exécution de programmes,
- accès à l'horloge en temps réel.

On peut appeler une fonction système en chargeant des registres du processeur 8086, avec des paramètres spécifiques, et en déclenchant ensuite un "**appel système**", par l'intermédiaire d'une "**interruption soft**" spécialisée, que nous précisons plus loin.

Le noyau du MS-DOS est chargé en mémoire vive, pendant l'initialisation du système, à partir du fichier "MSDOS.SYS".

Le processeur de commandes (SHELL)

C'est l'interface utilisateur du système ; elle lit, traduit, et transmet les commandes données par l'utilisateur au système, comme par exemple le chargement et le lancement d'un programme. On peut le remplacer par son propre processeur, en modifiant le fichier de configuration du système "CONFIG.SYS", qui est lu et interprété au démarrage de la machine.

Il est divisé en trois parties :

- Une partie résidante en mémoire, c'est à dire qui n'est jamais déchargée, ni remplacée, pendant l'utilisation du système. Elle est chargée en mémoire, juste "au-dessus" (adresses supérieures) du noyau MS-DOS, et gère des fonctions spéciales, telles que "CTRL-C" ou la terminaison de programmes chargeables. Elle transmet à l'utilisateur les erreurs de fonctionnement du système, et contient le code permettant de charger la troisième partie (I.e. la partie chargeable) de lui-même.
- Une partie d'initialisation, qui est chargée au-dessus de la partie résidante précédente, au moment du démarrage du système. Elle traite les commandes du fichier de commandes de démarrage ("AUTOEXEC.BAT"), et elle est ensuite déchargée de la mémoire.
- La partie "*externe*", ou "*chargeable*", qui est chargée en "*haut*" de la mémoire vive, c'est à dire aux adresses les plus grandes. Elle lit les commandes, entrées au clavier ou par l'intermédiaire d'un fichier de commandes (fichier "*batch*"), et provoque leur exécution. La partie de mémoire où elle se trouve peut être "*écrasée*", c'est à dire détruite par chargement ou exécution d'une commande ou d'un programme. La partie résidante du processeur, quand un programme se termine, vérifie que la mémoire n'a pas été modifiée à cet endroit ; si ce n'est pas le cas, elle recharge une copie "propre" de cette partie externe.

Les types de commandes possibles

Il y a trois sortes de commande connues du shell :

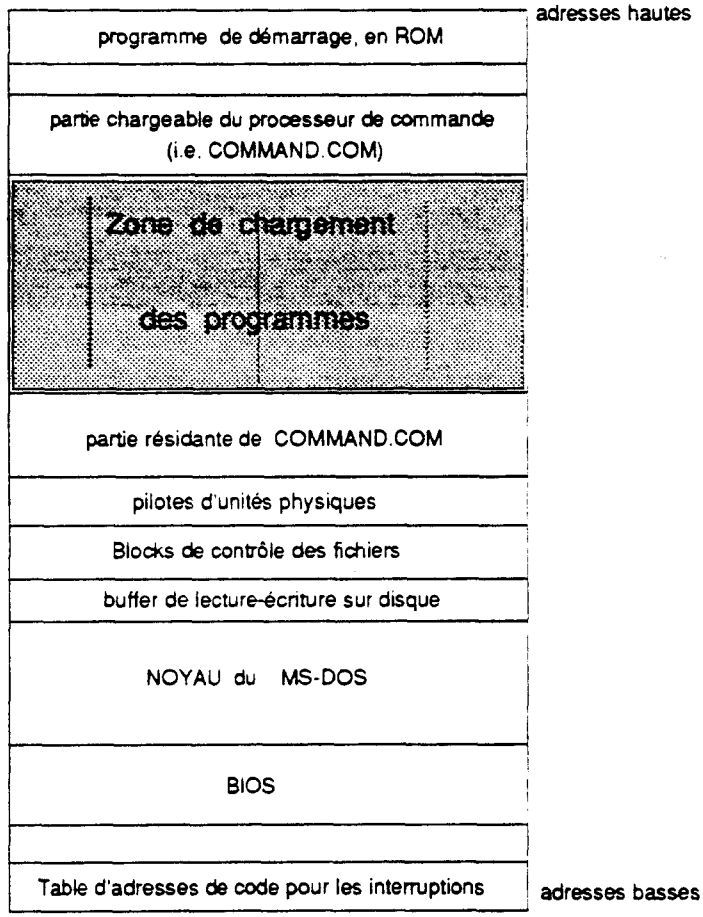
- Les **commandes internes**, qui sont traitées par du code qui se trouve dans la partie chargeable de "COMMAND.COM" (I.e. le processeur de commande). On peut citer des commandes telles que *dir, copy, ren*, ...
- les **commandes externes**, qui sont chargées en mémoire à partir d'un fichier sur disque, dans la zone de chargement de programme de la mémoire vive. Les fichiers ont une extension *.COM* ou *.EXE* Dans le cas de nos interfaces décrites ici, il s'agit toujours de faire fonctionner ensemble des programmes à extension *.EXE* qui sont chargés l'un après l'autre en mémoire.
- les **fichiers de commandes**, encore appelés fichiers "*BATCH*", qui contiennent une liste alphanumérique de commandes internes ou externes -- munies de leurs paramètres si nécessaire -- ou même d'autres fichiers de commandes. Il sont traités par la partie chargeable du shell.

Quand une commande, à extension .EXE, est appelée par l'utilisateur, le shell utilise et appelle la fonction du système "*EXEC*", qui crée une structure spécifique à la commande, structure appelée "*Program Segment Prefix*" (PSP), qui est chargée en même temps que le code exécutable de la commande, juste aux adresses inférieures au début du code. Cette structure contient des informations diverses, destinées au système. Avant que la commande ne se termine, en fin de déroulement, elle doit appeler une fonction de fin d'exécution, qui li-

bère la place mémoire qu'elle occupait, et redonne le contrôle du système au programme qui a provoqué son chargement. Notons que MS-DOS ne permet aucunement le traitement multi-tâche, et donc que deux programmes ne peuvent se dérouler simultanément.

Enfin, après initialisation et chargement du système, la mémoire se divise comme suit :

Pour que la compréhension du fonctionnement des interfaces soit simplifié, nous devons



Configuration mémoire, après chargement de MS-DOS

II-11

maintenant parler des fonctionnalités de MS-DOS que nous avons utilisées.

II.5.2.2. Le Program Segment Prefix

Cette structure, que nous avons déjà citée auparavant, est essentielle pour l'exécution des programmes sous MS-DOS. Elle est d'une longueur de 256 octets, et contient des liens vers l'environnement MS-DOS, ou contexte d'utilisation MS-DOS, au moment du lancement de la commande ou du programme auxquels elle est associée.

En ce qui nous concerne, elle contient surtout, aux offset 02h et 2Ch du début de son implantation en mémoire, respectivement l'adresse du dernier segment mémoire occupé par la commande chargée, et l'adresse du segment contenant l'environnement MS-DOS courant (des chaînes de caractères décrivant des états spécifiques du système).

Ceci nous permet, pour nos interfaces, de déterminer la taille de "*paragraphes*" (mots de 16 bits du 8086) à garder résidants en mémoire, lors de l'installation de la ressource résidante (Cf II.5.3), ou du GKS résidant (Cf II.5.4), et de libérer cette place manuellement, par un programme spécialisé, quand le programme résidant devient inutile.
Cela nous permet aussi de restituer l'environnement du MS-DOS, tel qu'il était avant le chargement de ces logiciels.

II.5.2.3. Les interruptions soft

Elles peuvent être déclenchées, de façon synchrone, par un programme d'application qui exécute une instruction d'interruption du processeur 8086 : *INT Numero-Interruption*. Dans nos interfaces, nous nous servons essentiellement des deux interruptions soft suivantes:

- l'interruption soft de numéro 21h, spécialisée dans l'appel de fonctions système.
- l'interruption 60h, inutilisée par MS-DOS, qui permet d'appeler les routines des ressources résidantes, après que nous l'ayons initialisée.

Nous citons ci-après les interruptions utilisées, ainsi que leur fonction, dans le cas des ressources résidantes (Cf II.5.3), et du GKS résidant (Cf II.5.4).

Cas des ressources résidantes

Installation de l'interruption 60h, pour qu'elle pointe sur la table des fonctions disponibles de la ressource :

```
mov ds , Code_segment_de_la_table
mov dx , Offset_de_routine_de_sélection_fonction
                ; ds:dx = point d'entrée de la routine associée à l'int 60h

mov ah , 25h        ; sélection de "set-interrupt-vector", fonctions système dos
mov al , 60h        ; numéro d'interruption à installer
int 21h             ; lancer l'installation (appel fonction 25h du système)
```

Terminaison du programme, en le laissant résidant :

```
mov dx , nb_paragraphes_a_garder_residant
mov al , 00h            ; code retour inutilisé
mov al , 31h            ; sélection de "terminate-and stay-resident"
int 21h                 ;charger le programme et le laisser résidant
```

Appel d'une fonction du programme résidant :

```
Nom_de_fonction  proc near
                 mov bx , Numero_de_fonction ; dans la table des fonctions de la ressource
                 mov cx , Taille_des_parametres ; en bytes
                 int 60                  ; appel de l'interruption 60h
                 ret Taille_des_parametres  ; retour au programme d'application appelant
```

L'appel "*int 60h*" sauvegarde le contexte du programme appelant, copie la pile de celui-ci dans la pile réservée à la ressource résidante, puis appelle la fonction désirée grâce au numéro passé dans le registre BX. Il y a ensuite restitution du contexte de la fonction appelante ; enfin on fait un retour d'interruption soft classique ("*iret*").

Cas du GKS résidant

Dans ce cas, les interruptions soft sont utilisées uniquement pour appeler des fonctions du système :

- envoi de message à l'écran (un copyright et un message de bonne installation)

  fonction *write_file_or device* (numéro 40h)

- retour au système en laissant GKS4.0 résidant

  fonction *terminate-and-stay-resident*(numéro 31h)

Mais on a aussi les appels suivant :

- lecture de l'adresse pointée par l'interruption spécialisée pour les appels de fonctions du système (int 21h)

  fonction *get-interrupt-vector* (numéro 35h)

- positionner cette interruption sur notre procédure de filtre "*FILTRE*

  fonction *set-interrupt-vector* (numéro 25h)

- récupérer, dans le programme résidant, les appels à des fonctions GKS, par écriture dans un fichier. L'écriture dans un fichier MS-DOS passe obligatoirement par un appel système avec l'int 21h, d'où les redirections précédentes de cette interruption soft et cette procédure *FILTRE*, pour sélectionner correctement ce qui est un appel au GKS résidant.

Enfin, pour revenir à l'état initial :

- restitution de l'interruption système initiale (int 21h)

  fonction *set-interrupt-vector* (numéro 25h)

- suppression du code résidant et du bloc de variables d'environnement du programme

  fonction *free-allocated-memory* (numéro 49h)

Les "*interruptions soft*", qui permettent en particulier l'appel de fonctions du système, grâce à l'appel spécialisé de numéro 21h, sont essentielles dans notre cas. En effet, nous devons en quelque sorte modifier le système MS_DOS, surtout dans le cas du GKS résidant où nous modifions l'interruption la plus utilisée du système, pour pouvoir faire coexister la partie application (partie P1) et la partie interfacée (partie P2). Dans les deux cas, on passe par des interruptions pour accéder à P2 : dans le premier, on appelle directement l'interruption 60h, qui permet d'appeler la fonction de la ressource ; dans le deuxième, on modifie le système (int 21h), et on se sert des fonctions de manipulation de fichiers, disponibles dans l'implantation du langage utilisé pour écrire l'interface à P2 dans P1 ; plus précisément pour manipuler un fichier dont on récupère (en entrée de GKS) ou transmet (en sortie de GKS) les données échangées entre P1 et P2.

II.5.2.4. Manipulation de fichiers

Ces fonctions permettent d'accéder à des fichiers, d'une manière similaire à celle du système UNIX. Les fichiers sont désignés par une chaîne "ASCIIZ" (chaîne de caractères ASCII terminée par un caractère 0, caractéristique du langage C), qui contient le nom d'un drive, un chemin d'accès de répertoire, un nom de fichier, et une extension. Quand on veut créer, ouvrir, ou écrire et manipuler un fichier, on peut se servir, et c'est d'ailleurs ce que font les implanteurs de langage sur MS-DOS, des fonctions suivantes par appel de fonction système (int 21h), appelées "*File handle and record manipulation*" :

3Ch : *Create-File(-Handle)*

3Dh : *Open-File(-Handle)*

3Eh : *Close-File*

3Fh : *Read-Record*

40h : *Write-Record*

et d'autres fonctions inutiles pour notre exposé.

Ces fonctions constituent un noyau fort pratique de manipulation de fichiers, et peuvent être utilisées avantageusement, pour faire un interfaçage de type direct intermédiaire, ce que nous avons fait pour le deuxième exemple : le GKS4.0 résidant (Cf II.5.4).

Cette rapide présentation de MS-DOS, et de certaines de ses particularités, n'a pas d'autres buts que de familiariser le lecteur avec des notions de système utilisées dans les deux exemples d'interface logicielle qui suivent.

II.5.3. Premier exemple : ressources résidantes

Nous donnons ici un premier exemple d'interface directe vraie sous le système MS-DOS. Sans entrer profondément dans les détails techniques, nous décrivons son contenu et son fonctionnement.

Cet interfaçage avait comme but initial de créer une méthode générale d'interface d'un langage quelconque sous MS-DOS avec les programmes produits par les outils MICROSOFT en PASCAL. On voulait surtout être capable de proposer un "binding" (Cf. GKS Annexe II-0) pour les langages Turbo-PASCAL 3.0 [TB-PASCAL 85] et Turbo-PROLOG [TB-PROLOG 86], implantations respectives de PASCAL et PROLOG par la société BORLAND.

Il fût ensuite utilisé pour disposer de pilotes d'unités graphiques chargeables et déchargeables à volonté, dans l'implantation GKS 3.0 de METADESIGN.

Cette interface est directe car elle utilise à la fois des ressources logicielles des langages avec lesquels on interface, ainsi que des fonctionnalités du système d'exploitation MS-DOS, comme la gestion d'une interruption système inutilisée, pour faire coexister en mémoire le programme d'application et la ressource résidante, ainsi que pour assurer la communication entre les deux parties.

II.5.3.1. Présentation

Une RESSOURCE RESIDANTE est l'outil qui permet d'appeler des utilitaires de tout ordre, contenus dans des bibliothèques de procédures, sachant que ces utilitaires se trouvent chargés en un seul bloc en mémoire vive.

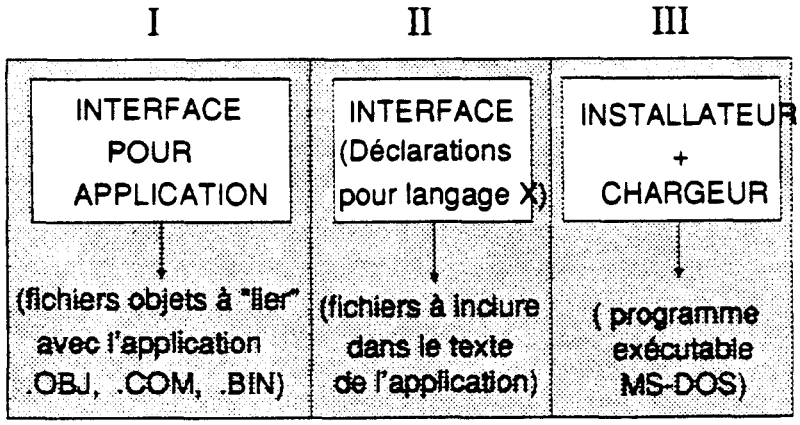
- Une ressource résidante est utilisée par un programme application nécessitant, pour son fonctionnement, les utilitaires qui y sont contenus.
- La ressource résidante doit être chargée UNE SEULE FOIS en mémoire vive avant toute utilisation par un programme.
- Le programmeur d'application dispose de la liste des procédures disponibles dans la ressource résidante. C'est un document livré par le créateur de la ressource.

Note : Une ressource résidante n'est utilisable que par UN SEUL LANGAGE de programmation, celui du programme d'application, à moins qu'il existe pour cette même ressource une(des) interface(s) pour application adaptée(s) au langage utilisé.

Autres définitions possibles :

- ressource système supplémentaire.
- édition de lien semi-dynamique pour le langage d'application.

De quoi est constituée une ressource résidante :



Les éléments d'une ressource résidante

Fig II-12

Comment fabriquer une ressource résidante :

A : *écrire en langage X le programme d'application, en respectant la syntaxe d'appel des procédures décrite dans les fichiers à inclure (pragma ou directive "include" du langage X).*

B : compiler les sources de l'application, et faire l'éditions de liens ou inclure les fichiers à extension COM du programme.

C : exécuter une seule fois l'installateur chargeur.

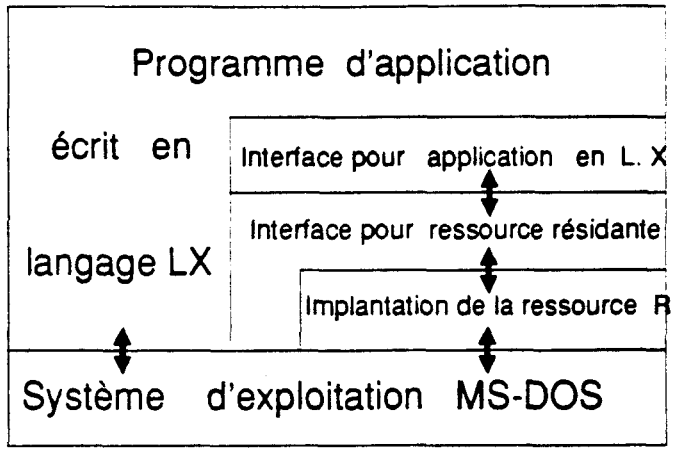
D : exécuter le programme d'application.

II.5.3.2. Modèle de fonctionnement

Pour se servir d'une ressource résidante, le programmeur d'application doit "**installer**" celle ci. C'est à dire lancer le programme qui va charger en mémoire les utilitaires de la ressource. Il peut ensuite se servir comme à son habitude des utilitaires lui permettant de créer des programmes : éditeurs de texte, compilateurs, éditeurs de liens, debuggers ... Si il installe deux fois la ressource résidante, le fonctionnement de celle-ci n'est plus garanti.

Toute application utilisant une ressource résidante est constituée de quatres parties :

- la partie application écrite en langage LX.
- l'interface pour application.
- l'interface pour ressource résidante.
- l'implantation de la ressource résidante.



Modèle en couche de l'interface ressource résidante(int 60H)

Fig II-13

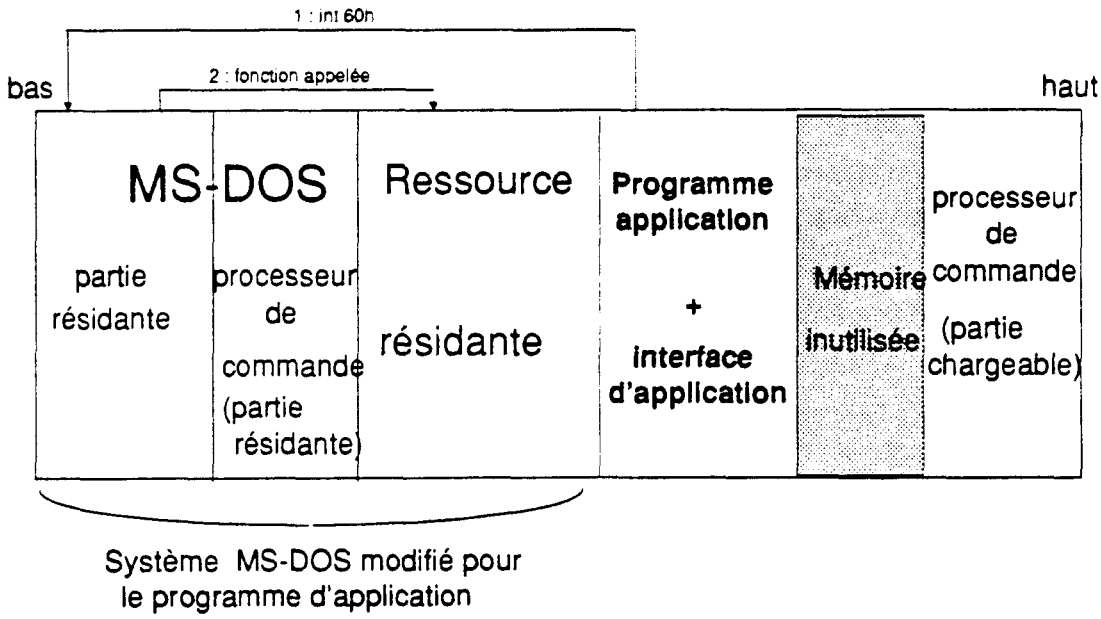


Image mémoire d'une application utilisant une ressource résidante

Fig II-14

Architecture et image mémoire d'une ressource résidante :

II.5.3.3. La partie application

C'est l'ensemble des parties écrites en langage X, ou en plusieurs langages distincts, formant un ensemble cohérent en tant que programme. UN et UN SEUL LANGAGE est autorisé à se servir de la ressource résidante de par le simple fait que celle-ci est dédiée à ce langage, donc qu'elle ne fonctionnera pas avec un autre.

Ceci est dû au fait que MS-DOS n'est pas un système multitâche, et que les implanteurs de langages doivent créer et manipuler eux-mêmes le partitionnement et la gestion en "segment:offset" de la mémoire vive, provoquant de ce fait des incompatibilités entre les implantations de langages.

Pour pouvoir utiliser la ressource résidante, le programme en langage LX doit respecter un protocole syntaxique, dont la forme est contenue dans un ou plusieurs fichiers d'interface de deux types possibles :

- les fichiers de types à utiliser pour cette ressource avec le langage LX. Ils donnent une description, en langage LX, des types compatibles avec ceux de la ressource résidante.
- les fichiers de procédures ou fonctions, avec noms et paramètres de celles-ci, disponibles pour cette ressource.

Suivant le contenu et l'implantation de la ressource résidante, ces fichiers seront plus ou moins complexes. Ils sont fournis au programmeur d'application avec l'installateur de ressource .

Exemple en Turbo-PASCAL 87 3.0 :

ressource résidante : procédures d'envoi et réception de réels et entiers

```
procédureVALUE_INTEGER(I:integer)
  external 'ZOOTBPA.BIN';
procédure VAR_INTEGER(var I: integer);
  external VALUE_INTEGER[11];
procédure VALUE_REAL(R: real);
  external VALUE_INTEGER[22];
procédure VAR_REAL(var R: real);
  external VALUE_INTEGER[33];
```

Note : On remarquera que l'on a utilisé la connexion logicielle fournie par le constructeur du langage Turbo-PASCAL.

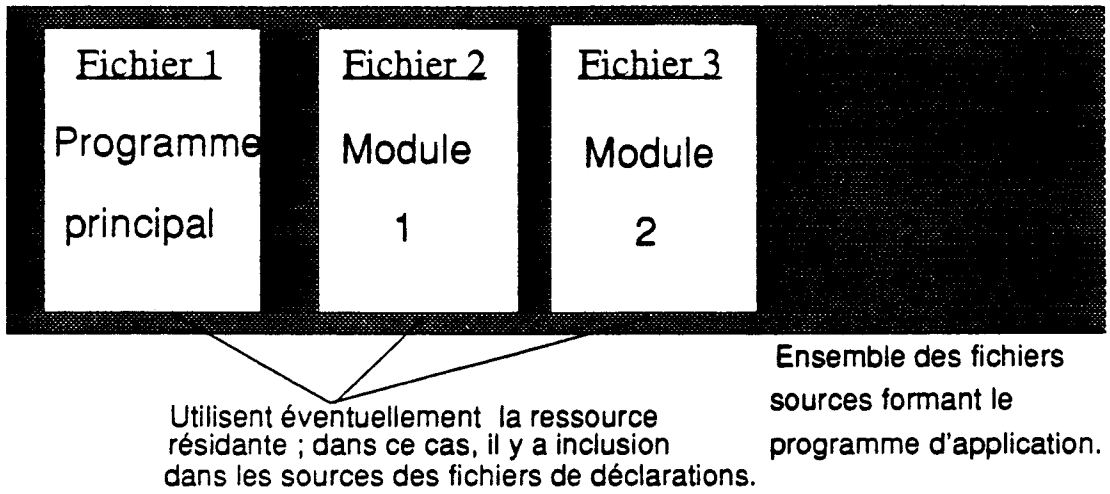
Fichier de types GKS à utiliser dans une application en TURBO-PASCAL

```
Wks_type = (screen,plotter);
Wks_category = (input,outin,output);
Device_units = (metres,other);
Etat_input = (ok,none);
Point = record
        X,Y : real;
              end;
Points_array = array [1..50] of Point;
```

Si les deux fichiers précédents s'appellent respectivement "CONVTBPA.INT" ("INT" pour interface), et "CONVTBPA.TYP" ("TYP" pour fichier de types), on aura un schéma d'écriture du programme d'application du genre :

```
program (input, output);
.
... déclarations de types et de variables...
$Include:"CONVTBPA.TYP"
... déclarations de procédures et fonctions (sous-programmes) de l'application ...
$Include :"CONVTBPA.INT"
.
begin
...
Value-real(3.14);
...
end.
```

Si l'application est constituée de plusieurs modules ou "overlays", on aura une architecture logicielle d'application telle que :



un programme d'application utilisant une ressource résidante

Fig II-15

II.5.3.4. L'interface pour l'application

Cette partie, souvent limitée, permet de faire le lien entre un appel à une fonctionnalité de la ressource résidante et le programme d'application.

Quand le programme appelle une procédure, dont la syntaxe est compatible avec le langage LX, et qui est disponible dans la ressource, celle-ci est en fait un élément de l'interface pour l'application; ce sous-programme fait les manipulations de données et de système MS-DOS, nécessaires au bon fonctionnement des deux parties (application et résidante).

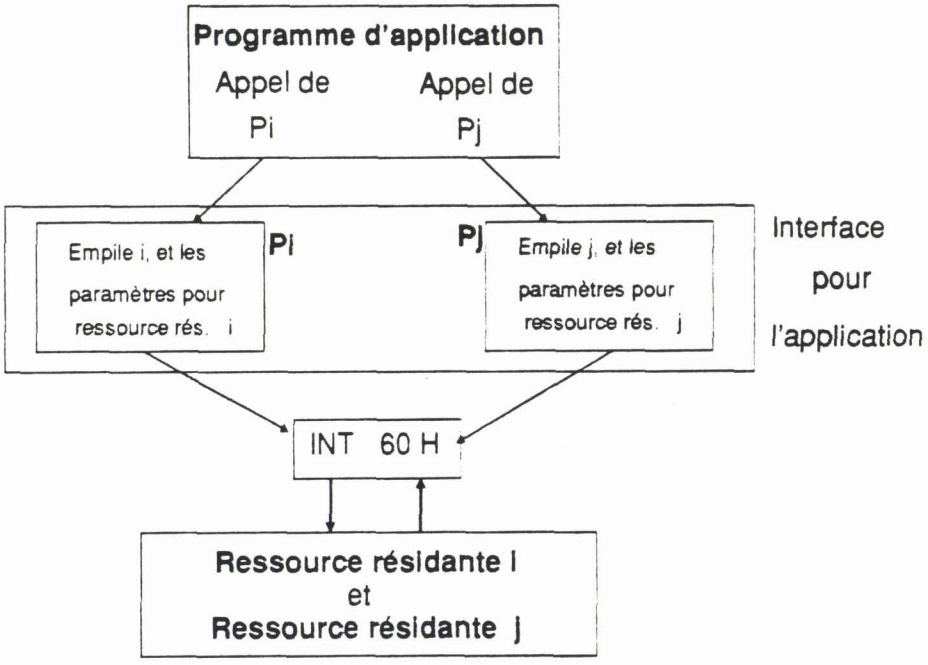
Si une édition de lien est nécessaire pour utiliser la partie résidante avec le langage LX, et cela est alors indiqué dans le guide d'utilisation de la ressource, elle sera faite avec cette partie "interface pour l'application".

Dans les interfaces que nous avons réalisées, aussi bien pour l'utilisation des pilotes d'unité graphique que pour l'interfaçage Turbo-PASCAL - GKS, on passe simplement à la ressource résidante le numéro de procédure appelée, ainsi que la taille des paramètres empilés, puis on appelle l'interruption 60H (laissée libre pour l'utilisateur par le MS-DOS et le BIOS), dont le vecteur d'interruption est modifié pour pointer vers le programme exécutable contenant la ressource résidante.

Résumé : La partie "Interface pour application", dans tous les cas assez restreinte, ne sert qu'à faire le lien entre l'application et la ressource. Elle se présente soit sous la forme d'une bibliothèque à utiliser par le programmeur à l'édition des liens, c'est le cas pour MS-C [MS-C 87], Turbo-PROLOG [TB-PROLOG 86], MS-FORTRAN [MS-FORTRAN 87], MS-PASCAL [MS-PASCAL 85], soit comme un fichier MS-DOS à extension COM ou BIN à relier avec le programme (cas de Turbo-PASCAL).

Dans tous les cas, le programmeur dispose du ou des fichiers décrivant la syntaxe d'appel, qui contient la liste des sous-programmes ou procédures utilisables par le langage LX.

Voici un schéma décrivant le flot des données dans les parties de la ressource résidante :



Le déroulement d'un appel à une ressource résidante

Fig II-16

II.5.3.5. Interface pour ressource résidante :

Cette partie assure essentiellement :

- le changement de contexte entre les deux process "programme application" et "ressource résidante",
- le protocole de communications entre les deux process,
- l'appel à une procédure précise de la ressource.

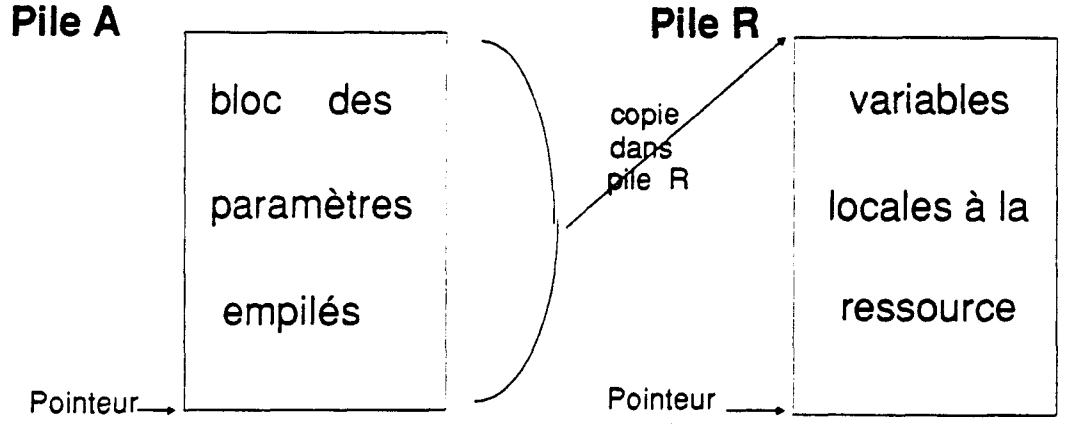
Du point de vue changement de contexte, elle sauvegarde les registres du 8086 du programme appelant, fait la manipulation de segments mémoire nécessaire au fonctionnement de la ressource résidante.

Ceci étant fait, elle communique à la ressource résidante les paramètres du programme appelant par l'intermédiaire de la pile 8086. Elle appelle, une fois que les paramètres ont été transmis à la ressource, la procédure visée de celle ci, grâce à sa position dans l'exécutable (une adresse mémoire), qui est rangée dans un tableau contenu dans l'installateur-chargeur.

Au retour de la procédure appelée, elle restitue le contexte du programme appelant, et y retourne avec le retour d'interruption 60H.

(Interface pour application)(interface pour ressource résidante)

Programme appelant	Ressource résidante
Contexte appelant (A)	Contexte du résidant (R)
registres 8086:	registres 8086
AX, BX, CX, DX, DI, SI, SS, SP	AX, BX, CX, DX, DI, SI, DS, SS, SP



Déroulement de la communication inter-langage :

1) sauvegarde du contexte A,

2) mise en place du contexte R,

3) copie de la pile de A dans la pile de R, correctement (peut nécessiter des modifications d'empilages ou de codage des données),

4) appel du sous-programme contenu dans la ressource R

II.5.3.6. Implantation de la ressource résidante

C'est la partie où existe réellement la procédure utilitaire appelée par le programme application. Elle est écrite en un langage quelconque, les seules contraintes étant que celui-ci puisse être assemblé avec la partie interface pour ressource résidante, en un process MS-DOS habituel, et qu'il permette en plus de savoir quelle est la taille allouée par le MS-DOS à cet ensemble, de manière à pouvoir "décharger" la ressource de la mémoire.

L'annexe II-1 donne un exemple de programme écrit en Turbo-PASCAL interfacé avec un noyau GKS résidant. On remarquera surtout les fichiers de type à utiliser, ainsi que la forme des procédures que l'on appelle dans la ressource résidante.

II.5.3.7. Synthèse et conclusion

Ce premier exemple montre ce qu'est une interface directe vraie. Essayons de faire la synthèse des éléments de l'interface logicielle précédente :

Les langages des programmes d'application utilisent un outil d'interface disponible dans leur implantation.

L'outil est constitué d'une **déclaration syntaxique** (ex. en TURBO-PASCAL 3.0 la déclaration des procédures externes disponibles; Cf "CONVTBPA.INT"), et d'un **mécanisme système** (ex. en TURBO-PASCAL 3.0 la liaison avec l'implantation assembleur des procédures externes déclarées précédemment ; à CONVTBPA.INT correspondCONVTBPA.BIN).

Dés lors que nous sommes capables de **produire le lien entre le langage de la ressource et ce mécanisme système**, nous pourrons interfacer correctement les deux langages.

Dans le cas de nos ressources résidantes, nous produisons automatiquement un fichier assembleur, à partir des déclarations de procédures externes. Ce fichier est transformé en code exécutable (à extension .OBJ ou .BIN ici, partie interface pour application), qui contient en fait l'appel aux procédures de la ressource résidante, moyennant éventuellement des traitements de gestion de contextes et de codage des données.

Il faut ensuite **fabriquer la ressource, ainsi que son interface** (interface pour ressource résidante ici).

Dans le cas présent, il fallait utiliser une particularité de MS-DOS qui permet de charger un programme et le laisser résidant en mémoire (Cf II.5.2.3.). Nous avons dû aussi passer par un artifice système (appel de l'interruption libre 60H) pour transférer le contrôle à la ressource résidante et retourner à l'appelant.

Réaliser une interface directe vraie paraît simple à première vue. Mais on s'aperçoit vite, et d'autant plus que le système d'exploitation est pauvre, que des problèmes aussi divers qu'inattendus apparaissent. Finalement, les problèmes rencontrés sur ce système, viennent essentiellement du fait que les implanteurs des langages L1 ou L2 ne peuvent fournir chacun qu'une moitié des outils nécessaires à la création de l'interface, et que ces outils sont incompatibles entre eux.

II.5.4. Deuxième exemple : GKS 4.0 résidant

II.5.4.1. Présentation

Cet exemple montre ce qu'est une interface directe intermédiaire. Elle est directe, car on lie réellement, grâce au système d'exploitation, les 2 parties ; elle est intermédiaire, car les données à communiquer sont transmises par l'intermédiaire d'un fichier (Cf II.4.2.2.).

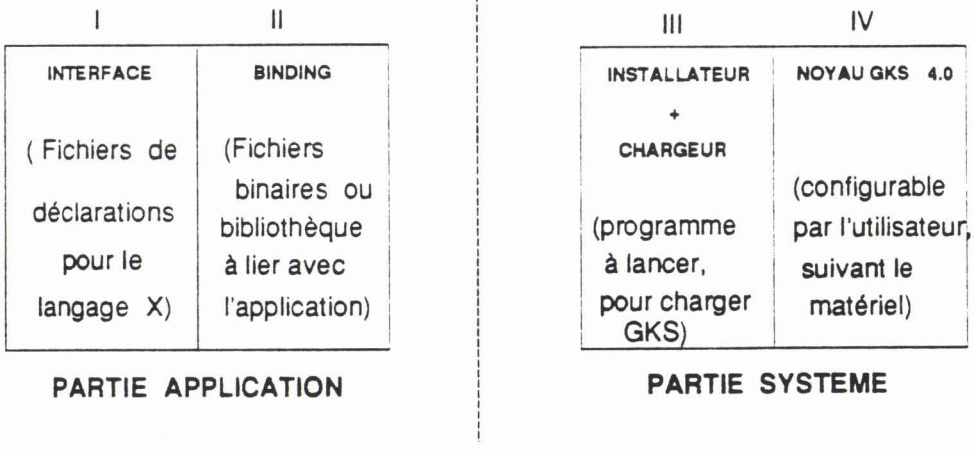
Cette interface a permit la généralisation et la formalisation des implantations de binding GKS, pour l'implantation GKS 4.0 de METADESIGN. Pas moins de sept implantations de langage différentes, provenant d'éditeurs de logiciels, ont pu ainsi être interfacées avec ce noyau GKS, pour des langages d'informatique scientifique (MS-FORTRAN, MS-PASCAL, Turbo-PASCAL), pour le langage orienté système C (MS-C [Dalmasso 88], Turbo-C), et pour le langage ADA (ALSYS ADA [Verdin 88]).

Pour implanter un nouveau binding, peu de contraintes sont exigées sur le langage de P1, nouveau langage à interfacer avec GKS 4.0 :

- connaître le codage des types de bases (entier, réel, chaîne, enregistrement)
- pouvoir créer un "File Handle" MS-DOS, lire et écrire sur celui-ci, et communiquer ainsi avec le noyau GKS.

On peut envisager aisément une interface langage GKS, pour une implantation de langage IA sur MS-DOS, tel que D-PROLOG, car celui-ci est implanté avec MS-PASCAL et MS-C, et parce que l'on peut modifier le noyau D-PROLOG par une édition de liens statiques (Cf II.6.3.4.).

De quoi est constitué GKS résidant :



Constitution logicielle de GKS 4.0

Fig II-17

L'utilisateur ne se préoccupe que des parties I et II, pour interfacer GKS avec son application, ainsi que de la partie IV, pour configurer le GKS sur son matériel -- carte graphique particulière, traceur, appareils d'entrée tels que souris, table à digitaliser... -- , grâce à un autre programme dédié à cette tâche.

comment créer une application GKS ?

- **A** écrire dans le langage X le programme d'application, en respectant la syntaxe et la sémantique du binding, telles qu'elles sont décrites dans la norme associée au langage utilisé, et les particularités d'implantation du GKS 4.0.
- **B** compiler et faire l'édition de liens de son programme d'application
- **C** exécuter l'installation-chargement de GKS 4.0
- **D** lancer son programme d'application

Note : Une fois que GKS 4.0 est installé, le MS-DOS fonctionne normalement, et l'utilisateur de la machine peut l'utiliser comme à son habitude. La modification du MS-DOS est donc invisible à l'utilisateur.

II.5.4.2. Modèle de fonctionnement

A l'installation en mémoire de GKS 4.0, l'installateur-chargeur effectue les actions suivantes :

- rediriger l'interruption d'appel de fonctions du système (INT 21h) vers une procédure "*FILTRE*", que nous avons créée, qui triera les appels au MS-DOS ou au GKS. Ceci comprend la sauvegarde de l'ancien "vecteur d'interruption" de l'int 21h, c'est à dire un pointeur vers une adresse de code à exécuter, et la réaffectation de ce pointeur vers le code de *FILTRE*

- terminer le programme et le laisser résidant en mémoire (int 21h, fonction 31h).

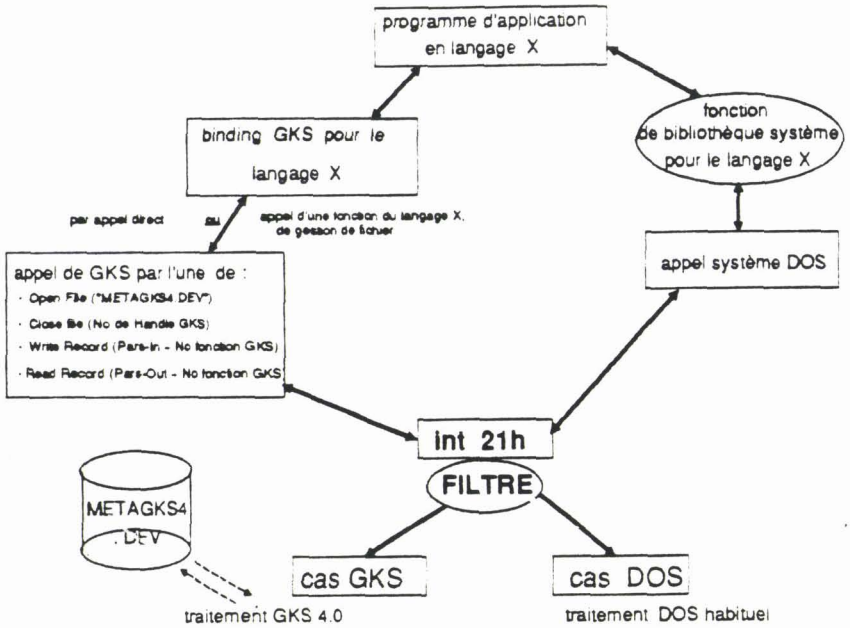
Lors de l'appel d'une fonction GKS par le binding, le cheminement des informations se fait de la manière suivante :

- écriture (ou lecture) dans un fichier, par l'intermédiaire d'un "file handle" de nom "METAGKS4.DEV", de la fonction GKS à exécuter, et des paramètres passés; cette opération se faisant automatiquement, et de façon interne, par une procédure de lecture - écriture sur fichier, fournie dans les librairie du langage LX, ou bien il faudra fabriquer soi-même cette manipulation de File Handle, à connecter ensuite avec le langage LX.
- la procédure précédente fait un appel "*Write to File Handle*" (int 21h , fonction 40h), et le déroulement du programme se poursuit dans le noyau GKS résidant, par la redirection initiale de l'int 21h.
- la procédure *FILTRE* trie l'appel et détermine le destinataire de celui-ci : MS-DOS ou GKS. Dans le premier cas, on poursuit le déroulement vers le code habituel du système, sinon, pour les quatres cas suivant de manipulation de fichier, "*Open File Handle*", "*Close File Handle*", "*Read Record*", "*Write Record*", on devra faire certains tests, pour savoir si l'opération est destinée au MS-DOS ou à GKS :
- 1) *Open File Handle* : on teste le nom du fichier. Si ce nom est "METAGKS4.DEV", on renvoie "-1" comme numéro de File Handle ouvert, sinon l'appel est destiné au MS-DOS.
- 2) *Close File Handle* : on teste le nom du Handle. Si c'est bien GKS qui doit être fermé, on réalise cette opération, et on retourne immédiatement au binding.
- 3) *Read Record* : on lit les paramètres en retour pour la fonction GKS appelée, dans le handle "METAGKS4.DEV", si c'est bien à GKS qu'on s'adresse.
- 4) *Write Record* : on transmet les paramètres en entrée, destinés à GKS, dans le fichier "METAGKS4.DEV", si c'est au GKS que l'on s'adresse.

Pour reconnaître si on s'adresse au GKS, il suffit de tester le numéro de Handle auquel on s'adresse. Il doit avoir la valeur -1, qui n'est jamais utilisée par MS-DOS.

- on duplique le bloc de paramètres en sortie, créé par la fonction GKS, dans le buffer du fichier "METAGKS4.DEV" (routine OUT_PARAMETRES, Cf. annexe II-2).
- On passe au contexte du GKS résidant, après avoir sauvegardé celui du programme appelant, on empile les paramètres du buffer de METAGKS4.DEV, comme si on faisait un appel classique MS-PASCAL, qui est le langage d'implantation du noyau GKS 4.0, et on appelle la fonction GKS visée (par la routine DISPATCHER, Cf annexe II-2).

La figure suivante résume tout cela :



Filtrage des appels système et GKS

Fig II-18

Nous ne reviendrons pas sur le mécanisme d'installation du GKS résidant, il est le même que pour le premier exemple des ressources résidantes.

Il est important de noter que nous avons réellement modifié l'interruption d'appels au système (int 21h). Comme nous filtrons les appels , et que seuls ceux pouvant intéresser le GKS sont interceptés réellement, le système MS-DOS a une apparence normale pour l'utilisateur. De plus, on utilise normalement les fonctions de manipulation de fichier MS-DOS, à la fois dans l'interface langage et dans le noyau résidant, et on simule les appels MS-PASCAL pour appeler les fonctions GKS.

Le schéma suivant montre l'image de la mémoire vive, quand GKS et le programme d'application sont tous deux chargés en mémoire. Nous y avons cependant inséré, par des arcs étiqueteuse et numérotés, le cheminement des données lors d'un appel GKS, dans tous les cas (arcs en traits pleins), et aussi quand GKS renvoie des données au programme appelant (arcs en traits pointillés).

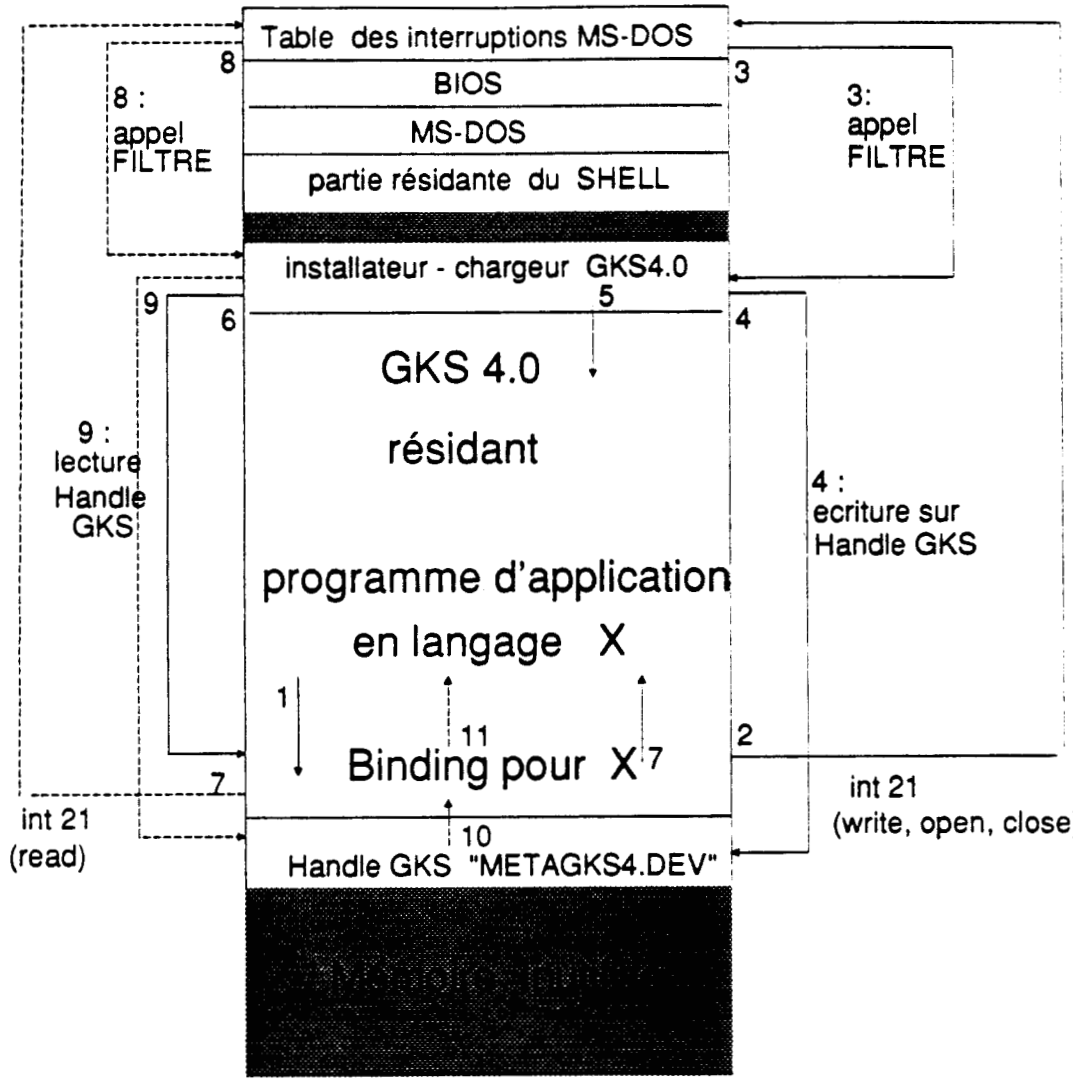


image mémoire, et appel d'une primitive GKS

Fig II-19

II.5.4.3. Binding en langage X

Cette bibliothèque logicielle, à lier normalement avec le programme d'application, est chargée de faire le transfert et le contrôle des informations, du langage X vers le langage MS-PASCAL, grâce auquel est implanté le noyau GKS. Elle constitue la partie II de l'application.

Les fonctions qu'assure le binding sont les suivantes :

- créer le File Handle GKS, demander son ouverture et sa fermeture, lors d'un appel des primitives "*OPEN-GKS*" et "*CLOSE-GKS*".
- tester que le "File Handle" de GKS est bien ouvert.

- si des codages de données ne correspondent pas entre le langage de l'application et le MS-PASCAL, il traduit ces données pour qu'elles soient assimilables par MS-PASCAL. Inversement, s'il y a des paramètres en retour, et que ceux-ci sont incompatibles avec le langage X, il réalise le codage inverse.
- écrire le bloc de paramètres et le numéro de primitive appelée dans le fichier GKS.
- lire les paramètres en sortie, et le numéro d'erreur GKS détectée par le noyau, s'il y en a, dans le fichier GKS.
- tester s'il y a des erreurs GKS, et si c'est le cas, les traiter suivant les implantations des primitives "*ERROR_HANDLING*" et "*ERROR_LOGGING*" . Notons que les textes sources de ces 2 primitives sont fournis au programmeur.

Rappelons que ce sont les opérations de création, de lecture ou d'écriture, et de fermeture de fichier, du langage X, qui appellent les fonctions de lecture-écriture sur File Handle du système MS-DOS, appels que la procédure FILTRE détourne vers les primitives du noyau GKS. Si le langage X n'utilise pas ces primitives du système, il faudra réaliser un appel direct aux fonctions MS-DOS.

Parmi les routines du binding, quelques unes jouent un rôle spécifique, car elles agissent directement sur le File Handle GKS :

- "*OPEN-GKS*", qui demande l'ouverture du fichier "METAGKS4.DEV", et reçoit en retour, de la part du noyau, un numéro de Handle GKS à utiliser, après ouverture (-1),
- "*CLOSE-GKS*", qui ferme, donc détruit, le File Handle GKS,
- "*EMERGENCY-CLOSE-GKS*" qui fait la même opération que CLOSE-GKS.

le déroulement d'un appel de routine de binding est le suivant :

- on teste si GKS est installé et ouvert,
- on prépare le bloc de paramètres en entrée, en codant différemment les données si nécessaire,
- on écrit le bloc d'entrée, comprenant les paramètres et le numéro de fonction GKS appelée, dans le handle GKS.
- on teste les erreurs GKS, si cela est demandé par le programmeur, en lisant dans le Handle GKS un éventuel numéro d'erreur.Si la primitive reçoit des paramètres en sortie de GKS (requêtes, input ...), on lira simplement le bloc de paramètres en sortie, qui contiendra également, dans ce cas, le numéro d'erreur GKS détectée.

L'annexe II-2 de ce chapitre contient la définition des routine *OPEN-GKS*, *CLOSE-GKS*, *POLYLINE* et *REQUEST-LOCATOR* pour les langages ADA et C. Il faut noter qu'en C, conformément à la norme d'interface langage C, toutes les primitives sont implantées sous forme de fonctions, qui retournent le numéro d'erreur GKS détectée.

II.5.4.4. Synthèse et conclusion

Ce deuxième exemple montre ce qu'est une interface directe intermédiaire. On doit noter que ce modèle d'interface permet l'attaque du noyau GKS par plusieurs langages différents dans une même application, à condition de disposer d'un binding pour chacun d'eux, et de réserver l'ouverture et la fermeture du GKS à un seul langage. Essayons de résumer cet exemple.

Les langages des logiciels d'application utilisent une bibliothèque d'interface, spécialisée pour l'appel de la partie P2 ; c'est l'interface à P2 (IP2). Dans GKS 4.0, il s'agit du binding.

La bibliothèque spécialisée, qui obéit ici à des règles définies dans une norme, assure **le transfert et le codage des données** vers la partie P2, et fait un appel direct ou indirect, suivant l'implantation du langage, à un **mécanisme du système**, qui ici est l'interruption système int 21h, pour transférer le déroulement à P2, par l'intermédiaire d'une opération de lecture-écriture sur un File Handle MS-DOS (ce pourrait aussi être un fichier standard).

Ce mécanisme du système, comporte la **gestion d'un fichier ou d'une zone tampon** contenant les informations à transférer.

Cette zone tampon est manipulée par les deux parties. Ici, il s'agit du fichier (ou File Handle GKS) de nom "METAGKS4.DEV". La façon de lire et d'écrire dans cette zone est dictée par le système d'exploitation, qui dispose de fonctions standards pour réaliser cela. Il s'agit dans notre cas des fonctions système concernant les File Handle, fonctions qui sont fournies par MS-DOS.

La partie **P2 est "symétrique" de la partie P1**. Comme celle-ci, elle dispose d'une partie spécialisée destinée à gérer la zone tampon.

Dans notre exemple, le système ne permettant pas de faire communiquer deux processus indépendants, nous avons du modifier le système en redirigeant certains appels de fonctions MS-DOS. La partie spécialisée est donc ici constituée du système modifié, mais contient aussi la gestion de la zone tampon et le transfert des données dans celle-ci. La modification du système n'est pas obligatoire en général.

On peut penser qu'il y a peu de différences entre les interfaces séparées et les interfaces directes intermédiaire, mais ce qui fait qu'elles diffèrent, c'est le fait que les deux parties P1 et P2 sont liées, et la communication entre ces deux parties est donc facilitée. On notera toutefois que nous avons du pratiqué des modifications du système MS-DOS, chose à faire le moins souvent possible.

II.6. Interfaçage pour les implantations courantes des langages IA

Dans cette partie, nous passons en revue les implantations les plus courantes de langages IA, et pour chacune d'entre elles, nous évaluons les possibilités d'interfaçage logiciel.

Remarque : il est possible que les versions d'implantations étudiées ici ne soient pas les plus récentes, et donc que des affirmations ne soient plus toutes justifiées aujourd'hui; pour les langages LISP et PROLOG en particulier, dont les implantations étudiées datent de 86 en général, et dont certaines ont été modifiées depuis.

II.6.1. Cas du langage LE-LISP 15.2

II.6.1.1. Quelques mots sur le langage LISP

LE-LISP est un dialecte LISP conçu par J.CHAILLOUX [Chailloux 85b]. Nous nous intéresserons aux versions existantes sur gros systèmes (MULTICS), et sur micro-ordinateurs (MS-DOS). Nous allons voir les possibilités d'appel à des sous-programmes externes par ces deux implantations.

LE-LISP est écrit dans le langage LLM3, conçu par J. CHAILLOUX, afin de faciliter son portage sur tout type de système ou matériel [Chailloux 85a]. Le langage LLM3 est en quelque sorte un langage assembleur, adapté à la description et la manipulation des objets LISP.

Ce langage peut manipuler des objets du type :

- Pointeurs
- Nombres entiers
- Nombres flottants
- Chaîne de caractères
- Vecteurs de pointeur

On accède aux symboles grâce à un pointeur. Le format des instructions est le suivant :

- Etiquettes, codop, op1.....opm, commentaire

On dispose de registres rapides A1....A4, PC et SP (accumulation, compteur ordinal, pointeur de piles). On dispose dans le langage LLM3 d'opérandes immédiats et d' opérandes mémoire, d'instructions de transfert de pointeurs, d'opérations de comparaison, et de contrôle.

Il existe, dans ce langage, de nombreuses instructions de gestion de pile, de gestion de mémoire, et de récupération d'espace mémoire ("garbage collector").

Enfin, on dispose d'instructions de manipulation de listes. Rappelons, à ce sujet, qu'en LISP la structure essentielle est la liste ; mais on peut aussi manipuler les symboles, les nombres, et faire du calcul arithmétique élémentaire.

On dispose donc, à un niveau élémentaire, de toutes les fonctions permettant de fabriquer une implantation du langage LE-LISP. En fait, Le langage LLM3 a pour but de rendre le noyau de LISP indépendant de la machine sur laquelle doit être implanté LE-LISP.

Pour porter le langage LE-LISP 15.2 sur un système d'exploitation, il faut reprendre les sources de LE-LISP écrites en LLM3, et les compiler en langage machine, s'il existe une implantation de LLM3 dans le langage de la machine cible [Devin 85].

Le portage de la machine virtuelle LLM3 à été réalisé, parmi d'autres systèmes, pour les systèmes MULTICS et UNIX. LE-LISP existe donc pour les machines utilisant ce système d'exploitation. D'autre part, la société ACT Informatique a réalisé le portage de ce langage, en collaboration avec l'INRIA, sur le système MS-DOS.

II.6.1.2. Particularités des objets LISP

La donnée de base de LISP est l'atome. C'est une donnée élémentaire, comme par exemple un entier ou une chaîne de caractères.

Le deuxième élément de base et le plus important est la liste. LISP gère d'une manière transparente au programmeur n'importe quel type de liste. Une liste peut contenir des données ou du code. Un programme ou une fonction LISP est en fait une liste. Cette particularité fait que ce langage permet la manipulation symbolique des données ou du code.

Les concepts LISP se sont stabilisés dans les années 60 et présentent quelques caractéristiques proche des langages procéduraux. La déclaration et l'utilisation d'une fonction sont finalement assez similaires à ce que l'on trouve dans un langage procédural exception faite du typage des données qui n'existe pas en LISP.

On peut donc envisager d'interfacer assez facilement des sous-programmes procéduraux avec des sous-programmes LISP, le problème principal que l'on aura à résoudre étant la correspondance d'implantation en machine des données de base : entiers, réels, chaîne de caractères, ... Ceci veut dire que LISP est capable d'appeler un sous-programme procédural externe. L'inverse n'est pas vrai.

Les données manipulées en CAO ou avec le langage graphique GKS sont essentiellement numériques ou appartiennent à un type de base existant en LISP. Mais pour faire de la manipulation symbolique ou utiliser des techniques d'intelligence artificielle, on a besoin de structures de données beaucoup plus riches que ces types simples, et en particulier d'une nature éminemment dynamique.

Si l'on peut donc dans une partie écrite en LISP manipuler et traiter facilement des informations en provenance d'un système graphique ou de CAO, il n'en sera pas de même dans le cas inverse où un sous-programme procédural devrait récupérer et traiter des informations dynamiques (listes, arbres, graphes) en provenance d'un sous-programme LISP. Le programmeur de la partie LISP devra faire en sorte que les données transmises soient de type suffisamment simple pour être traitées par un sous-programme écrit en langage procédural. Il y a donc une forte probabilité pour qu'une interface de transformation et transmission des données à la partie procédurale soit à envisager.

LISP permet un interfaçage de base facile avec un langage procédural mais la nature des informations traitées en IA nécessite une préparation de celles-ci en vue de leur communication à la partie procédurale. Les données de base manipulées par les parties IA écrites en LISP et les parties CAO ou graphiques écrites en langage procédural diffèrent essentiellement par leur aspect dynamique et leur richesse structurelle.

II.6.1.3. Principe d'interfaçage avec LE-LISP

Le mécanisme logiciel d'interfaçage disponible en LE-LISP consiste en des déclarations de sous-programmes externes à LE-LISP, avec la primitive DEFEXTERN. On peut passer toutes sortes d'arguments à ces sous-programmes (entiers, flottants, chaînes de caractères, vecteurs d'objets LISP, objets quelconques), de même on peut recevoir tous types d'objets provenant de sous-programmes externes ; mais il faudra assurer que les données ainsi communiquées soient manipulées correctement par les deux parties, en particulier assurer le transfert et la préparation des données suivant leurs types (ex : chaîne de caractères LISP et chaîne de caractères C).

Pour pouvoir appeler un sous-programme externe, déclaré en LISP par "DEFEXTERN", nous disposons de 2 méthodes [Chailloux 85b].

- 1) Lier le code compilé de la fonction externe et le noyau de l'interprète LE-LISP en un même programme, de façon statique.
- 2) Réaliser la même chose par une opération de lien dynamique.

Que ce soit pour la première ou la deuxième méthode, on pourra utiliser la nouvelle fonction normalement avec l'interprète LISP modifié. Si elle est plus sûre et procure une image mémoire statique de l'interprète (intéressante pour les programmes finis), l'édition de lien sta-

tique est lourde, car elle nécessite parfois une recompilation de certains modules du noyau LE-LISP, quand on modifie des sous-programmes externes. Ceci n'est pas très adapté au développement d'applications (pas de prototypage aisé).

Par contre l'édition de liens dynamiques est plus pratique car elle peut être utilisée, quand c'est possible, avec le système d'exploitation cible, directement dans l'interprète LISP ; l'une des fonctions LISP de l'application doit appeler le compilateur ('?cc'), qui transforme le source de la fonction externe en code objet. On lie ensuite ce module compilé à l'interprète standard LE-LISP, en utilisant la primitive "CLOAD" du système.

Exemple :

```
$LE-LISP
? ; on compile un module écrit en C : 'POLYGONE.C', en créant le binaire POLYGONE.O
? ! cc -c POLYGONE.C
= t
? ; on lie ce module à l'interprète
?(cload "POLYGONE.O)
= POLYGONE.O
```

Avec cette méthode, les modules externes peuvent être conservés dans les images mémoires ; le lien dynamique est donc aussi durable que le lien statique.

Exemple d'utilisation de *DEFEXTERN*

code source en C :

```
window *make-window (x, y, l, h)
        int x, y, l, h
{...
        /* crée l'objet structuré fenêtre et rend un pointeur sur cet objet */
...}
select-window(w)
window *w
{...
        /* recoit un pointeur sur un objet du type fenêtre */
...}
```

code source correspondant en LE-LISP :

```
? (defextern _Create-window (fix, fix, fix, fix) external)
= _Create-window
? (defextern _Select-window (*) external)
```

```
= _Select-window
?
? (setq a (_Create-window 10 10 400 400))
= (12. 2348) ; c'est un pointeur manipulable en LISP
? ; repassons le pointeur à la partie C:
? (_Select-window a)
= 0
```

II.6.1.4. Commentaires sur le mécanisme d'interfaçage LE-LISP

La fonction LE-LISP "DEFEXTERN" est définie par :

(de DEFEXTERN symb ltype type)

L'argument "*type*" permet d'utiliser des fonctions externes à LISP, ou à attributs particuliers. Dans cet exemple "*external*" est l'argument *type* pour _Create-window.

Grâce au contrôle qu'effectue LE-LISP sur les valeurs *ltype*, ce système d'interfaçage comporte un contrôle primitif, et une manipulation interne de types de données communiquées au C, et inversement.

Malheureusement, dans l'implantation MS-DOS de LE-LISP [LE-LISP2.15 85], ces possibilités n'existent pas : pas de compilateur, d'éditeur de liens dynamiques, pas de chargement CLOAD, et pas de liaison possible avec DEFEXTERN.

Il faut donc envisager un autre système d'interface. Cependant, cette implantation permet, par la primitive COMLINE (*strg*), d'envoyer au système d'exploitation toute la ligne de commande définie par le contenu pointé par *strg* (le macro-caractère "!" est équivalent à COMLINE).

Nous citons cette primitive LE-LISP car c'est sans doute la seule permettant d'accéder à des sous-programmes externes dans le système MS-DOS.

On peut en effet fabriquer avec d'autres outils fournis sur ce système (éditeur de liens, compilateurs spécifiques...), des programmes exécutables externes contenant les fonctions que l'on veut appeler de LE-LISP. Ici, il faudra fabriquer complètement le contrôle de types de données, les parties LISP et la partie externe devront exercer ce contrôle indépendamment l'une de l'autre. Ne disposant pas de moyen système de connexion, ce genre d'interface ne peut être que du type indirect.

Nous pouvons envisager le schéma de fabrication d'interface spécifique à une fonction externe de la manière suivante :

- Cahier des charges de la fonction externe
- Conception-fabrication de cette fonction dans un langage algorithmique choisi
- Réalisation de la partie LISP : le nom de la fonction

  écriture d'une partie qui construit le nom de la fonction, et les paramètres à transmettre (in), ainsi que ceux à lire au retour (out)

  appel de COMLINE 'nom programme' contenant la fonction externe

  au retour de COMLINE lecture des paramètres en retour dans un fichier "out"
- Réalisation du programme externe exécutable :

  lecture du fichier d'appel "in", et activation de la fonction

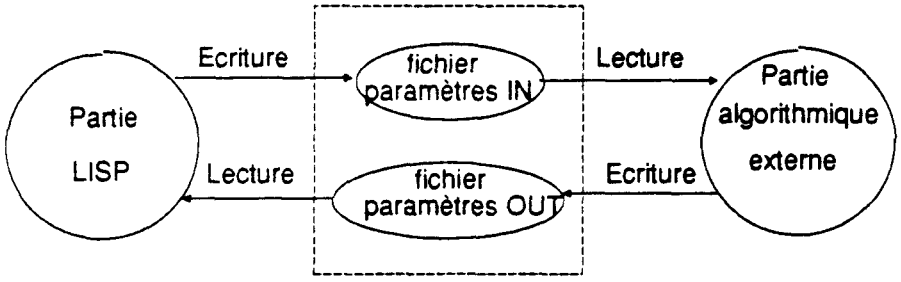
  vérification et contrôle des paramètres

  exécution de la fonction avec écriture éventuelle du fichier des paramètres en retour (out)

  fin du programme exécutable.

Le schéma des parties existantes dans ce type d'interface est le suivant :

Le contrôle de type se fait ici à l'écriture ou à la lecture des fichiers paramétrés.



L'interface logicielle LELISP avec "COMLINE", sous MS-DOS

Fig II-20

Mais ceci a de nombreux inconvénients :

- impossibilité de faire coexister partie IA et partie algorithmique.
- la partie IA doit piloter la partie algorithmique ; conception du système au niveau langage IA seulement.
- coûteux en temps, en mémoire vive utilisée, en mémoire de masse.

- utilisation d'objets lisp par la partie externe impossible. On ne peut le faire que pour des types élémentaires.

Le seul avantage est que le contrôle des types de données est simple, car les données transmises sont simples, et qu'il est effectué par les parties concernées par ces données (lecture des paramètres IN ou OUT sur un fichier).

II.6.1.5. Conclusion pour LE-LISP

On voit, avec le cas de LE-LISP 15.2, que le système d'exploitation et l'implantation du langage sur ce système sont déterminant, du moins en ce qui concerne la communication de données entre les deux langages, ainsi que pour la souplesse de réalisation d'interfaces.

On peut dire que quand le langage ne dispose pas d'outil logiciel permettant de faire des interfaces, on rencontre beaucoup de problèmes dans la réalisation de celles-ci. D'autre part, le seul type d'interface réalisable sera indirect, et il faudra s'assurer que l'on sait faire dérouler les deux parties l'une après l'autre, en les déchargeant ou non de la mémoire.

II.6.2. Cas de VAXLISP

VAXLISP 1.1 est une implantation de COMMON LISP [Steele Jr 84] réalisée par Digital Equipement Corporation.

II.6.2.1. Principe d'interfaçage avec VAXLISP

Ce dialecte permet d'appeler des routines externes, par des moyens tout à fait similaires à ceux de LE-LISP. Dans ce langage, on dispose de deux outils utiles à l'interfaçage :

- La primitive *DEFINE_EXTERNAL_ROUTINE* qui permet de définir la fonction externe vue du système VAXLISP (noms, paramètres, accès et contrôle de type...).
- La fonction *CALL-OUT*, qui permet d'activer le sous-programme externe comme une fonction LISP, mais en précisant le nom de la routine externe.

Exemple :

1 : On écrit en FORTRAN le fichier source suivant :

```
function NUMBERS (X,Y)
      IMPLICIT INTEGER*4 (A-Z)
      NUMBERS = Y * (X + Y**X) / X
      RETURN
END
```

Cette fonction FORTRAN manipule les entiers X et Y et retourne la valeur de $Y / X * (X + Y^X)$

2 : On compile et on fait l'édition de liens, pour obtenir un fichier VAX binaire.

3 : Sous interprète VAXLISP, on écrira :

```
?(DEFINE-EXTERNAL-ROUTINE
(NUMBERS : IMAGE-NAME "DBA2 : NUMBERS-FUNCTION"
: RESULT INTEGER, XY))
```

?NUMBERS

Ceci définit l'accès par LISP à cette fonction externe. Elle s'appellera "NUMBERS", enverra deux entiers comme arguments, et recevra un entier comme résultat.

4 : On peut alors appeler en LISP cette nouvelle fonction :

(CALL-OUT NUMBERS 5 7)

23536

II.6.2.2. Commentaires sur le mécanisme d'interfaçage de VAXLISP

On remarquera que, à la différence de LE-LISP, on doit passer par une primitive *CALL-OUT* du système VAXLISP, qui permet de faire les vérifications et les contrôles sur le nom et les types de données si cela est désiré. La primitive VAXLISP *DEFINE-EXTERNAL-ROUTINE* dispose de fonctionnalités intéressantes, pour le contrôle de la fonction externe et de ses arguments, lors de son appel :

- *Status Return Checking* (vérification de l'état du registre d'erreur au retour au programme appelant).
- *Entry Point* (point d'entrée); il peut être différent du nom de routine appelée.
- *Image Name* (fichier contenant le code exécutable de la routine externe).
- *Result* (précise les types attendus par LISP au retour de l'appel).
- *Type Checking* (Vérificateur de type entre paramètres formels et paramètres effectifs).

De plus, pour chaque paramètre formel, on peut préciser les attributs suivant :

- *Access* (accès en mode in ou in-out)
- *Lisp Type* (le type d'objets lisp attendus comme paramètres effectifs)
- *Passing Mechanism* (passage de paramètres par valeur, par nom (référence), ou par référence du descripteur).
- *Vax Data Type* (le type de donnée du paramètre effectif reçu par la procédure externe)

On peut dire que le mécanisme d'interfaçage système entre VAXLISP et des procédures externes, écrites en des langages respectant les conventions de passage de paramètres VAX/VMS, est tout à fait similaire à celui de LE-LISP. Cependant, sous VAXLISP, le contrôle sur les types de données envoyées et reçues par LISP est plus élaboré, plus riche que celui deLE-LISP.

Résumons ce mécanisme avec un schéma :

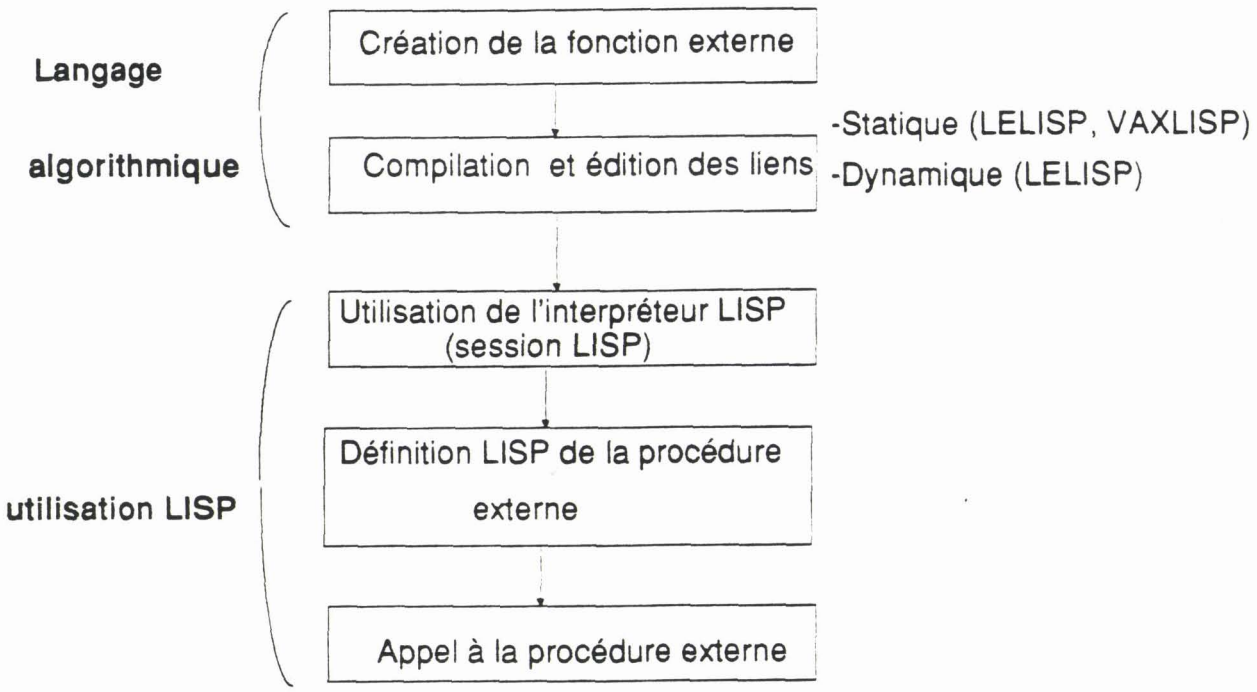


Fig II-21

Remarquons que VAXLISP dispose en plus d'un outil très utile pour les langages externes : les "Aliens structures" ou *structures étrangères*. Ces structures sont utiles pour manipuler dans LISP des types d'objets du langage algorithmique interfacé. Elles ressemblent en gros aux types structurés PASCAL. On les définit comme suit :

(Nom-de-structure-étrangère

(identificateur1 options valeur)

(identificateur2 options valeur))

par exemple on définira la structure étrangère suivante :

(DEFINE-ALIEN-STRUCTURE space

(Area-1 : UNSIGNED-INTEGER 0 4)

(Area-2 : UNSIGNED-INTEGER 4 8))

= SPACE

Les options suivantes permettent d'agir et de manipuler de façon spécifique ces structures étrangères :

- ':*CONC-NAME Access_function_name*'.

  LISP crée des fonctions d'accès au champs de structure dont le début des noms est précisé par CONC-NAME (ex : CONC-NAME GALAXY).

- ':*CONSTRUCTOR constructor_function_name*'

Ceci permet à l'utilisateur de définir le nom par lequel il peut créer un nouvel objet du type de la structure étrangère; par exemple :

```
?DEFINE-ALIEN-STRUCTURE SPACE (:CONSTRUCTOR create)
...
?(create SPACE :AREA-1 5 :AREA-2 10)
ALIEN-STRUCTURE SPACE #5x036E8
```

- ':*COPIER copier_fonction_name*'

Pour copier le contenu d'un objet de type structuré

- ':*PREDICATE function_name*'

Pour savoir si un champ,
désigné par [Alien-Structure-Name] [predicate-Function-Name] [nom-champ], existe ou non.

- ':*PRINT-FUNCTION*

Utilitaire qui permet d'envoyer une trace de la structure sur un canal de sortie, de l'imprimer.

On peut aussi, par un système similaire, préciser le type et l'accès aux champs de la structure étrangère, type et accès qui sont ceux classiquement utilisés dans les langages algorithmiques.

II.6.2.3. Conclusion pour VAXLISP

L'utilisateur peut, grâce à ces structures étrangères, utiliser des types du genre structure dans son programme application VAXLISP. Le système lui fournit de plus les outils pour contrôler les données manipulées dans ces structures.

Ce moyen, bien que pratique, n'est pas conseillé, car ce genre de manipulation est tout fait dangereux du point de vue effets de bord.

VAXLISP permet donc un interfaçage aisé, dicté à la fois par les outils du système d'exploitation et par l'implantation de COMMON-LISP (Alien structures [Steele Jr 84]. Les interfaces réalisées seront de type direct vrai, puisque VAXLISP fournit le mécanisme logiciel de connexion, et parce que le système VMS impose au concepteur de langages un protocole d'appel de routine standard, et qu'il gère lui-même les communications. Notons néanmoins que le contrôle des données communiquées n'est pas élaboré, et que de plus il peut s'avérer dangereux s'il est utilisé par des programmeurs peu rigoureux.

II.6.3. Cas des Implantations de PROLOG

II.6.3.1. Quelques mots sur le langage PROLOG

Nous étudions ici les possibilités d'interfaçage système avec des langages algorithmiques, pour les implantations de PROLOG.

Des études ont déjà été réalisées, pour incorporer des capacités d'interfaçage en PROLOG. Ceci est assez naturel, car on peut considérer les prédicats prolog comme une procédure à paramètres, du point de vue système ; cette procédure renvoie vrai si l'action s'est bien déroulée, et faux sinon. HUBNER et MARKOV, dans [Hubner & Markov 86], ont réalisé un interpréteur de dessins, en ayant tout d'abord interfacé PROLOG et GKS, par l'intermédiaire des pipes UNIX. Dans [Rueher & al 85], on enrichit prolog avec un environnement de modélisation et de manipulation d'objets géométriques, pour décharger le concepteur d'application prolog des parties strictement graphiques. Le logiciel est conçu à partir d'un interpréteur de commandes géométriques simples (affichage, opérateurs, transformation). Ce travail rejoint celui de HUBNER et MARKOV, en ce sens que dans les deux cas on construit un interpréteur d'objets géométriques, au moins pour l'affichage ; cependant on n'y précise pas les mécanismes système d'interfaçage.

C'est sur ces aspects système et logiciel de base que nous insistons surtout ici, pour les implantations suivantes de PROLOG :

- PROLOG II, CNET, Multics
- D-PROLOG MS-DOS
- Turbo-PROLOG MS-DOS
- succinctement PROLOG/P

II.6.3.2. Particularités des objets PROLOG

PROLOG est un langage déclaratif qui peut être considéré comme l'antithèse d'un langage procédural. Il est cependant possible de modéliser des objets graphiques ou de CAO avec ce langage [patk2d, acm-prolog, orchampt, favard].

L'inclusion dans le langage d'un système d'unification et de résolution implique qu'un grand nombre de données sont créées de manière automatique à l'insu de l'utilisateur ou du programmeur (résolution), et que l'instanciation des paramètres d'un prédicat évaluable (pouvant être vu comme un procédé standard d'interface avec un sous-programme externe) est de nature versatile (unification).

Du coup, un interfaçage de base nécessite un contrôle important sur les types de données transmises et une écriture explicite de toutes les formes d'instanciation des paramètres (prises par ceux-ci lors d'une unification) dans les interfaces aux sous-programmes procéduraux externes.

On retrouve donc dans le cas du langage PROLOG les même problèmes que l'on rencontre avec le langage LISP, avec une amplification des problèmes du à l'aspect très déclaratif du langage et à la génération automatique de données lors d'une exécution de programme.

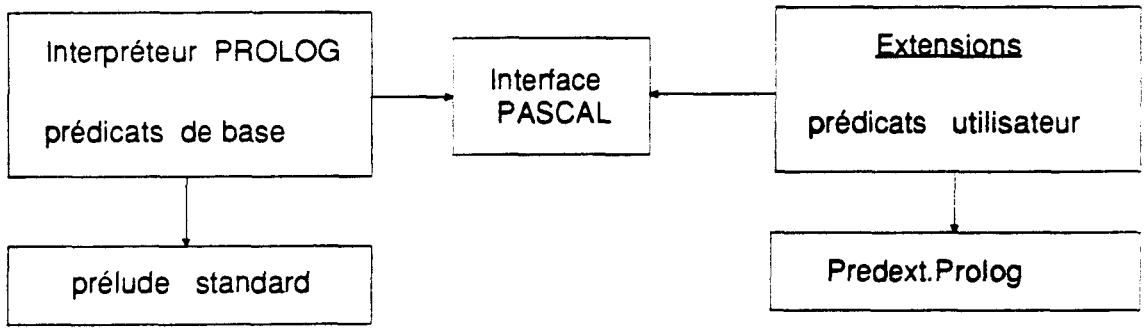
On entrevoit facilement la possibilité d'appel de services CAO ou graphiques ou de calcul par la partie I.A., mais encore une fois l'inverse va nécessiter des transformations de données importantes pour qu'elles soient consommables par la partie procédurale.

II.6.3.3. Cas de PROLOG II sous MULTICS

Cette implantation du langage PROLOG [Barbeyre & al 83], issue des travaux de l'équipe de A.COLMERAUER à Marseille-Luminy [Giannesini & al 85], et réalisé par le CNET, dispose des outils nécessaires à l'interface système avec PASCAL.

Présentation du mécanisme d'interfaçage :

L'interpréteur présente la structure suivante :



architecture de l'interface PROLOG/CNET sous MULTICS

Fig II-22

L'interpréteur est constitué d'un segment exécutable PROLOG auquel est associé la définition des prédicats contenus dans le segment "prélude.standard".

L'utilisateur peut enrichir les prédicats de base de son interpréteur PROLOG, en étendant le programme PREDEXT.PASCAL (Prédicats Externes Pascal) avec ses routines externes écrites en PASCAL. Il devra de plus fournir une définition externe PROLOG dans le segment PREDEXT.PROLOG qui est chargé automatiquement par le système.

Commentaires

Ici encore, l'utilisateur doit écrire un programme PASCAL, le compiler. Chaque modification des paramètres de routine externe , du point de vue déclaration et définition, implique une recompilation du segment exécutable PREDEXT.PASCAL et une modification du PREDEXT.PROLOG.

Cependant, le développement et la réalisation de la routine externe, ainsi que les corrections, ne nécessitent pas la modification de PREDEXT.PROLOG. Dans le cas présent, le concepteur de prédicat externe devra assurer lui-même les vérifications et contrôles nécessaires lors du passage des paramètres (types et valeurs). Enfin, ici encore, il faut envisager le système avec la partie en PROLOG II comme "maître" et la partie en PASCAL comme "esclave".

L'appel de PROLOG II par PASCAL peut se faire par l'intermédiaire des accès systèmes de PASCAL ; mais on ne pourra pas facilement activer une résolution (spécifique de PROLOG) à partir du langage PASCAL. Enfin, l'utilisateur devra lui même gérer ses liens entre PROLOG et PASCAL sans vérification par le système. L'interfaçage est une fois de plus du type direct vrai.

Conclusion

Dans cette implantation de PROLOG, on voit très nettement l'influence du système d'exploitation cible : l'interpréteur PROLOG II utilise le format protocolaire habituel de MULTICS (des "segments"), et l'utilisateur dispose de l'édition de liens dynamique. Il suffit donc au programmeur d'enrichir un segment exécutable, spécialisé pour l'appel de PASCAL et lié à l'interpréteur PROLOG, avec ses propres routines, qu'il implantera dans ce segment.

C'est bien le système d'exploitation, par ses par facilités de programmation offertes au programmeur, qui impose ici le mécanisme de l'interfaçage. Notons cependant qu'aucun mécanisme de contrôle de type n'est prévu, ce qui est une faiblesse du procédé ; car le créateur des ressources devra assurer, dans l'implantation de ses procédures externes, le contrôle des données transmises de l'interpréteur PROLOG II vers les routines PASCAL, et inversement.

II.6.3.4. Cas de D-PROLOG sous MS-DOS

Cette implantation de PROLOG CRISS [Donz & Hurtado 84], a été réalisée par Ph DONZ et Luc d'ARRAS à la société DELPHIA à GRENOBLE. Elle a été écrite successivement en Microsoft PASCAL (versions 3.20, puis 3.31), puis en Microsoft C (version 4.0, ...).

Présentation du mécanisme d'interfaçage

D-PROLOG dispose d'un prédicat prédéfini, qui permet à l'utilisateur du langage de définir un nouveau prédicat, dont l'effacement provoque une action, décrite par une routine écrite dans un autre langage. Vu le langage d'implantation de D-PROLOG, il est normal que ce langage soit MS-PASCAL. le prédicat prédéfini est :

alien (in-id, in-n)

On définit ainsi une primitive, ou encore prédicat externe, de nom *in-id*, dont le numéro d'ordre dans la liste des prédicats externes est *in-n.*

Exemple : 'alien (open-gks, 4)' permet de déclarer un nouveau prédicat externe *open-gks*, qui sera le quatrième de la liste des prédicats étrangers connus par l'interpréteur D-PROLOG.

Remarquons que l'utilisateur doit être sûr d'utiliser le bon numéro repérant le prédicat. On a ici une référence par position, plutôt que par nom. Avant de pouvoir utiliser ce prédicat, le programmeur devra réaliser la suite d'opérations suivantes :

- définir et implanter la procédure PASCAL représentant le prédicat externe
- l'incorporer au module de prédicats étrangers: "PL30PEX.PAS"
- compiler le module PL30PEX.PAS avec le compilateur MS PASCAL
- refaire l'éditions de liens du système D-PROLOG avec le module modifié.

Le module PL30PEX.PAS contient la procédure

CALLEXT (No-pred-externe: entier, vars Code-retour: entier).

Le programmeur devra insérer sa procédure MS-PASCAL comme un sous-programme, qui sera appelé par CALLEXT. Il dispose, entre autres, de toute une série de fonctions de contrôle, existantes déjà dans D-PROLOG, qui lui permettent de vérifier les types de données, et de contrôler les communications, entre le système D-PROLOG et le module objet MS-PASCAL qu'il vient d'écrire.

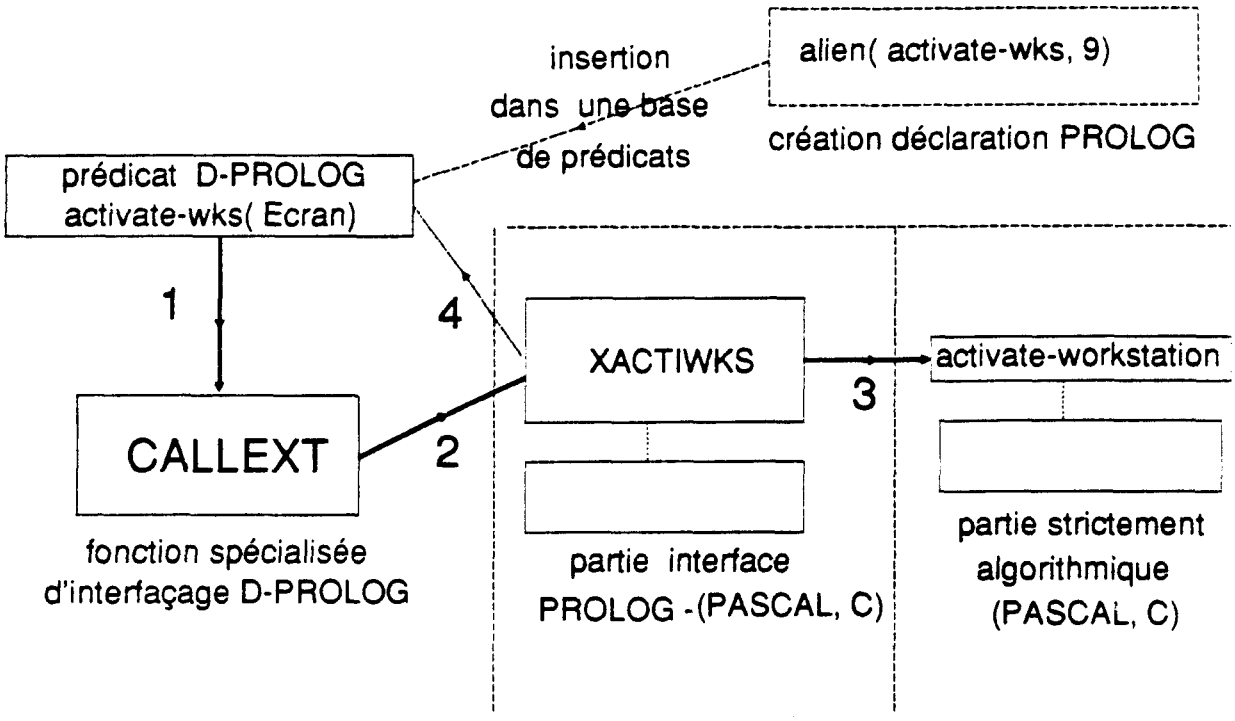
Exemple :

```
procedureXACTIWKS;                                  (* Activate Workstation *)
var
      {$include:'GKS_TYPS.DEC'}
      t,v : tindex;
      wkstyp : wks-type ;
begin
      If nbpar = 1 then begin
      getpar (1,t,v,);                              (* accès aux paramètres envoyés *)
      If isIdent(t) then
        If isWkstype(v, wkstyp) then                (* pas d'erreurs de paramètres *)
         activate_workstation(wkstyp)               (* on peut lancer la fonction externe *)
        else RC := 3                                (* v n'est pas du type espéré *)
      else RC := 3;                                 (* le type précisé n'existe pas *)
      end
      else rc := 2;                                 (* nombre d'arguments invalides *)
end; (* XACTIWKS *)
```

Cette procédure *XACTIWKS* doit être incluse dans le module PL30PEX. Le contrôle des types et valeurs est réalisé par les fonctions : *nbpar, isIdent, isWkstype* qui testent respectivement le nombre de paramètres passés lors de l'appel D-PROLOG du prédicat externe, l'identificateur de la variable v, et le type de la variable v.

Le schéma II-23 montre le déroulement d'un appel :

- 1) *alien(activate-wks, 9)* est la déclaration d'un prédicat externe D-PROLOG, de nom "activate-wks", qui est le neuvième prédicat externe disponible dans le noyau PROLOG.
- 2) lors de l'effacement de "activate-wks(ECRAN)", l'interpréteur D-PROLOG appelle la procédure spécialisée CALLEXT du noyau, avec 9 comme numéro de procédure à exécuter, qui est la procédure *ACTIWKS* dont nous venons de parler. *CALLEXT* vérifie l'existence d'un tel prédicat, et appelle la procédure correspondante ensuite.
- 3) *XACTIWKS* , après réalisation des contrôles éventuels, fait l'appel *activate_workstation(screen)*, sous-programme contenu dans le noyau GKS, ou dans l'interface langage.



Création, utilisation, déroulement d'un prédicat "étranger" en D-PROLOG

Fig II-23

Commentaires

Ainsi, l'utilisateur peut, quand il écrit sa procédure XACTIWKS, contrôler la validité des données communiquées entre D-PROLOG et MS-PASCAL.

avantages :

- interfaçage relativement facile.
- contrôle automatique, ou à mise en fonction simple, des paramètres, de leur type, de leur nombre, par l'interpréteur D-PROLOG.

inconvénients :

- lourd.
- réédition statique des liens à chaque modification.
- consommation de place mémoire et de temps CPU (le système MS-DOS est très vite saturé).

Pour obtenir un nouveau noyau D-PROLOG [D-PROLOG 85b], interfacé avec le prédicat défini par l'utilisateur, nous avons vu qu'une édition de liens **statique** avec les utilitaires propres à MS-DOS (éditeur de liens "LINK.EXE"), était nécessaire. Cet aspect statique de la modification rend pénible l'ajout et la modification des prédicats externes ; le prototypage est par ailleurs difficile.

Remarquons que les derniers problèmes cités sont surtout dus au fait que MS-DOS est un système d'exploitation pauvre, qui ne dispose pas de l'édition de liens dynamique, et dont la gestion de mémoire vive est limitée.

L'interfaçage logiciel entre D-PROLOG et MS-PASCAL est bien conçu, vis à vis du système d'exploitation sur lequel il est implanté. On remarque une similitude avec PROLOG II MULTICS, qui lui a l'avantage de disposer de l'édition dynamique des liens, mais qui par contre ne propose aucun contrôle de base sur les types de données transmises.

A titre d'exemple, voici un programme D-PROLOG, utilisant les primitives MS-PASCAL de METADESIGN GKS 3.0, qui trace un cadre sur un écran.

```
go :
        load(bank, gks) & GKS &
        open-gks &                                  (*ouverture GKS*)
        set-window(1,100.0,300.0,100.0,200.0)&      (*positionner fenêtre utilisateur *)
        open-wks(screen, outin) &                   (* ouvre une station écran *)
        activate-wks(screen) &                      (* activation de cette station *)
        set-wks-viewport( screen, 100, 300.0, 100.0, 200.0) &
                                                    (* définit la clôture GKS de l'écran *)
        tracer-cadre &                              (* tracer un cadre *)
        sortie.                                     (* fin session GKS et sortie *)
tracer cadre :
        set-linewidth( 1.0) &                       (* sélectionne l'épaisseur de trait *)
        set-linetype( 1) &                          (* sélectionne le type de trait continu *)
        set-polyline-color-index( 1) &              (* sélectionne la couleur des traits *)
        send-points-to-gks( 100.0, 100.0, 100.0, 300.0,
            200.0, 300.0, 200.0, 100.0, 100.0,100.0) &   (* envoi des points définissant un cadre *)
        polyline( 5).
                                                    (tracé du polygone définit par les cinq points du bu

sortie :
        clear-wks( screen) &                        (* efface l'écran graphique *)
        deactivate-wks( screen) &                   (* termine la session GKS *)
        close-wks( screen) &
        close-gks.
```

Nous pouvons observer, grâce à cet exemple, que le mécanisme proposé permet à l'utilisateur de D-PROLOG d'utiliser les mêmes noms que dans la partie procédurale, et de pouvoir utiliser les prédicats externes d'une manière tout à fait appropriée à son application.

Conclusion :

L'implantation, par DELPHIA, de PROLOG nous semble être un bon exemple de méthode rigoureuse d'interfaçage logiciel. Les interfaces sont du type direct vrai. Le réalisateur de l'interface dispose des outils nécessaires au contrôle des données communiquées, et peut de plus enrichir ce contrôle efficacement. On pourra bien sûr reprocher au mécanisme d'interfaçage sa lourdeur, à cause de l'édition des liens statique, mais ceci est dû au système MS-DOS.

L'annexe II-3 contient les sources d'une interface D-PROLOG - METADESIGN GKS3.0, ainsi qu'un programme d'exemple associé.

II.6.3.5. Cas de Turbo-PROLOG sous MS-DOS

Turbo-PROLOG est une implantation de PROLOG, sur MS-DOS, dont l'éditeur est la société américaine BORLAND. Les premiers prototypes datent de 1984. La syntaxe est différente des implantations PROLOG standards [Mellish & Clocksin 81, Giannesini 85], par la structure de présentation des programmes (déclarations initiales de types, de prédicats et de clauses), et par la syntaxe d'écriture des règles (if, and...).

Présentation du mécanisme d'interfaçage

Le système Turbo-PROLOG permet théoriquement, du moins d'après le manuel, d'interfacer des modules PROLOG avec des procédures écrites dans un autre langage quelconque, sous MS-DOS. Les langages auxquels il est fait référence sont essentiellement C, PASCAL, et l'assembleur du 8086.

Le schéma de construction d'un prédicat externe est classique :

a) déclaration de prédicat externe comme prédicat global Turbo-PROLOG, en précisant le nom du prédicat, le type des arguments passés, leur mode de passage ou d'utilisation pour l'unification (in, out), et le langage externe utilisé pour l'écriture du prédicat.

Exemple :

```
global prédicates
    _add (integer, integer, integer) - (i,i,o), (i,i,i) language C
    scanner (string, token) - (i,o) language PASCAL
```

Remarque : on doit préciser les **modèles de passage de données**, appelés **"flow-pattern"** en Turbo-PROLOG, que le langage sera "autorisé" à unifier, lors de déclenchement de régles ou d'effacement de clauses. Par exemple, pour le prédicat externe _add, le "flow pattern" (i,i,o) indique que les deux premiers paramètres sont unifiables en "entrée" par PROLOG, et que le dernier est unifiable en sortie. Ceci permet un certain contrôle de ce que le langage peut créer comme appel, et limite

par la même occasion le nombre de prédicats externes à écrire, car chaque modèle de passage cité implique obligatoirement l'existence du prédicat externe correspondant. Par exemple, il faudra créer deux procédures C pour faire fonctionner le prédicat add cité en exemple :

_add1(int a, int b, int *r) ; les 2 premiers paramètres sont en entrée,
le dernier en sortie.

et

_add2(int a, int b, int c); les 3 paramètres sont en entrée.

b) écriture et compilation des sous-programmes à interfacer, dans le langage externe utilisé.

c) édition de liens visant à obtenir un noyau Turbo-PROLOG qui contient le code exécutable de ces procédures externes. On utilise l'éditeur de liens statique de MS-DOS.

d) l'utilisation des prédicats déclarés en a) est alors possible, avec l'interpréteur Turbo-PROLOG modifié, ou directement en créant un programme exécutable MS-DOS qui contient le noyau de Turbo-PROLOG et la partie interfacée.

Commentaires

Notons que l'interfaçage décrit ici n'est possible qu'à partir de la version 2.0 de Turbo-PROLOG, les versions précédentes fournies par la société BORLAND ne fonctionnant pas de ce point de vue.

Malheureusement, ce procédé est très lié à la façon dont Turbo-PROLOG utilise le système MS-DOS. Par exemple, nous avons voulu suivre la procédure décrite dans le manuel pour interfacer une procédure MS-PASCAL qui reçoit un entier, et le renvoie doublé :

procedure double_0 (in-integer : integer ; var out-double-result : integer);

qui correspond à la déclaration suivante en Turbo-PROLOG :

global-predicates double (integer, integer) - (i,o) language PASCAL.

L'édition de liens n'a jamais pu être faite correctement, pour deux raisons : la première a déjà été citée précédemment, à savoir que les premières versions de cette implantation de PROLOG ne fonctionnaient pas pour l'interfaçage (bibliothèque PROLOG.LIB incomplète), la deuxième raison étant que le code d'initialisation des programmes PROLOG est en conflit avec le modèle de fonctionnement des programmes MS-PASCAL (pas le même modèle de mémoire, utilisation des registres 8086 différente). En effet, le langage Turbo-PROLOG suppose que le langage externe sauve le contexte Turbo-PROLOG dans ses moindres détails, avant de dérouler son propre algorithme, ce que ne fait pas le langage interfacé, sauf d'autres langages de BORLAND, comme Turbo-C.

Ceci est dû à l'obligation que subissent les implanteurs de langage sur MS-DOS, de créer leur propre code d'initialisation du programme, sans pouvoir garantir aucune compatibilité avec des langages d'autres éditeurs de logiciel.

On comprend alors aisément que les modules objets Turbo-PROLOG et MS-PASCAL sont totalement incompatibles, et que l'édition des liens ne puisse être réalisée correctement.

Notons enfin que la même procédure réalisée directement en assembleur 8086, en prenant garde de sauvegarder totalement le contexte Turbo-PROLOG à l'entrée de la procédure, c'est à dire les registres du 8086 pour MS-DOS, a pu fonctionner sans problèmes. Voici cette procédure assembleur interfacée avec Turbo-PROLOG, et le code PROLOG correspondant :

```
trace
global predicates
        double(integer,integer) -
                (i,o) language pascal
goal
        readint(INVAR),
        double(INVAR,OUTVAR),nl,
        write(OUTVAR).
```

```
;       name    ESCAL_INTERFACE_TBPROLOG_ASSEMBLEUR

;dgroup GROUP DATA
;assume DS:dgroup,SS:dgroup

assume  DS:nothing,SS:nothing

CODE_DBL segment 'CODE'
assume  CS:CODE_DBL

;-----------------------------------------
; procedure DOUBLE_C(
;                 IN,AF : integer;
;           vars OUTVAR : integer):(double);
;-----------------------------------------
        public  DOUBLE_C
DOUBLE_C proc far
        push BP
        mov BP,SP
        mov AX,[BP+6]
        add AX,AX
        les DI,d[bp]
        mov es:[di],AX
        pop BP
        ret 6
DOUBLE_C endp
CODE_DBL ends
end
```

Conclusion :

L'interfaçage réalisé est une fois de plus du type direct vrai (outil logiciel + outil système standard). Il ne permet pas de contrôle des données, mis à part pour des types simples, comme entier, réel, caractère, chaîne, et symbol (objet du même genre que chaîne), car ce sont les "types de base" de Turbo_PROLOG.

Les objets "compound" n'étant pas décrits du point de vue de leur structure interne, on a beaucoup de mal à effectuer un contrôle sur ces objets.

Enfin, la structure de liste manipulée en Turbo-PROLOG n'est pas décrite ; il faut alors procéder par succession de tentatives du type "essais-erreurs", avant de pouvoir comprendre la structure et les manipulations que fait Turbo-PROLOG sur ces objets dynamiques.

Remarquons de plus que ce procédé d'interfaçage ne fonctionne aussi que dans un seul sens: Turbo-PROLOG appelle un autre langage. Le manuel du langage Turbo-C (TB-C 87) prétend que l'on peut mélanger les deux langages, mais en fait la marche à suivre proposée se résume à :

- écrire les parties dites de "haut niveau", ou de contrôle, en PROLOG, écrire les autres en C.
- et faire en sorte que le code source PROLOG appelle la partie C au bon moment.

Il n'existe donc pas, à proprement parler, de coopération entre les deux langages; les parties écrites en C sont "attachées" au programme principal PROLOG, et donc sont esclaves du déroulement de la partie PROLOG.

II.6.3.6. Conception particulière d'une interface Turbo-PROLOG MS-PASCAL

Comme nous l'avons dit au paragraphe II.5.1, nous avons tenté de résoudre ces problèmes techniques d'interfaçage, spécifique de MS-DOS, en réalisant une interface directe vraie, du genre "ressource résidante" telle que nous l'avons défini dans le paragraphe précité, entre Turbo-PROLOG et le langage MS-PASCAL, avec lequel est implantée la ressource externe supposée être GKS.

Du côté de la ressource résidante, le procédé a déjà été largement décrit précédemment (Cf II.5.3.). On doit simplement redéfinir la partie "interface pour ressource résidante", pour qu'elle soit adaptée à ce qui est créé comme "interface pour l'application". Par contre, beaucoup de choses sont modifiées sur ce point, car Turbo-PROLOG impose des contraintes importantes sur la transmission de données.

Chaque prédicat déclaré en PROLOG doit être écrit dans le langage externe, en autant de fois que l'indique le "flow pattern" de la clause "global predicates" du source PROLOG (Cf. II.6.3.5.).

La partie "interface pour l'application" doit obligatoirement être écrite en assembleur, en faisant très attention aux registres, car il ne faut pas créer de conflit avec le contexte d'exécution de PROLOG; ne pas "écraser" le contenu des registres SI,DI,SS,DS en particulier. De plus, nous n'avons pas trouvé de moyen d'automatiser la création de cette partie, à partir des déclarations du source PROLOG, comme c'était le cas de Turbo-Pascal et des langages Microsoft.

Le source PROLOG, écrit en Turbo-PROLOG, qui réaliserait le petit programme d'exemple, tel que nous l'avons décrit pour le cas de D-PROLOG (Cf. II.6.3.4), est quasiment identique à quelques points de détail près : mise en place des déclarations de type, variable, prédicat externe dans le contexte d'utilisation Turbo-PROLOG (Domains, Global Predicates, Goal), ainsi que le remplacement des appels de "load(bank,GKS)" et "GKS", par des inclusions de modules Turbo-PROLOG appropriés.

Notre interfaçage a les avantages suivants :

- développement indépendant de la ressource résidante (deux niveaux d'interface).
- l'interface est possible dans les deux sens, si on sait charger-décharger un programme Turbo-PROLOG, comme nous le faisons pour les ressources résidantes, et déclencher une interruption libre du système (nous n'avons pas tenté cela). L'éventail des outils système (utilisation des registres, appels de fonctions système, appels d'interruption,...) proposé par Turbo-PROLOG, permet à priori de réaliser ces manipulations.
- Maintenance, déverminage de la ressource résidante, sans influence sur l'interface (pas de fabrication d'une nouvelle interface).

Ses principaux inconvénients sont :

- contrôle sur le passage des types de données obligatoire, à réaliser par l'implanteur de la ressource résidante,
- refaire le programme de génération automatique d'interface pour l'application, pour chaque langage interfacé avec Turbo-PROLOG, ce qui est difficile à priori, vu le manque de renseignements sur le codage des données en Turbo-PROLOG,
- grosse consommation de mémoire vive, car on additionne deux contextes de langage sur un système ne pouvant gérer plus de 640 Ko, d'autant plus qu'il faut créer le programme PROLOG hors de l'environnement intégré.
- uniquement valable pour des implantations de langage sur MS-DOS (à cause des ressources systèmes utilisées par MS-DOS).

II.6.3.7. Cas de PROLOG/P sur MS-DOS

Cette version de PROLOG [PROLOG/P 85] a pour origine l'interpréteur PROLOG/CNET PASCAL/MULTICS V2.00, que nous avons décrite précédemment au paragraphe II.6.3.1, sous le nom de PROLOG/CNET. Elle est destinée au système MS-DOS, et l'interpréteur PROLOG est écrit avec le langage MS-PASCAL 3.20.

Le principe d'interfaçage est le même qu'avec PROLOG/CNET : l'utilisateur dispose du module "*predext.pas*", qu'il peut modifier pour ajouter un prédicat évaluable externe, écrit dans un langage Microsoft. Après compilation de *predext.pas* modifié, on réalise l'édition des liens statique avec l'éditeur MS-DOS, pour disposer d'un interpréteur PROLOG modifié, dans lequel le prédicat externe est utilisable.

Du point de vue du système d'exploitation, c'est à dire au niveau de l'interfaçage logiciel système, le mécanisme de communication est identique au D-PROLOG de DELPHIA (II.6.3.2). On dispose dans *predext.pas* d'une fonction "*predext*" (vs. procédure *callext* en D-PROLOG), à un seul argument : le numéro du prédicat externe à exécuter. Cette fonction retourne une valeur énumérée (vs. en D-PROLOG, un paramètre de retour) : le résultat de l'évaluation, qui peut être *vrai, faux, erroné.*

Par contre, la conception globale du contrôle de données est différente : le prédicat est défini en PROLOG à l'insertion du prélude standard "PRELSRC.STD", et l'utilisateur ne dispose pas d'outil particulier de contrôle ou de vérification, lors de la définition du prédicat externe en PASCAL, prédicat dont il doit insérer le code source dans la définition de la fonction *predext.*

En PROLOG/P, le mécanisme d'interfaçage est donc, du point de vue système, quasiment identique à celui de D-PROLOG (Cf schéma II-30), mais ne propose par contre aucun moyen de contrôle des données transmises de PROLOG/P vers PASCAL, et inversement ; il faut

connaître précisément le fonctionnement de l'interpréteur, pour pouvoir interfacer un prédicat externe. C'est un procédé "primitif", et sans élégance, dans lequel l'utilisateur doit finalement modifier "sans filet" l'interpréteur PROLOG/P.

II.6.3.8. Conclusion pour PROLOG

On peut réaliser dans les diverses implantations de PROLOG une interface avec un langage procédural par l'intermédiaire de prédicats évaluables qui peuvent être vus comme des procédures. La technique consiste à utiliser le prédicat évaluable à la manière d'un sous-programme écrit dans un langage externe.

Si l'implantation PROLOG est capable de communiquer avec un langage procédural ou si elle-même est écrite à base d'un langage procédural, on pourra implanter assez facilement une communication avec des prédicats évaluables.

Cependant, nous avons mis en lumière un problème propre au langage PROLOG, qui est en particulier clairement explicité par le constructeur de Turbo-PROLOG : la versatilité des types de paramètres instanciés dans un prédicat évaluable lors de son appel nécessite que l'implanteur du sous-programme externe teste dans le code le type d'instanciation courante des paramètres du prédicat. Suivant la position du prédicat dans le programme PROLOG, les paramètres peuvent être instanciés en des valeurs totalement inattendues par l'implanteur du sous-programme externe.

Le contrôle de type lors d'un appel de prédicat évaluable est donc un paramètre de réalisation de l'interfaçage dont l'importance est accrue en PROLOG. L'implantation la plus élégante et la plus solide d'une solution pour ce problème est celle que nous avons rencontrée dans Delphia-PROLOG, qui propose au programmeur une technique dotée d'un contrôle automatique de type des paramètres (incluant les valeurs indéfinies) lors de l'appel du prédicat évaluable.

II.6.4. Cas des implantations de SMALLTALK

II.6.4.1. Quelques mots sur le langage SMALLTALK

SMALLTALK est construit, du point de vue système, à partir d'une machine virtuelle utilisant les "bytecodes" [Goldberg & Robson 83, Krasner 83]. Quelques soient ses implantations, le texte source de ce langage est traduit (compilé) dans le langage de la machine virtuelle, puis ce code est interprété.

Il faut donc connaître en détails l'implantation de cette machine virtuelle, et la modifier directement pour interfacer un langage externe. Dans le cas de gros systèmes, on dispose de l'édition de liens dynamique qui facilite ce travail. Sur de petits systèmes, il faudra intervenir au niveau du code source de la machine virtuelle, du moins nous le pensons. On peut cependant, grâce aux méthodes "primitives" de SMALLTALK, en les écrivant dans un langage

procédural, réussir à interfacer SMALLTALK avec d'autres langages. Cependant, ce principe relève d'une méthode expérimentale, c'est à dire qu'il n'y aucune base théorique montrant le bien fondé de cette méthode, sauf la nécessité d'un efficacité maximale à l'exécution d'une méthode En particulier, il existe des problèmes d'adressage et de manipulation des objets [SMALLTALK/V 87] de l'environnement de l'interpréteur en cours d'utilisation, par le langage externe.

Regardons ce que nous proposent les implanteurs de ce langage, sur machine compatible PC et sur TEKTRONIX 4404.

II.6.4.2. Particularités des objets en SMALLTALK

Smalltalk représente à la fois un modèle objet de base, un environnement de programmation et de prototypage de logiciel, et enfin une système d'exploitation mono-utilisateur à part entière.

Toute donnée Smalltalk ne se conçoit qu'en terme d'une structure contenant des champs de type élémentaire (entiers, réels, ...) ou plus complexes (instances de classe). On peut alors penser qu'une correspondance entre objets de partie procédurale et objets Smalltalk soit facile à mettre en oeuvre, par exemple en définissant des classes où les champs des objets manipulés correspondent à des valeurs significatives de la partie procédurale (dimension d'un objet géométrique, caractéristiques fonctionnelles, ...), et les méthodes interfacées directement avec des sous-programmes externes en temps que méthodes primitives.

Malheureusement le défaut principal de Smalltalk à ce sujet est d'être fermé sur un environnement, et ce genre de correspondance n'est en fait pas facile à mettre en oeuvre. Une solution qui pourrait être envisagée pour faciliter l'intégration est l'utilisation d'un programme de transformation de code Smalltalk en code C, qui muni d'un système de communication associé doté d'un langage de description d'interface permettrait de relier d'une manière semi-automatique - par l'intermédiaire du programme de transformation de code Smalltalk et des déclarations sur les données procédurales - des parties écrites en Smalltalk avec des parties procédurales. Le système OpenTalk de ParcPlace est un premier exemple de ce que l'on peut envisager comme solution.

II.6.4.3. Cas de SMALLTALK/V

SMALLTALK/V [SMALLTALK/V 87) est une implantation partielle et modifiée, sur compatibles PC (système MS-DOS), de SMALLTALK-80. Cette implantation ne respecte pas exactement la hiérarchie des classes et leur contenu, tels qu'ils existent en SMALLTALK 80.

Dans ce langage, nous ne pouvons pas définir de classes "externes", ni créer des instances d' "objets externes". Par contre, nous pouvons créer des méthodes primitives (GOLD83) externes, pour des objets que nous avons auparavant définis et décrit en SMALLTALK/V. Elles sont appelées **"méthodes primitives définies par l'utilisateur"** (User defined

primitive methods). Leur but est de disposer d'implantations de méthodes plus efficace que la liste de "bytecodes" issus de la compilation du code source SMALLTALK d'un méthode réalisant la même fonction.

Rappelons la forme du code SMALLTALK d'une méthode utilisant la primitive utilisateur :

- 1 appel de la primitive utilisateur : <primitive : no>,
  ex : <primitive : 92>
- 2 partie Smalltalk exécutée si il y a échec de la primitive;
  ex: ^self checkIndex : integer.

Le concepteur du langage propose une méthode du même genre que nos "ressources résidantes" : la création de procédure externe que l'on charge en mémoire et que l'on ne décharge pas. Cependant, la procédure externe interfacée doit manipuler l'objet récepteur du message correspondant à la méthode, ce qui nécessite des manipulations difficiles, et très dépendantes de l'implantation du langage sur le système MS-DOS.

Pour simplifier la réalisation de méthodes primitives, les réalisateurs de SMALLTALK/V ont prévu un ensemble de "macros" en assembleur 8086 [MASM 85], car l'utilisateur est supposé écrire sa méthode externe en assembleur 8086, qui facilitent quelque peu la réalisation de l'interface. Ces macros doivent être insérées judicieusement dans le corps de la méthode primitive, écrite en assembleur.

enterPrimitive (qui doit être le début du code source)

exitWithSuccessnumberOfArguments, (pour sortir sans erreur)

exitWithFailurenumberOfArguments, (sortie en cas d'erreur)

getObjectAddress ObjectPtrReg, OffsetReg, segmentReg, (qui donne l'adresse mémoire sur 20 bit, au format MS-DOS, de l'objet récepteur du message)

getClass objectPtrReg, classPtrReg, workSegmentReg, (donne le pointeur de la classe de l'objet pointé)

markPtr objectPtrReg, worksegmentReg, (pour permettre au garbage-collector de marquer l'objet pointé)

isSizeEven objecPtrReg, workReg, workSegmentReg (pour savoir si on doit faire des repositionnement sur adresses paires en mémoire (exigé par MS-DOS)

Nous n'avons pas poussé très loin l'étude de cet interfaçage, par exemple jusqu'à interfacer SMALLTALK avec un noyau GKS. L'exemple fourni fonctionne bien, mais montre comme il est délicat d'interfacer Smalltalk avec un langage externe. Il faut remarquer que les structures de données manipulées ici sont très simples. Cette méthode est difficile à mettre en oeuvre, et limitée dans ses concepts, simplement destinée à créer des primitives particulières et spécifiques d'une classe, pour obtenir une meilleure efficacité.

Par l'intermédiaire de l'assembleur 8086, on peut envisager l'interfaçage avec un langage plus évolué, car le langage assembleur de Microsoft est "connectable" avec beaucoup d'implantations de langage existant sur le système MS-DOS.

II.6.4.4. Cas de SMALLTALK-80

Contrairement à ce que l'on pourrait attendre d'une implantation de SMALLTALK-80 sur un système du genre UNIX, la version de ce langage sur la machine TEKTRONIX 4404 ne permet aucun lien avec la machine virtuelle SMALLTALK, plus précisément avec la mémoire d'objet et l'interpréteur. D'autre part, rien de comparable aux "méthodes primitives d'utilisateur" de SMALLTALK/V n'existe pour cette implantation. De ce fait, la réalisation d'une interface logicielle avec un langage externe ne peut être réalisée au moyen de méthodes et routines primitives [Goldberg & Robson 83].

Néanmoins, nous avons cherché, et trouvé, un moyen d'interfacer SMALLTALK-80 avec l'implantation du langage C existant sur la machine TEKTRONIX 4404. Ce procédé est basé sur un modèle d'interface directe intermédiaire, et sur l'utilisation de "pipes" Unix. Nous décrivons ici la structure et le fonctionnement de cette interface.

Pour réaliser cette interface, nous avons utilisé principalement trois classes de SMALLTALK-80 : **Pipe**, **Subtask**, et **TekSystemCall** La méthode utilise la possibilité d'exécuter un fichier binaire (programme ou commande) externe à l'environnement SMALLTALK, en lui communiquant des arguments par l'intermédiaire d'un "pipe" (tube) Unix. Ceci est réalisé par une méthode de classe de **TekSystemCall** qui s'appelle : **ExecSystemUtility** commandName **withArgs** orderedCollection. Cette méthode utilise obligatoirement les descripteurs de fichier standards de sortie et d'erreur (StdOut (1) et StdErr (2)) [Rifflet 86], comme points d'entrée du pipe en lecture pour le programme externe, et suppose que le résultat du programme se trouve dans ce pipe après exécution et retour à l'environnement SMALLTALK. Le déroulement de la routine associée est le suivant :

- création du pipe, instance de **Pipe**,
- création d'un processus, à partir d'une instance de la classe **SubTask**,
- activation et déroulement du processus, attente de terminaison de son déroulement,
- "lecture" du coté lecture du pipe, et transmission de son contenu comme résultat, par une chaîne de caractères (instance de **String**), ou par un tableau de bytes (instance de **ByteArray**).

Voici le texte source correspondant à cette méthode :

```
execSystemUtility : aCommand withArgs : anOrderedCollection
    "permet de lancer un programme externe, communiquant avec Smalltalk par un pipe Unix,
    dont les descripteurs en lecture et écriture sont passés comme arguments du programme ex-
    terne"

    | pipe task inputSide resultOfProgram |
    pipe <--- Pipe new.
        fork: aCommand
        withArgs: anOrderedCollection
        then: [ pipe mapWriteTo: 1; mapWriteTo: 2.
              pipe closeWrite; closeRead ] .

    task start isNil
     ifTrue: [pipe closeWrite; closeRead.
         self error: 'Cannot execute', aCommand ].
    pipe closeWrite.

    Cursor execute
     showWhile: [inputSide <--- PipeReadStream openOn: pipe.
              resultOfProgram <--- inputSide contentsOfEntireFile ].
    task waitOn.
    inputSide close.
    task abnormalTermination
     ifTrue: [self error: 'Error from system utility: ',
         (resultOfProgram copyUpTo: Character cr) ].
    task release.
    ^resultOfProgram
```

Il est clair que l'on peut alors transmettre des arguments à un programme externe écrit en langage C. A titre d'exemple, voici le texte source de la méthode ++ , que nous avons rajoutée à la classe **Integer**, qui additionne deux entiers. Le résultat, écrit dans le pipe par le programme externe, et créé par la méthode précédente de SMALLTALK, est lu par celle-ci et transmis à la méthode ++ .

```
++ anInteger
"addition externe d'entiers par un programme écrit en C"
| retour |
retour <--- TekSystemCall
        execSystemUtility : 'addentiers'
        withArgs : (OrderedCollection
       with: ((self printStringRadix: 10),' ')
       with: ((anInteger printStringRadix: 10),' ')).
```

```
(retour isKindOf: String)
 ifTrue: [ ^retour asNumber ]
 ifFalse: [self error: 'le resultat attendu doit etre une chaine de caracteres' ].
```

D'autre part, le programme **addentiers**, ayant comme arguments deux entiers et les deux descripteurs d'entrée et de sortie du pipe, réalise effectivement l'addition de ces deux entiers, et écrit le résultat dans le descripteur d'entrée du pipe.

```
/* Programme C de connexion avec Smalltalk-80, utilisant un pipe (utilisant les descripteurs 1
* et 2 de stdout et stderr); Additionne deux entiers.
*    en Entrée : les deux entiers a additionner.
*    en Sortie : rien.
*   Particularité : renvoie le résultat de l'addition dans le descripteur en écriture du pipe */
#include    <stdio.h>
main (n, v)
 int    n ;
 char   *v[] ;
{int add, fdIn, fdOut ;
 char *tab ;

/* association des descripteurs au pipe cree par SMALLTALK */
 fdIn = atoi (v[1]) ;
 close (fdIn) ;              /* car on ne lira pas sur le pipe */
 fdOut = atoi (v[2]) ;

/* addition des deux entiers et ecriture sur le pipe */
 add = atoi (v[3]) + atoi (v[4]) ;
 tab = _itostr (val, 10, "0123456789", (int *) NULL) ;
 write (fdOut, tab, 5) ;
 close (fdOut) ;
}
```

On utilise cette nouvelle méthode '++' d'une manière tout à fait conventionnelle : 3 ++ 4 renvoie 7.

Même si ce procédé est moins dangereux et risqué que le principe utilisé dans SMALLTALK/V, surtout concernant la manipulation de l'environnement SMALLTALK, il est cependant certain que son inconvénient majeur est un manque d'efficacité, dû à :

- la création d'un processus Unix,
- l'activation d'une série de méthodes spécialisées dans la communication avec le milieu externe,
- la communication par un buffer de fichier.

En conclusion, on peut dire qu'une utilisation raisonnable de ce principe serait l'exécution d'un programme C de taille importante, concernant des calculs coûteux, ou la réalisation d'une tâche spécialisée, avec un langage différent de SMALLTALK.

II.6.4.5. Conclusion pour SMALLTALK

Les deux implantations de SMALLTALK étudiées ici, montrent qu'il n'existe aucun moyen pratique pour interfacer ce langage avec des langages procéduraux externes. On peut penser qu'un mécanisme particulier d'interfaçage, adapté à cette technique, comme par exemple la possibilité de définir des "classes externes", qui décriraient la structure des objets de celles-ci, mais qui utiliserait des routines externes (écrites en langage procédural) pour implanter les actions associées aux méthodes, faciliterait la réalisation d'une telle interface. Mais SMALLTALK est conçu à la fois comme un langage et comme un environnement fermé, ce qui explique en partie l'inexistence d'un tel procédé.

Nous avons vu que les réalisateurs prévoient tout de même un mécanisme spécifique à leur implantation : par exemple la possibilité pour SMALLTALK/V de créer ses propres méthodes primitives, intégrées à l'environnement (interfaçage de type direct vrai), ou encore la communication par un processus et un tube Unix en SMALLTALK-80 (interfaçage de type direct intermédiaire), mais chaque procédé est conçu comme une facilité supplémentaire offerte à l'utilisateur. En fait, il n'existe pas de modèle théorique "propre" et pratique, pour réaliser un tel interfaçage.

Ceci se comprend aisément lorsque l'on étudie avec soin l'implantation de SMALLTALK [Krasner 83], en particulier le fonctionnement de la machine virtuelle, avec le gestionnaire et récupérateur d'objets ("garbage-collector"), l'interpréteur de "bytecodes", et le compilateur de texte source SMALLTALK. A tout instant, les structures de données représentant les objets, par nature dynamiques, peuvent être modifiées ou déplacées en mémoire. Comme nous l'avons dit au début de ce chapitre (Cf II.2.4.), c'est un point gênant, en ce qui concerne les possibilités d'interfaçage avec d'autres langages. Pour pouvoir interfacer proprement et facilement SMALLTALK avec un langage externe, il faudrait disposer d'objets spécialisés, permettant de connaître l'adresse mémoire des objets à manipuler, de bloquer la récupération de mémoire (garbage-collection) pendant le déroulement du programme externe, de préparer des copies des structures d'objets adaptées et assimilables par le langage externe utilisé, d'activer directement, et sans partie lente, les routines externes. Ce qui mène bien sûr à une intervention au niveau de la machine virtuelle elle-même.

Même dans cette éventualité, le langage externe est l'esclave de SMALLTALK, et la communication inter-langage sera limitée. De plus, communiquer dans l'autre sens, une partie écrite en langage procédural utilisant l'environnement SMALLTALK, nous semble très difficile à réaliser, ne serait ce que par la richesse, la complexité, et la nature fermée des implantations de SMALLTALK.

Remarquons toutefois que si notre conclusion est négative pour le langage SMALLTALK, il existe une solution qui nous semble bonne pour les langages à objets en général, et qui existe implicitement dans des réalisations de langages hybrides (C++, Objective C). Elle consiste à mixer les modèles des langages à objets et des langages procéduraux; dès lors, par la coexistence de ces deux modèles au sein d'un même langage, la coopération entre eux est tout à fait possible, même avec des langages procéduraux différents de celui qui est inclus dans le langage hybride, car la communication entre deux langages procéduraux est finalement plus facile à réaliser qu'entre un langage spécialisé en I.A. et un langage procédural.

II.6.5. Cas de C++ et des langages hybrides

Les langages hybrides sont des sur-ensembles de langages particuliers, en général procéduraux, et intègrent des capacités de structuration orientée-objet. Les concepts de classe, d'instance, et d'héritage existent et permettent d'utiliser les notions de bases de la programmation orientée objet. On leur a en fait greffer une couche orientée objet.

Comme ces langages sont construits à partir de langages procéduraux, ils permettent la programmation procédurale du langage initial et permettent d'écrire directement du code du langage de base. Par exemple, on peut écrire du code C à l'intérieur d'une méthode C++.

On peut donc penser qu'il est facile de faire communiquer une partie écrite dans le langage de base avec une autre partie écrite avec le langage hybride. Si donc on implémente la partie IA avec le langage hybride et que l'on utilise le langage de base pour communiquer et transmettre des informations entre les deux parties, on aura résolu le problème de communication de base. Sachant cette fois ci que l'on dispose d'un langage à objets au niveau de la partie IA et donc d'une structuration importante, on devrait être capable d'intégrer d'une manière correcte les deux parties.

Malheureusement les langages hybrides bâtis au-dessus de langage procéduraux restent des langages fortement typés. Or le typage des données rend la manipulation symbolique difficile, car il rend impératif et statique la nature des données manipulées. Du coup, les mécanismes de base existant dans les techniques d'IA sont difficilement implémentables dans ces langages hybrides, car ces mécanismes nécessitent de gérer des données de nature dynamique et de type inconnu avant l'exécution du programme. En conclusion, il faut reconstruire dans le langage hybride un noyau adapté aux techniques d'IA. Cette façon de procéder a déjà été utilisée dans la fabrication de systèmes experts écrits en langage procédural comme PASCAL.

Cependant, il existe d'autre types de langages hybrides à objets, comme par exemple ceux construits au-dessus de dialectes LISP comme CLOS, FLAVORS, LOOPS, CEYX. Ces langages sont plus adaptés aux techniques IA puisqu'ils sont conçus à partir de langage de base en IA. Mais les systèmes de CAO en particulier et les logiciels procéduraux n'utilisent pas ce genre de langage.

Cependant, si l'on est prêt à réécrire les parties utilisant des techniques IA avec le langage hybride, et si ce langage hybride est bâti sur un langage procédural, on disposera d'avantages certains pour réaliser efficacement l'intégration et pour disposer d'un logiciel efficace.

En supposant donc que l'on réécrive un noyau permettant l'utilisation des techniques de programmation (moteur d'inférence, règles, démons et attachement procédural, techniques d'ordonnancement ...) et de représentation des connaissances (bases de connaissances à base de règles, représentation à base d'objets ou de *frames*, représentations multi-vues ...) issus de l'IA, on aura les avantages suivants pour fabriquer un système logiciel où les techniques de CAO et IA sont intégrées ensemble :

- Les objets informatiques issus des logiciels de CAO, souvent de type simple et bien connus (nombres entiers, réels, chaînes de caractères, tableaux ou vecteurs de nombres (pour représenter des matrices de points dans le cas de surfaces ou d'objets géométriques définis par leur arètes) seront aisément communicables dans les deux directions (de la partie procédurale vers la partie IA et inversement), car manipulés et implémentés en machine de la même manière.
- Le passage d'une structure plate (dans les logiciels procéduraux) à une structure élaborée (dans la partie hybride) se fera simplement, par création d'objets utilisant les données de la partie procédurale pour affecter correctement les attributs dans la partie hybride.
- La communication de données entre les deux parties, en particulier de nature dynamique, peut être réalisée plus facilement car on contrôle réellement dans les deux parties le type des données crées, autant d'une manière statique (données déclarées et créées explicitement dans les programmes) que d'une manière automatique (données crées et manipulées de façon automatique et dynamique par les parties IA).

C'est donc sans doute par le moyen d'une adaptation des langages que pourra réaliser le plus facilement et proprement l'intégration entre des logiciels de CAO et IA, malheureusement au prix d'un gros travail de réécriture des parties IA. Notons que certains logiciels de CAO récents (MARCH d'Intergraph, ICAD de McDermott) fabriqués à partir de langages hybrides mettent déjà en pratique certaines techniques IA, et permettent en particulier un enrichissement notable du modèle descriptif par des représentations multi-vues, permettant ainsi d'offrir à l'utilisateur des fonctions l'aidant réellement dans sa conception.

En conclusion, dans le cas des langages hybrides, on est confronté au dilemme suivant : soit ces langages sont implémentés au dessus d'un langage procédural et on pourra dans ce cas espérer que ce langage procédural communique aisément avec le langage procédural d'implantation du SCAO - mais dans ce cas il faudra construire un noyau permettant d'utiliser des techniques IA au-dessus du langage hybride - ou alors on disposera à priori d'un langage hybride permettant d'utiliser ou d'implémenter facilement les techniques IA en perdant dans ce cas l'avantage d'une communication aisée avec le langage procédural d'implantation du SCAO. Une solution éventuellement envisageable serait la réalisation d'un langage intégrant

les trois paradigmes mis en jeu : la programmation procédurale, la programmation par objet, et la programmation déclarative. Mais ceci implique la réécriture totale du système, aussi bien la partie IA que la partie CAO, et on ne peut plus parler dans ce cas d'interfaçage ni d'intégration.

II.6.6. Conclusion de cette étude de cas

Le schéma conceptuel des interfaces étudiées est le suivant :

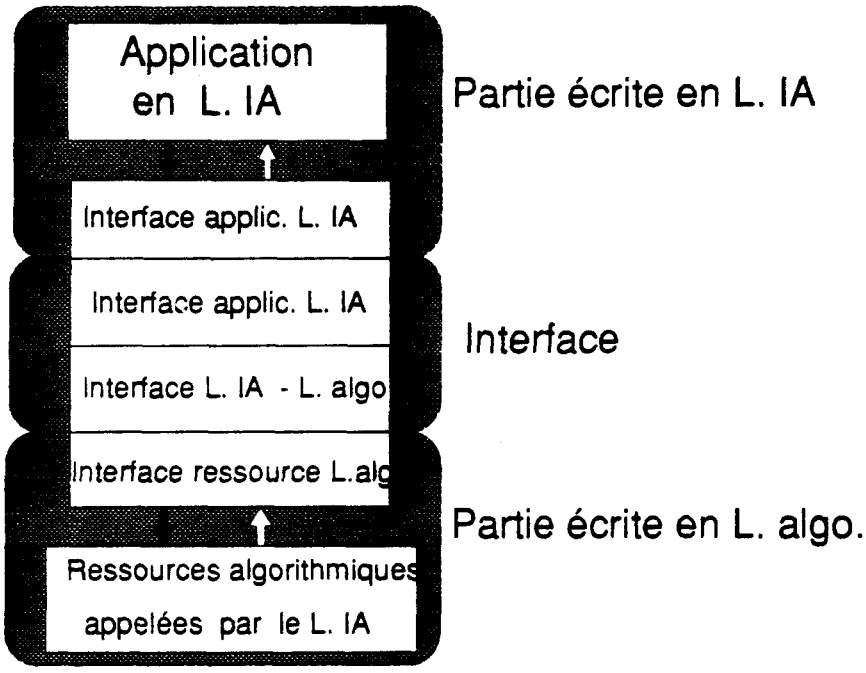


Fig II-24

Quelle que soit le type d'interface utilisé (séparé, direct faux, direct intermédiaire, direct vrai), les deux parties "*partie écrite en langage IA*" et "*partie écrite en langage algorithmique*" de la figure II-30 existent toujours. seule la partie "*interface*" est conçue différemment suivant le type d'interface. En fait, cette dernière partie varie beaucoup suivant les réalisations effectives. Une constatation importante à faire est que l'interface pour l'application en langage IA peut être importante, et qu'on sera souvent obligé de créer des parties logicielles destinées à préparer les informations du contexte IA pour les parties algorithmiques, car les données sont dynamiques, adaptées à la partie IA, et les structures sont particulières au langage IA. C'est ce que l'on constate dans les travaux abordant l'interfaçage [Rueher & al 85 , Hubner & Markov 86], pour des contextes spécifiques.

La fabrication peut être schématisée de la manière suivante :

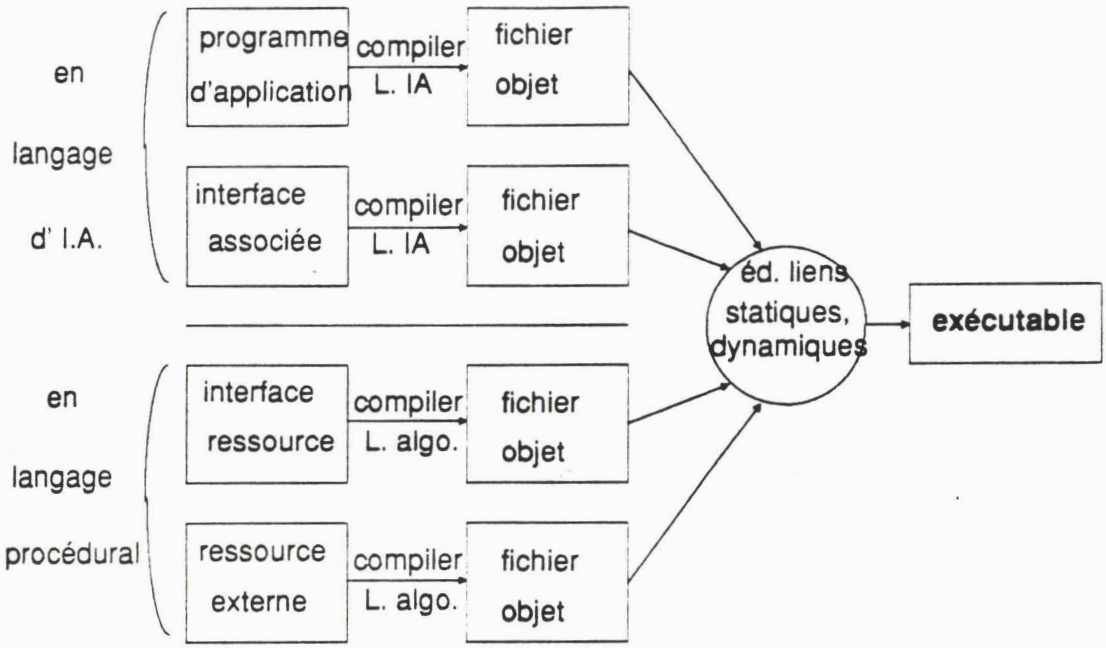


Fig II-25

Les langages qui satisfont à ce schéma sont :

- D-PROLOG (MS-DOS, MS-PASCAL) linker Microsoft
- Turbo-PROLOG (MS-DOS, assembleur 8086, Linker Microsoft)
- PROLOG II (MULTICS, PASCAL_SOL, liens dynamiques)
- LE-LISP 15.2 (UNIX, MULTICS), (C, FORTRAN), liens statiques ou dynamiques)
- VAXLISP (VAX VMS, VAX (C, FORTRAN, PASCAL), liens statiques VAX).

Les implanteurs et éditeurs de langage privilégient les interfaces directes vraies, mais il faut être conscient du fait qu'une étude pratique approfondie doit être faite, au niveau de la compatibilité réelle entre langages, pour éviter des problèmes d'interfaçage logiciel système, problèmes qui sont toujours gênant, mais non fondamentaux

Finalement, plus le système d'exploitation de la machine cible est évolué et rigoureux, et plus on aura de chance de réaliser l'opération sans trop d'efforts.

II.7. Généralisation de l'interfaçage

Cette partie est une réflexion informelle sur les interfaçages possibles, voir nécessaires, entre les systèmes d' intelligence artificielle et les systèmes classiques de CAO, ou plus généralement les logiciels procéduraux.

Nous avons identifié trois possibilités de connexion entre ces deux types de système :

- Une partie experte utilise les fonctionnalités d'une partie procédurale pour résoudre une portion réduite et/ou très spécialisée du problème (calcul scientifique, algorithme spécialisé, logiciel de simulation, ...).
- Une partie d'un logiciel construit sur un modèle procédural délégue une part du travail à une partie intelligente.
- Les deux parties coopèrent entre elles pour trouver ou proposer une solution à un problème.

Pour chaque possibilité de connexion envisagée, nous essayons d'en évaluer l'intérêt, de décrire le fonctionnement de la connexion, de voir quand on doit envisager l'utilisation d'une telle connexion.

On cherche ici à donner une vue d'ensemble du problème de l'interfaçage entre des logiciels basés sur des techniques procédurales classiques et des logiciels utilisant les techniques IA courantes. Les interfaçages proposés par les constructeurs de logiciels sont souvent du premier type (Cf. les paragraphes qui précèdent), à savoir que l'on peut créer un système intelligent qui utilise des parties algorithmiques spécialisées. Ce type d'interface est le plus facile à mettre en oeuvre, mais il nous semble insuffisant, car il suppose une utilisation particulière des logiciels procéduraux existant.

II.7.1. Appel d'une ressource procédurale

II.7.1.1. Intérêt de cette connexion

Cette connexion est la plus répandue dans l'industrie du logiciel, et c'est celle que nous avons étudié dans ce chapitre.

Nous ne reviendrons pas en détail sur l'intérêt de ce type d'interface, car nous l'avons déjà amplement décrit au paragraphe II.2.2. Pour résumer, les apports sont :

- La possibilité d'utiliser des parties algorithmiques du SCAO, parce que le langage d'intelligence artificielle utilisé n'est pas adapté à l'écriture d'une telle partie, ou parce que l'on ne veut pas refaire ce qui existe déjà et fonctionne très bien dans une partie procédurale. Cela peut aussi servir à gérer "intelligemment" des algorithmes spécialisés dans un domaine [Jonckers 85].

- l'enrichissement, et une évolution nécessaire des SCAO, de manière à ce qu'ils puissent réellement apporter une aide au concepteur : pour la partie de conception proprement dite, par exemple quand il faudrait être capable de donner un avis ou une aide personnalisée et adaptée à l'utilisateur; pour utiliser au mieux un système souvent complexe; pour avoir une meilleure appréhension et compréhension du projet global en cours.

II.7.1.2. Les parties interfacées

Dans ce type de connexion, l'élément de base connecté à la partie dite intelligente est le sous-programme externe. Celui-ci est déclenché au moment opportun d'une prise de décision. Le sous-programme réalise une action que l'on peut décrire de façon impérative et renvoie si nécessaire des informations d'un format prédéfini à la partie I.A.

Le déroulement de l'appel se passe de la manière suivante :

- Prise de décision (inférence, induction), qui décide de l'activation de la partie procédurale externe.
- Préparation des informations et données à transmettre (contenu et structure), avec un contrôle si nécessaire de la validité de ces données pour la partie procédurale.
- Utilisation d'un mécanisme d'interfaçage logiciel pour transférer le déroulement du programme à la partie externe.
- lancement de la partie externe.
- Analyse, contrôle, transformations, et transfert des données reçues, pour leur utilisation par la partie IA.
- Retour à la partie IA, grâce au mécanisme d'interfaçage logiciel entre les deux parties.
- Reprise normale du déroulement de la partie IA.

On suppose ici que les deux ensembles logiciels travaillent de façon séquentielle, car un fonctionnement multitâche nécessiterait des mécanismes de synchronisation dont nous ne nous préoccupons pas ici.

Cette séquence s'adapte bien aux langages LISP et PROLOG, ou aux méthodes primitives de SMALLTALK. Néanmoins il existe un problème conceptuel dans le cas de SMALLTALK : Il s'agit d'interfacer ce langage à objets avec des langages classiques, mais rien n'est prévu dans SMALLTALK pour réaliser cela.

Les langages hybrides tels que C++ ou Eiffel sont une bonne alternative à SMALLTALK, car ils permettent une implémentation facile d'objets externes, puisqu'ils sont conçus comme des "sur-langages" d'un langage classique (préprocesseur C++ par exemple) et aussi parce qu'ils disposent de mécanismes de connexion vers des langages externes.

II.7.1.3. Problèmes du contrôle des données dynamiques

Les programmes écrit en langage IA se caractérisent aussi par l'aspect dynamique des données qu'ils manipulent. Certes les langages procéduraux permettent de créer et manipuler dynamiquement des données; mais la création, la manipulation et la destruction de celles-ci doivent être explicitement précisées dans le code du programme. A l'inverse, la gestion des données se fait automatiquement dans les langages d I.A. (PROLOG, LISP); d'autre part les données créées lors de l'exécution de ces programmes sont de nature très versatile.

En PROLOG par exemple, les mécanismes d'unifications, la possibilité du retour arrière (backtracking) dans la résolution, et l'instanciation d'une règle, ont pour effet de fabriquer des données de structures différentes pour une même variable de prédicat, et ceci à différentes étapes de déroulement du programme.

Ceci veut dire que si un prédicat PROLOG (vu comme un prédicat externe) doit être écrit comme un sous-programme en langage procédural, il faut prévoir d'avance les différentes formes que peut prendre une variable lors de l'unification sur ce prédicat. Néanmoins, il faut aussi étudier le cas où la variable est produite par le langage externe et utilisée par PROLOG au retour de l'appel du sous-programme.

De ce problème sont issues les différentes techniques d'interfaçage que l'on retrouve dans les langages IA : "flow patterns" de Turbo-PROLOG, contrôles en série dans le cas du langage D-PROLOG, "Alien structures" en VAXLISP. Ces mécanismes ont pour but de contrôler les structures de données dynamiques, dans les appels de prédicats ou fonctions externes.

De tels mécanismes conviennent bien lorsque les connexions entre les deux types de langage ne sont pas complexes. Elles sont insuffisantes en cas de réalisations "industrielles", et il faut alors prévoir un type de communication de plus haut niveau.

Dans le cas de LISP, la structure de donnée de base non élémentaire est la liste. Ce type de donnée est standard et de structure bien connue et finalement adaptée à la fois à des traitements symboliques ou procéduraux [Winston & Horn 84, Abelson & al 88]. Elle est donc facilement contrôlable et se manipule assez simplement. Ceci permet de prévoir des mécanismes de contrôle de plus haut niveau et plus systématiques. Comme LISP est un langage qui comporte des aspects procéduraux non négligeables, le problème des données dynamiques et non typées est certainement beaucoup plus facile à résoudre ici que pour d'autres langage d'I.A..

En général, tous les programmes IA utilisent l'instanciation et sont de gros consommateurs de mémoire, du fait de la création automatique de données. C'est pourquoi, en cas d'interfaçage des deux types de langage, il faut réaliser un contrôle efficace et sûr des données échangées, et assurer une souplesse et une robustesse importantes des parties écrites en langage procédural.

II.7.1.4. Quand utiliser cette connexion

Indépendamment de ces problèmes logiciels de base, le programmeur IA doit se poser la question de savoir pourquoi intégrer une partie procédurale et comment utiliser celle-ci. Elle peut :

- proposer un outil standard de résolution de problème (c'est le cas d'un algorithme de calcul spécialisé),
- réaliser d'une certaine manière un scénario d'utilisation du système procédural (comme par exemple déclencher une "macro-action" du système de CAO),
- résoudre un sous-cas du problème général traité dans la partie intelligente (cas des algorithmes de placement-routage spécialisés).

Le programmeur devra donc garder à l'esprit le but de la connexion logicielle dans son programme, afin d'utiliser au mieux la partie procédurale. Suivant l'utilisation et le type de partie algorithmique utilisée, le contrôle nécessaire dans les zones d'interfaces sera de taille et de capacité variables.

II.7.1.5. Conclusion

L'utilisation d'une partie écrite dans un langage procédural classique, au sein d'un logiciel IA, permet :

- de pallier aux manques du langage IA utilisé,
- d'utiliser (ou de réutiliser) les réalisations existantes du domaine,
- de simuler des utilisations d'un système procédural existant et dont se sert la partie intelligente.

Cependant, les problèmes de logiciel de base à résoudre dans ce type d'interfaçage ne sont pas très simples, du fait de l'aspect dynamique et non typé des données utilisées et créées par les systèmes d'intelligence artificielle.

En particulier, un mécanisme important de contrôle et de vérification des données échangées entre les deux parties du système logiciel doit être intégré à la connexion, à cause de la nature radicalement opposée de la programmation symbolique et/ou déclarative vis à vis de la programmation procédurale.

II.7.2. Appel d'une ressource déclarative

II.7.2.1. Intérêt de cette connexion

En programmation procédurale classique, les problèmes que l'on peut résoudre sont déterministes. On utilisera un ou plusieurs algorithmes spécialisés pour résoudre un problème particulier. Suivant le domaine de spécialité du programme ou du logiciel final, les solutions apportées sont plus ou moins satisfaisantes, car elles sont issues d'algorithmes spécialisés qui ne proposent pas toujours de solution satisfaisante. Parfois, et c'est le cas quand il faut satisfaire un grand nombre de contraintes, on ne sait pas fabriquer d'algorithme traitant le problème.

Faire appel à une partie déclarative peut être intéressant quand :

- Il n'existe pas d'algorithme idéal qui résolve complètement le problème. Par contre, il existe des techniques empiriques, basées sur l'expérience d'experts du domaine. On peut donc utiliser efficacement des techniques d'I.A, donc faire appel à des parties spécialisées en IA, quand on sait qu'une partie du problème est insuffisamment traitée par des algorithmes. Par exemple certains algorithmes de routage ont des difficultés à résoudre les routages d'alimentation, il est dans ce cas intéressant de déléguer cette partie du travail à une partie intelligente qui reproduit le raisonnement d'experts du domaine sur des cas caractéristiques non résolus par l'algorithme.
- On sait fabriquer des algorithmes qui résolvent le problème, mais se révèlent être de très gros consommateurs de temps machine, ou d'espace mémoire. Autrement dit, les algorithmes sont inefficaces du point de vue informatique. Ceci peut arriver dans des cas où l'on a une explosion combinatoire de l'ensemble des solutions, et que la recherche de ces solutions nécessite des calculs importants. La partie procédurale délégue le travail qu'elle ne sait pas effectuer (choix entre plusieurs solutions intermédiaires, prise de décision relative à l'expertise dans le domaine) à une partie utilisant une technique classique IA (algorithmes de recherches, fonctions d'évaluations, etc...).

II.7.2.2. Fonctionnement de la connexion

Le type d'architecture logicielle de l'ensemble procédural + déclaratif, si une partie I.A. est appelée par une partie procédurale, peut être décrit de la manière suivante :

- les deux parties doivent être faiblement connectées.
- une partie spécifique d'interface transfère les données d'une partie à l'autre et en contrôle la validité.
- la partie I.A utilise ses propres données, plus celles de la partie procédurale. Ses données internes sont celles qui représentent sa connaissance, celles de la partie procédurale sont des faits concernant le problème. Il n'y a pas de dialogue entre les deux parties.

Le déroulement de l'appel se fait de la manière suivante, en supposant les problèmes d'interfaçage logiciel de base résolus :

- sélection de faits représentant suffisamment le problème par la partie procédurale.
- appel de la partie I.A. par l'intermédiaire de l'interface logicielle, avec "traduction" des données pour la partie I.A.
- tentative de résolution par la partie I.A.
- transfert et traduction inverse des résultats par l'interface logicielle.

Dans le cas où le problème est très spécialisé, et quand on sait qu'un module "expert" sait résoudre correctement la majorité des problèmes, on pourra utiliser cette architecture pour enrichir les parties procédurales d'une capacité de choix, de résolution, ou de décision.

II.7.2.3. Problèmes système

Dans ce type de configuration logicielle, deux points importants doivent être respectés :

- Il faut trouver un bon procédé de connexion entre les deux parties :
  La connexion se fera au sein d'un unique programme, auquel cas il faudra insister sur la compatibilité entre les langages utilisés au sein des deux parties.
  La connexion peut être vue comme l'activation d'un module expert externe, qui fonctionne indépendamment du processus procédural. Il faut alors insister sur la manière de transférer les données entre les deux parties, savoir gérer la production et la consommation de données par chacune d'elles (fonctionnement asynchrone).
- Il faut surtout éviter d'alourdir ou de diminuer l'efficacité de l'ensemble logiciel. Les parties intelligentes ne doivent pas être trop coûteuses en temps et en mémoire; le choix des techniques I.A. utilisées est déterminant.

les problèmes rencontrés au niveau système seront nombreux, et dépendront des choix d'implémentation pour chacune des deux parties : des parties faiblement connectées faciliteront la mise en oeuvre, mais sans doute au dépends de l'efficacité, tandis que le choix inverse permettra d'obtenir une bonne efficacité, au prix de gros efforts d'implantation de l'interface. Ils seraient alors beaucoup plus performants s'ils utilisaient des connaissances "expertes" quand l'algorithmique est impuissante où qu'ils doivent faire des choix pertinents sur le problème traité.

II.7.2.4. Conclusion

Les problèmes d'intégration d'un logiciel d'intelligence artificielle avec un logiciel procédural sont nombreux. Outre que les qualités d'efficacité du logiciel impératif sont perdues, à cause de la gourmandise en temps et en mémoire de la partie I.A., l'aspect dynamique des données de cette partie est difficile à traiter par la partie procédurale.

Il est donc essentiel d'évaluer le plus finement possible les rapport entre les avantages et les inconvénients de l'intégration d'une partie I.A. avant de faire la réalisation d'une telle connexion. Dans ce cas, *le contexte système du logiciel intégré ainsi que le contexte d'utilisation* d'un tel logiciel seront déterminant dans le choix de la réalisation de la connexion.

On trouvera néanmoins un avantage certain à l'apport d'une telle connexion dans un logiciel procédural, quand l'algorithmique est impuissante pour résoudre un problème particulier de façon déterministe ou quand un choix pertinent entre plusieurs solutions s'impose.

II.7.3. Coopération des systèmes déclaratifs et procéduraux

II.7.3.1. Intérêt de cette connexion

Beaucoup de problèmes sont ou peuvent être aujourd'hui résolus par des systèmes algorithmiques classiques. Tout le savoir, la connaissance et les techniques développés et accumulées depuis les débuts de l'informatique pour traiter les problèmes les plus divers : gestion, calcul scientifique, visualisation graphique, simulation, conception assistée par ordinateur; sont autant de domaines où la programmation procédurale a fait ses preuves. Les techniques classiques ne permettent pas toujours de traiter tous les domaines.

D'autre part , l'intelligence artificielle a apporté, par le biais des systèmes à base de connaissances ou plus généralement dits "intelligents", la possibilité de résoudre des problèmes non déterministes ou que l'on ne sait pas formuler ou traiter aisément de manière traditionnelle.

Les logiciels informatiques sont aujourd'hui de plus en plus gros, complexes, et utilisent un nombre croissant de techniques informatiques spécialisées. Un soucis constant et louable de la part des créateurs de logiciels est d'éviter de "réinventer la roue" sans cesse [Cox 86]. L'idée de faire coopérer ensemble des logiciels de natures différentes est actuellement à la mode (projets d'intégration européens ESPRIT [Eggen & al 90], usine à logiciels ESF [ESF 89, Hubert & Perdreau 90] ...).

C'est pourquoi les systèmes logiciels où plusieurs parties indépendantes coopèrent sont appelés à tenir un rôle essentiel pour l'industrie du logiciel dans le futur. On a souvent parlé de composants logiciels pour réaliser l'industrialisation de la production de logiciel, il faut aussi développer des techniques de base pour la coopération de logiciels de nature procédurale et déclarative. Les systèmes multi-agents intégrant plusieurs systèmes à base de connaissance de nature différente ont déjà fait l'objet d'études de plus en plus approfondies [Eggen & al 90, Marty & al 91].

Il faudrait maintenant formaliser, modéliser, et systématiser la coopération et l'intégration de logiciels intelligents et procéduraux. Cette formalisation est nécessaire pour bénéficier au mieux des capacités de chacune de ces deux techniques.

II.7.3.2. Conclusion

La coopération des système déclaratifs et procéduraux reste un problème ouvert à ce jour. Certes les projets de génie logiciel concernant la communication intégrée de logiciels différents tels que ESF ou ACKNOWLEDGE a bien fait progresser le domaine, mais chacun des deux ne s'intéresse qu'à une vision particulière du problème : communication intégrée et compréhension mutuelle des logiciels pour ESF, intégration de systèmes à base de connaissance différents pour ACKNOWLEDGE.

Les travaux de B.Trousse concernant la coopération entre systèmes à base de connaissance et outils de CAO par la création d'un environnement adapté [Trousse 89][Trousse 90][Neveu et al. 90] constitué de logiciels dédiés à la communication (MAILY), à la modélisation "intelligente" adaptée aux techniques IA (XCAD), sont une première réussite dans ce domaine. La technique utilisée consiste en la réalisation d'une part d'un système de communication adapté, et d'outils spécialisés en IA, capables de prendre en compte et de représenter les connaissances CAO d'une manière adaptée aux techniques IA.

La technologie de l'intégration doit donc encore faire beaucoup de progrès pour que l'on puisse sérieusement envisager la collaboration effective entre un système procédural et un système déclaratif.

II.8. Conclusion

L'intégration entre systèmes IA et systèmes de CAO est une étape nécessaire à l'évolution des systèmes de CAO actuels [Latombe 85]. Elle nécessite de résoudre plusieurs types de problèmes parmi lesquels la communication de base entre systèmes IA et systèmes CAO.

Nous avons ici étudié les possibilités de communications entre langages IA et langages procéduraux utilisés en CAO, essayé de formaliser les procédés de communication et de transmission des données qui pourraient être échangées dans un système intégrant IA et la CAO, et montré deux exemples d'application de ce formalisme. Puis nous avons évalué les possibilités de communication de données pour les langages IA les plus utilisés, avec un aperçu pour les langages de type hybrides. Enfin, nous avons essayé de voir dans quelle mesure on peut faire coopérer des SCAO avec des systèmes IA.

Si l'utilisation de sous-programmes externes écrits en langage procédural est en général possible, la technique effective dépend de l'implantation du langage IA et de celle du langage procédural, et est souvent accessible par le moyen d'une interface logicielle directe, qui semble être le moyen le plus satisfaisant de communication à ce niveau très bas d'intégration. Il n'existe cependant pas de méthode générale efficace de communication entre langages et les mécanismes fournis par les systèmes d'exploitation sont généralement coûteux.

A un niveau d'abstraction plus élevé, l'intégration IA-CAO passe par un enrichissement des modèles descriptifs des SCAO pour y intégrer une partie des structures sémantiques nécessaires à la réalisation des programmes IA dotés de capacités de raisonnement.

Finalement, une solution envisageable au problème général de l'intégration pourrait être la réalisation d'environnement intégrés adaptés tels que celui réalisé par B.Trousse [Trousse 89], car l'évolution de la puissance des systèmes et des machines améliore finalement les solutions de communication peu efficaces fournis par les systèmes d'exploitation. De ce fait, on peut fabriquer des logiciels dédiés à la communication entre SCAO et système IA, ce qui facilite l'intégration et la communication en la plaçant à un niveau d'abstraction plus élevé que celui que nous avons considéré dans notre étude.

Néanmoins, les techniques et formalismes que nous avons décrits ici pourront être repris avantageusement et apporteront une bonne efficacité, dans des cas particuliers où soit l'efficacité des transmissions de données, soit le caractère généralisateur de la communication est l'élément limitatif du problème d'intégration.

Acquaintance, n. *A person whom we know well enough to borrow from, but not well enough to lend to. A degree of friendship called slight when its object is poor or obscure, and intimate when he is rich or famous.*

Ambrose Bierce. *The devil's Dictionary.*

Introduction au chapitre III

L' objet de cette troisième et dernière partie est la réalisation d'un système d'aide pour l'aménagement spatial (en deux dimensions). Ce système est constitué d'une interface utilisateur de type courant en CAO, ainsi que d'un module intelligent de placement basé sur un modèle distribué.
Les SCAO sont dotés d'aides diverses pour guider l'utilisateur dans sa conception, mais les aides proposées actuellement ne font que guider le concepteur dans l'utilisation du système, et elles ne l'aident jamais dans la conception proprement dite [Lebahar 85, Trousse 89]. D'autre part une grande partie des tâches routinières en CAO quand on utilise un logiciel de schématique consiste en la manipulation manuelle des composants représentant les sous-parties ou les composants physiques pour mettre à jour ou réarranger l'agencement spatial de ces composants. Ces deux constatations constituent le point de départ de notre réflexion et des réalisations rapportées ici.

Avec le type d'aide couramment fourni dans les SCAO, l'utilisateur peut a tout moment être informé du paramétrage et de l'action sémantique d'une commande particulière, par l'intermédiaire d'un système d'aide que nous qualifions dans cette thèse de statique (Cf III.1.3). Il peut aussi disposer de renseignements sur le travail qu'il a effectué. Les renseignements alors fournis concernent le plus souvent les commandes que l'utilisateur peut appeler à un moment donné, ou des informations sur la partie de logiciel qu'il utilise actuellement. Finalement ce sera quasiment toujours des informations concernant le fonctionnement du SCAO qui seront fournis à l'utilisateur, un peu à la manière d'un manuel utilisateur [Shneiderman 87].

Par contre, aucun des systèmes d'aide actuels ne permet à l'utilisateur d'examiner la qualité de sa conception [Trousse 89]. Ce type d'aide nécessite de juger et de raisonner sur le travail de l'utilisateur. Par exemple en comparant chaque partie de son travail par rapport aux réalisations courantes que l'on trouvera dans les bases de projets, ou bien en signalant au concepteur dès que possible un point défaillant de sa conception. Dans le premier cas, on doit reconnaitre le type de réalisation effectuée par le concepteur (reconnaissance de cas ou de situation de conception), et on devra l'analyser par rapport à l'existant. Dans le deuxième cas, il faut juger le travail réalisé par l'expert en à partir de connaissances techniques du domaine d'application et du savoir-faire préexistant, et il faut proposer si nécessaire des modifications sur l'objet conçu. D'où l'intérêt de certaines techniques d'I.A. permettant la reconnaissance de cas ou de formes, ou la représentation diversifiée et puissante des connaissances expertes d'un domaine.

Nous présentons dans ce chapitre un système de résolution distribuée de conflits géométriques dans un espace à deux dimensions. Il s'agit en fait d'un système d'aide interactif "intelligent" qui tente de résoudre des problèmes de recouvrement géométrique entre composants d'une carte de circuit imprimés. Cependant nous avons travaillé en considérant le problème de l'aménagement spatial d'objets en deux dimensions, problème qui est omniprésent en cas d'utilisation de logiciel de schématique, soit directement de part le domaine d'application visé (cas de la CAO mécanique ou d'architecture), ou à cause de considérations techniques importantes pour réaliser le produit (cas de la CAO en électronique où l'on doit disposer les composants sur une carte à partir de contraintes fonctionnelles ou techniques).

Le système que nous avons développé et que nous décrivons ici permet principalement à l'utilisateur d'éviter beaucoup de manipulations manuelles fastidieuses en cas de modification du produit provoquant des conflits de recouvrement géométrique entre composants. Dès qu'un tel conflit est détecté, le système tente de résoudre le problème, suivant sa connaissance du conflit local et du domaine d'application.

Ce dernier chapitre est structuré de la manière suivante :

Nous présentons tout d'abord sommairement l'aspect habituel d'une interface utilisateur de SCAO ainsi que les systèmes d'aide que l'on rencontre couramment sur de tels logiciels (III.1).

Nous décrivons ensuite le problème à résoudre. Nous précisons d'abord ce qui motive la réalisation d'une telle interface intelligente (III.2.1), les caractères généraux d'un système d'aide à la conception de ce genre (III.2.2), et donnons une description sommaire de l'interface de base que nous proposons (III.2.3).

Nous décrivons ensuite MAPS, le modèle distribué que nous avons réalisé pour résoudre le problème important des conflits géométriques qui apparaissent lors de l'utilisation d'un outil de schématique impliquant l'aménagement spatial. Après une brève introduction sur les points qui caractèrisent MAPS (III.3.1), nous précisons les raisons de notre choix de modèle distribué (multi-agents décentralisé) pour le problème à résoudre. Nous introduisons ensuite le concept de "voisinage" de composant, notion clé pour "socialiser" les agents du système (III.3.3), après quoi nous décrivons le voisinage que nous avons choisi dans MAPS (III.3.4, III.3.5, III.3.6). Enfin nous présentons formellement la gestion distribuée de voisinage que nous avons réalisée dans MAPS (III.3.7). Nous concluons cette section en rappelant les points importants de MAPS (III.3.8).

Dans la section suivante (III.4), nous décrivons l'implantation du système en machine avec le langage Smalltalk. Après avoir précisé l'environnement logiciel utilisé (III.4.1), nous décrivons l'architecture logicielle de cette implantation (III.4.2). Viens ensuite la description succincte des classes qui composent le système (III.4.3 à III.4.6).

Nous avons également conçu, pour évaluer la faisabilité d'un véritable système d'aide intelligent, des algorithmes de résolutions de conflit spatial utilisant deux stratégies de résolution distribuée différentes (III.5). Après présentation du problème de résolution et des deux stratégies en question (III.5.1 et III.5.2), nous présentons les deux algorithmes de résolution satisfaisant à ces stratégies (III.5.3 et III.5.4) et évaluons l'intérêt de la résolution distribuée présentée pour le système d'aide à la conception envisagé. Nous terminons cette section en présentant nos idées de base pour réaliser un SCAO disposant d'une aide intelligente dans le domaine de l'agencement spatial, aide qui serait basée sur un modèle décentralisé du type de MAPS (III.6).

III MAPS : UN MODELE DECENTRALISE POUR SYSTEME DE PLACEMENT INTERACTIF

Les systèmes de CAO disposent couramment de mécanismes d'aide à l'utilisation. C'est un critère ergonomique indispensable pour un tel système [Gardan 86b, Quintrand &others 85, Shneiderman 87]. Cependant ces mécanismes offerts à l'utilisateur sont destinés uniquement à l'aider dans le maniement du logiciel pour les cas les plus courants.

Ce sont en fait des systèmes passifs [Shneiderman 87] qui n'évaluent ni ne reconnaissent la sémantique de ce que conçoit l'utilisateur. Au contraire de ce genre d'aide, on peut envisager des aides beaucoup plus sophistiquées qui guident l'utilisateur tout au long de sa conception, par exemple en préconisant à un instant donné l'utilisation d'un composant particulier plutôt qu'un autre, ou en signalant que la solution envisagée pour résoudre le problème de conception diffère de celles utilisées dans la majorité des réalisations trouvées dans la base de projet, et en précisant certains problèmes particuliers de fonctionnement du produit qui pourraient en résulter. Un tel système aiderait réellement le concepteur à évaluer immédiatement la validité de sa réalisation. De telles aides sont difficiles à réaliser, car elle nécessitent de reconnaitre l'intention de conception de l'utilisateur et de disposer d'une connaissance importante dans le domaine de conception.

Les systèmes d'aide actifs réagissant dynamiquement aux actions de l'utilisateur en les analysant et en le conseillant devront intégrer des techniques d'intelligence artificielle évoluées, ne serait-ce que pour intégrer le savoir-faire des experts ou la technologie du domaine d'application.

Sans vouloir réaliser une aide d'une telle capacité, nous avons remarqué que dans les SCAO électroniques, une grande partie du travail de l'expert consiste à placer et déplacer des composants pour aboutir à une solution satisfaisante. En effet, si le concepteur doit modifier une zone dans laquelle se trouve beaucoup de composants, il a toute chance de devoir modifier l'environnement local de cette zone, et donc de devoir manipuler manuellement tout un ensemble de composants et de connexions.

Partant de cette idée, nous avons voulu réaliser un système d'aide dont on désirerait qu'il soit capable de résoudre des conflits géométriques locaux entre composants. Il s'agit en fait, lorsque l'utilisateur a déplacé un composant, en provoquant de façon quasi inévitable un conflit de recouvrement géométrique avec un autre composant, de résoudre ce conflit par un système de résolution particulier.

III.1. Interface usuelle d'un SCAO

III.1.1 La partie présentation de l'interface

La partie du logiciel de CAO à laquelle nous nous intéressons ici concerne essentiellement celle où l'utilisateur choisit interactivement des éléments d'une base de composants, éléments qui dépendent du domaine technique, et les organise pour construire un élément composite ou un objet complet [Trousse 87]. Ce schéma de construction s'applique autant à l'électronique [Dejesus & Callan 85] qu'à la mécanique [Trousse 87], ou à l'architecture. Il s'agit en fait des parties de logiciel de CAO faisant intervenir directement des aspects de schématique dans la conception.

Pour réaliser ce genre de travail, on fournit à l'utilisateur du système un type d'interface assez caractéristique que nous appellerons "interface interactive graphique d'édition". On peut simplifier à l'extrême l'aspect visuel d'une telle interface par le schéma III.1 :

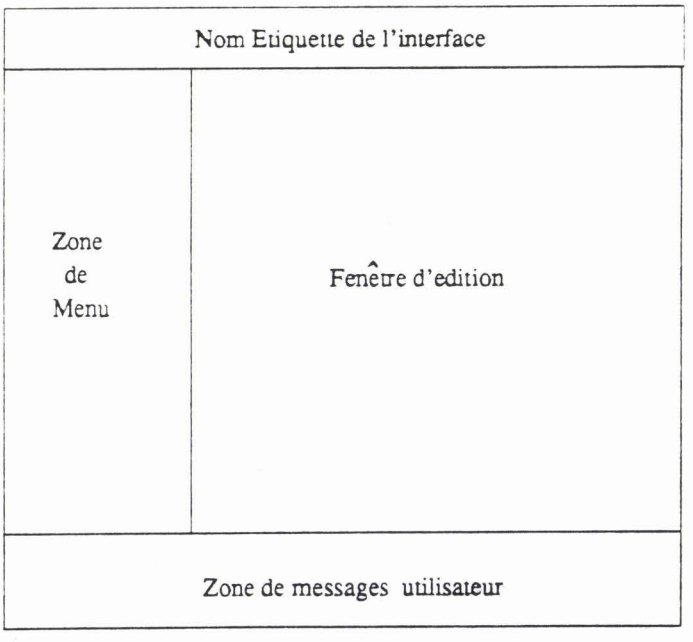


Fig III.1 Interface graphique interactive d'édition

L'interface graphique interactive d'édition divise un écran (terminal graphique) en plusieurs zones interactives où l'utilisateur peut voir à l'aide de schémas et de représentations diversifiées le produit conçu, et modifier sa conception.

Trois parties caractéristiques se retrouvent dans toutes les interfaces graphiques interactives d'édition. Ce ne sont pas les seules, mais l'on retrouve toujours ces zones d'interaction dans tous les logiciels de CAO.

1. La **zone de menu** liste une série de commandes autorisées à un moment donné. L'utilisateur choisit l'une de ces commandes en la désignant par pointage avec un curseur graphique manipulé grâce à un appareil d'entrée physique comme une souris ou une table à digitaliser.
   Un menu peut lui même comporter des sous-menus pour simplifier la présentation des commandes disponibles. Nous ne considérerons ici que les ordres de base de manipulation du schéma : créer, placer, déplacer, copier, détruire ...

2. La **fenêtre d'édition** est la zone dans laquelle l'utilisateur du logiciel peut avoir une représentation visuelle graphique de l'état actuel de son produit. Par exemple, dans le cas de CAO électronique, il verra lors de la phase de construction du schéma électronique une représentation normalisée correpondant au schéma logique du circuit, et lorsqu'il passera à la phase de placement-routage des composants, cette fenêtre contiendra une représentation physique de la carte.
   Une telle fenêtre dispose en même temps de capacités de représentation visuelle graphique (sortie graphique) et d'interaction à manipulation directe [Shneiderman 87], à savoir qu'il peut interagir directement avec les composants et éléments visualisés.

3. La **zone de messages utilisateur** est une zone d'information renseignant l'utilisateur en permanence sur la conception réalisée. C'est habituellement une fenêtre où les informations sont présentées sous forme textuelle. On peut obtenir des informations par résultat direct d'une commande ou comme écho d'une interaction particulière sur le schéma.
   Souvent cette zone permet aussi une forme d'interaction spécialisée à l'aide de langages de commandes.

La zone contenant le nom de l'application permet essentiellement d'identifier la partie du logiciel utilisée lorsque celui-ci fonctionne dans un environnement multi-fenêtres.

III.1.2 Interaction avec l'utilisateur

On peut déclencher une action simple ou réaliser une action complexe de trois manières différentes :

1. Par l'intermédiaire de la zone de menu, on choisira une action à réaliser sur un élément, un groupe d'éléments, ou sur le projet complet. Il s'agira ici principalement d'actions élémentaires dont le paramétrage est simple. Par exemple, "créer" un nouveau composant, ou le déplacer.
   Une case de menu peut parfois permettre d'activer le paramétrage d'actions complexes. En effet, le paramétrage d'une commande peut parfois être assez lourd, et une interaction par activation d'un choix dans un menu n'est alors plus suffisant pour réaliser ce paramétrage. Le choix correspondant à une action complexe dans un menu est en fait l'activation d'une commande par l'intermédiaire du langage de commande existant, avec présentation à l'utilisateur d'un formulaire de paramétrage à compléter, ou un squelette de phrase du langage de commande présenté dans la zone de message, nécessitant lui aussi la complétion des paramètres par l'utilisateur.

2. Par l'intermédiaire de la zone de messages, l'utilisateur peut décrire et déclencher n'importe quelle action autorisée. L'action est écrite dans la syntaxe du langage de commande, et déclenchée par validation (par exemple au clavier).

3. Par l'intermédiaire de la zone d'édition, l'utilisateur peut interagir directement avec les composants visualisés. Par simple désignation du composant, il peut agir sur celui-ci, par exemple pour un déplacement de composant, ou une modification simple d'attribut. D'autres protocoles d'interaction spécifiques au logiciel de CAO utilisé permettent d'interagir avec un groupe de composants.

III.1.3 Systèmes d'aide et de validation usuels

Disposer d'une aide "en ligne" est une nécessité fondamentale pour un logiciel interactif, et plus encore pour un SCAO qui est en général très riche du point de vue de ses posssibilités d'utilisation. Il est absolument indispensable quand il n'existe que des manuels d'utilisation rudimentaires. Il est utile pour l'utilisateur expérimenté, pour se remémorer comment utiliser une commande inhabituelle du système. L'utilisateur débutant trouvera quant à lui une façon rapide et pratique de prendre contact avec le logiciel.

Les systèmes d'aide couramment rencontrés renseignent l'utilisateur sur le fonctionnement et l'utilisation du logiciel. Les SCAO ne se distinguent pas particulièrement des autres logiciels informatiques en ce domaine.

Ben Schneiderman [Shneiderman 87]donne une liste d'aides intéressantes pour un logiciel intégré :

- Explications de plus en plus détaillées d'un message d'erreur.
- Explications de plus en plus détaillées d'une question posée à l'utilisateur ou d'un message particulier.
- Exemples d'entrées et de commandes valides.
- Explication ou définition d'un terme spécifique.
- Description du format d'une commande spécifique.
- Liste de commandes autorisées.
- Visualisation de parties spécifiques de la documentation.
- Visualisation des valeurs d'attributs courants du système.
- Mode d'emploi du système.
- Nouvelles pouvant intéresser les utilisateurs du système.
- Liste des aides disponibles.

Toutes ces aides sont statiques, en ce sens qu'elles permettent uniquement de renseigner l'utilisateur sur le fonctionnement du système.

Pour valider sa conception, l'expert dispose de logiciels spécialisés qu'il peut activer au moment où il le désire. Ces logiciels permettent de tester l'ensemble de l'état courant du produit conçu pour une fonctionnalité technique précise. Il pourra par exemple s'agir de tester les qualités de refroidissement d'un circuit.

Ces outils s'utilisent principalement d'une manière indépendante de la conception et ne font aucune correction de celle-ci; ils ne font que signaler des dysfonctionnements ou des erreurs quand cela leur est possible. Les outils de simulation permettent d'observer plus finement le comportement du produit conçu.

Les systèmes d'aides et outils de validation actuels sont certes utiles, mais leur aspect statique nous parait insuffisant.

III.2. Présentation du problème

III.2.1 Motivations d'une aide évoluée

III.2.1.1 Quelques limitations des systèmes d'aide

Nous avons vu que les systèmes d'aides que l'on rencontre usuellement dans les SCAO permettent seulement de renseigner l'utilisateur, néophyte ou expert du système, sur le fonctionnement et l'utilisation de celui-ci.

Or, avec un outil de CAO, une grande partie du travail de conception peut être décomposé en séquences de créations et de modifications. Une part importante du travail de l'utilisateur consiste donc à modifier et évaluer personnellement l'état courant du produit conçu.

Quand l'utilisateur modifie l'état actuel du projet, cette modification induit des effets de bord sur une partie ou la totalité du projet, non seulement sur l'aspect géométrique du produit, mais aussi sur ses fonctions. Ainsi, certaines contraintes qui étaient vérifiées avant la modification peuvent ne plus l'être après cette modification, et l'utilisateur devra donc repasser manuellement par un cycle d'évaluation et de test du projet avant de valider son travail.

Quand l'utilisateur évalue son produit, il utilise sa connaissance experte du domaine pour reconnaitre localement et globalement des insuffisances ou des défauts de conception. On peut alors remarquer qu'une part importante de raisonnement nécessaire à l'évaluation se fait localement sur un groupe de composants et pour des contraintes fonctionnelles bien précises. Les systèmes d'aides d'aujourd'hui sont bien incapables de libérer le concepteur des petites tâches routinières d'évaluation et d'expertise de ce genre.

Si des exigences d'aide à la conception se révèlent superflues pour certains SCAO, de tels outils faciliteraient grandement la tâche des concepteurs et leurs permettraient d'être plus

efficaces dans leur travail car libérés des opérations expertes routinières. Nous avons voulu contribuer à l'évolution des systèmes d'aides en réalisant une évaluation et une correction ou reconception immédiate d'un changement d'état du projet. Le problème abordé est celui de la résolution de conflits géométriques entre composants, et il fait typiquement partie des tâches routinières d'aménagement de l'espace que doit opérer un concepteur en CAO électronique, ou en mécanique.

Nous concluerons en séparant l'aide à l'utilisateur dans les SCAO en deux catégories : l'aide à l'utilisation et l'aide à la conception. Ce qui distingue effectivement l'aide en ligne dans un système de CAO par rapport à n'importe quel logiciel, c'est la part importante d'expertise du domaine nécessaire à une bonne utilisation du logiciel. Il est alors indispensable de proposer des mécanismes élémentaires d'aide à la conception.

III.2.1.2 Définition et intérêt de l'aide à la conception envisagée

Ce que nous appellerons ici un système d'aide à la conception fournit à l'utilisateur des outils actifs pendant le travail de conception par l'intermédiaire du système, outils qui concernent la conception elle-même.

Pour une telle aide, tous les aspects d'expertise du domaine de conception peuvent être envisagés. Un tel système d'aide doit intégrer une part de connaissance experte du domaine.

Les intérêts d'un tel système sont de deux ordres :

1. Pour un concepteur expérimenté, il libérera celui-ci des fréquentes tâches d'évaluation lors de modifications du projet. Il mettra aussi en évidence des particularités du projet en cours de conception qui peuvent être néfastes pour le produit fini.

2. Pour un concepteur novice, il pourra guider efficacement celui-ci en lui précisant des erreurs de conception graves. Il lui sera également bien utile pour évaluer son travail et le corriger efficacement.

Cependant l'aide apportée ne doit pas se faire au détriment de l'efficacité et de la qualité ergonomique du système interactif. Il faut donc restreindre la connaissance et l'apport de l'aide pour que celle-ci ne pénalise pas outrageusement l'interaction avec l'utilisateur. Signalons que certains systèmes d'aides à la conception ont déjà été réalisés, par exemple dans IRENE [Coutaz 90].

III.2.1.3 Le problème de l'agencement spatial de composants

Dans beaucoup de problèmes de CAO les concepteurs sont confrontés lors de l'édition de schéma du modèle descriptif, c'est à dire dans les logiciels de schématique [Gardan 86b, Quintrand 85], au problème de la disposition dans l'espace des différents composants formant le produit final. De façon générale, il s'agit de les disposer correctement dans un

espace à deux ou trois dimensions de manière à former un assemblage cohèrent décrivant d'une manière synthétique l'objet conçu.

En architecture, cet agencement spatial fait partie du problème de conception de base. En mécanique, ceci fait partie de la description des relations fonctionnelles existant entre les composants. En électronique, c'est la technologie des cartes de circuit imprimés ou intégrés qui nécessite la disposition des éléments dans un espace à deux (voire trois) dimensions fini.

En électronique, les implémenteurs de SCAO ont travaillé depuis longtemps sur des algorithmes automatiques de placement des composants sur la carte. Il existe aujourd'hui beaucoup de tels algorithmes, et nous ne les décrirons ni ne les citerons ici. Cependant, il existe des techniques de base dont deux sont importantes : le placement constructif et le placement itératif [Carre 85a, Orchampt 87].

L'idée du placement constructif est de développer incrémentalement un noyau de composants placés. On choisit à un instant donné un composant non placé suivant des règles de sélection précises, et on le positionne ensuite sur la carte également suivant des règles précises. En placement itératif, on travaille sur un placement initial aléatoire ou issu d'un placement constructif, et on effectue des échanges d'emplacement de composants afin d'optimiser le placement suivant certains critères choisis (comme par exemple la longueur de connexions entre tous les composants).

Ce genre de technique est quasi absente en architecture et en mécanique car le placement des composants est intimement liés à leur fonction. Cependant le concepteur doit réaliser ce travail manuellement, souvent grâce à un éditeur graphique spécialisé.

Les techniques du placement constructif et du placement itératif simulent quelque peu le comportement de base d'un concepteur, mais sont cependant incapables de fournir des résultat complets ou intéressants. Les concepteurs sont toujours obligés de reprendre une part parfois importante de ce travail de placement (les résultats sont pires encore pour le routage des connexions entre composants). Les solutions algorithmiques se montrent donc insuffisantes pour réaliser globalement le placement des composants. C'est pourquoi de nombreux travaux impliquant des techniques d'Intelligence Artificielle ont été étudiées ces dernières années [Clerc & al 87, Dejesus & Callan 85, Flemming & Coyne 90, Jabri & Skellern 89, Joseph 85, Keen & Seviora 87, Odawara & al 87, Orchampt 87, Sriram 86, Tsuchida & al 89, Voirol & Piguet 89, Yoshimura & al 90].

Plutôt que de proposer un système qui s'attache à résoudre le problème dans sa globalité, nous avons préféré prendre le point de vue de l'utilisateur qui se sert d'un logiciel de schématique, et qui est confronté régulièrement à des sous-problèmes d'aménagement spatial (en particulier lorsque l'objet conçu est très complexe) locaux.

Notre but n'est pas dans ce travail de fournir un système automatique de placement global, mais semi-automatique d'aménagement spatial local. L'utilisateur se sert d'une manière habituelle de son logiciel de schématique, active l'aide du système quand il le désire, et

celle-ci se manisfeste seulement lorsqu'il y a conflit géométrique ou incohèrence ou encore problème de conception.

III.2.1.4 Apports et intérêts des techniques IA pour l'aide à la conception

Réaliser un système d'aide à la conception "dynamique" comme nous l'avons qualifié et défini précédemment nécessite de prendre en compte des aspects ayant rapport avec la sémantique et la fonction des objets conçus. En effet, comment prodiguer des conseils à l'utilisateur concernant sa réalisation si l'on est incapable d'évaluer la situation actuelle du cas de conception. Il existe dans certains SCAO des logiciels de simulation (permettant de détecter des dysfonctionnements), ou des logiciels de vérification des propriétés, mais ils ne conseillent pas réellement l'utilisateur.

Les aides que l'on trouve actuellement dans les SCAO, sont soit situées au niveau même de la fonction des outils utilisés, ou au niveau de leur utilisation [Coutaz 90, Myers 88, Shneiderman 87]. On peut dire que les SCAO sont constitués d'un ensemble d'outils pratiques de conception : logiciels de schématique, de calcul de structure, de simulations divers. Ils permettent en fait au concepteur de construire leur solution manuellement avec des outils informatiques dédiés à une tâche particulière de conception. Tous ces outils sont incapables de raisonnement concernant l'objet conçu ou de conseil sur des solutions possibles. Les systèmes d'aide actuels ne prennent pas en compte la situation courante du prototype ni le savoir-faire des concepteurs.

Il est clair que des aides du genre "dynamique" (Cf III.2.1.1), ou plus simplement de véritable systèmes d'aide à la conception nécessitent des capacités de raisonnement sur l'objet en cours de conception [Trousse 89]. D'ailleurs beaucoup se sont posés la question d'intégrer des capacités de raisonnement dans les SCAO, souvent pour proposer une solution à la réalisation d'un outil automatique de conception pour lequel les techniques algorithmiques classiques se révèlent impuissantes ou insuffisantes [Favard 89, Latombe 77, Orchampt 87, Tsang 87]. On se pose également de plus en plus la question de savoir comment intégrer le raisonnement dans les SCAO [Gardan 91, Trousse 89].

D'autre part, pour être capable de "raisonner", le SCAO doit prendre en compte des aspects plus sémantiques du produit concernant sa structure physique ou géométrique, ses aspects fonctionnels, ses relations avec son "monde" environnant. Les modèles de données utilisés en algorithmique classique permettent de traduire efficacement des modèles théoriques afin d'effectuer des calculs de type déterministe adaptés au problème à résoudre. Cependant ils se révélent toujours insuffisant pour représenter des connaissances de nature plus heuristiques comme le savoir-faire de l'utilisateur ou une modélisation plus riche des objets. Toutes les techniques IA de représentation des connaissances sont justement destinées à pouvoir traiter ce genre de connaissance. On notera à cet égard l'unanimité des chercheurs sur les modèles de réprésentation de type objet ou frame [Carree 85a, 85b, 89, Cholvy & Foisseau 83a, 85, Dugerdil 85, El Dashman & Barthes 90, Flemming & Coyne 90, Hanser 90, Haurat & al 90, Mohan & Kayshap 88, Montalban 88, Neveu & al 90, Petit 90, Rieu 86, Trousse 90a, Vogel 85]. De plus, les modèles de données utilisés en CAO sont essentiellement numériques et destinés à des traitements bien définis; ils ne tiennent pas

compte d'autres aspects nécessaires pour effectuer un raisonnement.

Enfin, les algorithmes de conception automatique qui existent dans les SCAO, comme les algorithmes de placement-routage dans le domaine électronique, ne traitent que le problème de conception complet, sont lourds et coûteux en exécution, et sont incapables de résoudre les problèmes complètement [Carre 85a, 85b, Favard 89, Orchampt 87].

On voit donc bien l'intérêt, sinon le besoin, à utiliser les résultats du domaine de l'intelligence artificielle pour apporter aux SCAO les capacités de raisonnement nécessaires à toute capacité de déduction ou de conseil d'un véritable système d'aide à la conception.

III.2.2 Particularités d'une interface dotée d'un système d'aide à la conception

Vouloir intégrer un système d'aide à la conception dans une interface de SCAO implique des modifications indispensables à l'architecture logicielle de cette interface, ainsi que des particularités de comportement interactif avec l'utilisateur. Notre but n'est pas ici de rentrer en détail dans ces différences nécessaires par rapport à un SCAO habituel. Mais nous précisons cependant les points importants concernant ces changement. Il s'agit du contrôle des modifications apportées par l'utilisateur, de certains types de contraintes que doit gérer le système, et de plusieurs points particuliers.

Nous ne serons pas exhaustif ici, notre but étant plutôt de montrer un éventail d'implications que nous avons détectées, et qui nous ont poussé à étudier et réaliser l'interface que nous présentons dans ce chapitre.

III.2.2.1 Exemple initiateur

Pour montrer les particularités d' un système d'aide à la conception, nous commencerons par un exemple élémentaire en CAO d'architecture.

Lors de l'interaction représentée sur la figure III.2, supposons que l'utilisateur désire déplace la porte P1 en P'1. Ce déplacement implique des modifications de contexte du produit : lorsque la porte est déplacée en P'1, un conflit de recouvrement apparaît entre la porte placée en P'1 et le mur perpendiculaire à la direction du déplacement de P1, il faut d'abord déplacer le mur faisant obstacle à P1 vant de pouvoir placer celle-ci en P'1. Déplacer le mur dans la direction opposée au déplacement de P1 risque de provoquer un conflit de recouvrement entre P2 et ce mur. Une solution simple pour la résolution de ce problème de placement consiste à déplacer le mur dans le sens de déplacement de P1 puisqu'ici cela n'engendrera pas d'autre conflit géométrique.

Si aucun déplacement n'était possible pour ce mur sans provoquer de conflit, on choisirait de stopper la tentative de résolution du problème, ou on pourrait par contre descendre d'un niveau dans cette résolution en tentant de déplacer les éléments qui gênent le déplacement du mur.

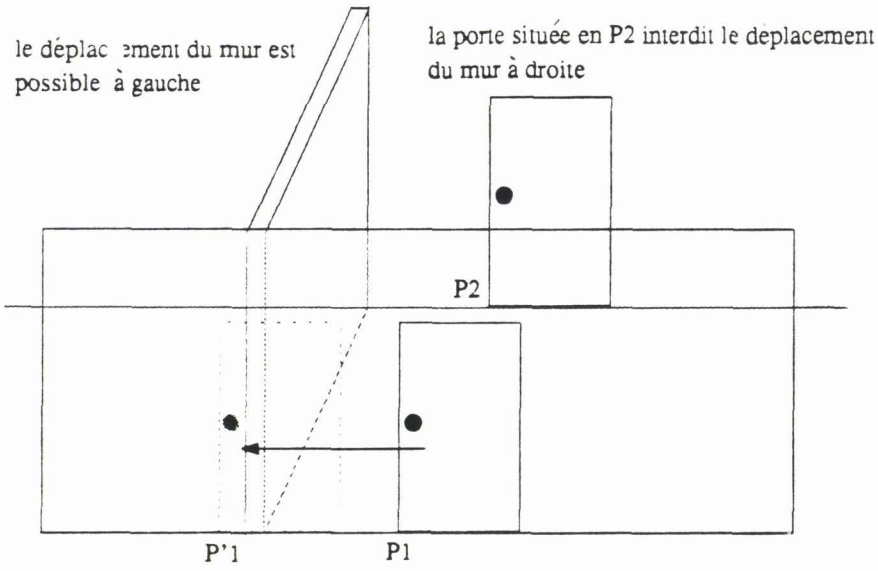


Fig III.2 influence topologique globale d'une modification locale du schéma.

remarques :

1. Comment faut-il présenter les modifications nécessaires du projet en cours ? on peut questionner graduellement l'utilisateur sur la validité d'une solution en construction, ou proposer directement une solution finale au problème. Cela dépendra du type d'interface utilisateur que l'on envisage.

2. Les composants du projet doivent-ils être envisagés dans la résolution du problème en tant qu'objets passifs que traite un moteur de résolution, ou bien actifs , c'est-à-dire participant directement à la découverte d'une solution?

Cet exemple montre qu'apporter une aide à la conception assurant la cohérence topographique de l'ensemble du projet implique à l'évidence l'introduction de techniques d'intelligence artificielle; dans le cas présent il s'agit de représenter les connaissances décrivant le projet, et de disposer d'un "moteur de résolution" adapté a cette représentation.

En fait, chaque action de l'utilisateur concernant directement ou indirectement l'état du projet peut avoir une influence globale sur cet état. En supposant que chaque composant participe activement à la résolution, le mécanisme de celle-ci peut s'envisager de la manière suivante :

- les premiers composants "touchés" par une incohérence locale déclenchent des mécanismes de raisonnement sur leur contexte local.

- les déductions engendrées par ce raisonnement peuvent porter sur des composants du contexte local des premiers, c'est à dire que les composants situés dans le voisinage des composants qui ont détecté une incohérence peuvent à leur tour être amenés à réagir à ce déséquilibre initial.

- De proche en proche, une grande partie des composants du projet est touché par cette "vague" de remise en cause.

La remarque précédente implique un phénomène fréquent et important dans les cas de conception où l'on utilise un ensemble de composants de base pour fabriquer un objet complexe : une simple modification locale peut remettre en question l'état global du projet par propagation de changement d'état de tous les composants. Ceci ne dépend pas de la technique utilisée pour représenter les connaissances et les objets du problèmes, mais plutôt de la nature du problème de conception en matière d'assemblage et d'aménagement spatial. Quelle que soit la technique utilisée pour résoudre le problème, il faut constater que l'on trouve une solution lorsque l'on trouve un état stable pour la totalité du problème. Or ceci n'est acquis que si tous les éléments participant au problème se trouvent eux-même dans un état stable. On cherche en fait à obtenir un état d'équilibre du système. Ceci est important l'une des raisons pour lesquelles on modélise si naturellement le systéme en le décomposant en éléments minimaux actifs : les composants du produit. Chaque composant est soumis à des contraintes de nature fonctionnelles ou géométriques, et trouver une solution au problème de conception étudié consiste à satisfaire toutes les contraintes qui pèsent sur cet élément.

III.2.2.2 Particularités de ce genre d'interface

Les interfaces intégrant une aide à la conception ont des particularités spécifiques. Parmi celles-ci, les suivantes nous semblent de première importance :

1. Le mécanisme de fonctionnemnet de l'interface est le suivant : l'utilisateur interagit avec le logiciel de la même manière qu'avec une interface habituelle. Cependant, chaque fois qu'une action est déclenchée (avec son paramétrage complet), une partie specialisée de contrôle doit analyser l'influence de cette action sur la sémantique du projet en cours; cette partie doit également détecter les erreurs de conception, de topologie si nécessaire (cela dépend du domaine de CAO), de fonctionnalités. l'utilisateur doit être informé de la nature des erreurs détectées, et si cela est possible doit conseiller l'utilisateur pour une solution éventuelle.

2. Le contrôle et le raisonnement doivent être faits sur des aspects topologiques du projet en cours, essentiellement en CAO électronique, ou en mécanique, ou en architecture.

3. Le contrôle et le raisonnement doivent également porter sur des contraintes sémantiques et fonctionnelles (par exemple une porte ne peut pas être proche de l'intersection de deux murs).

4. Lors de la détection d'un problème, l'utilisateur doit être informé de la nature de l'erreur détectée. Éventuellement, on pourra lui proposer immédiatement une solution simple si cela est possible.

5. Enfin le système d'aide est dirigé par les actions de l'utilisateur sur l'état du projet en cours.

III.2.2.3 Contraintes géométriques et topologiques en schématique

Dans le domaine de la schématique pour la CAO, nous rencontrerons plusieurs types de contraintes influant sur la description ou l'état du projet : des contraintes géométriques et/ou topologiques (la géométrie globale du schéma représentant le produit est elle cohérente), des contraintes sémantiques (le schéma représenté traduit-il correctement les fonctions du produit), et des contraintes fonctionnelles (est ce que chaque élément du produit joue correctement sa fonction).

Les contraintes de nature géométrique et/ou topologique sont naturelles pour l'utilisateur d'un logiciel de schématique. En effet, le fait de modifier un schéma du produit implique une modification sémantique du projet. Certaines contraintes géométriques ou topologiques doivent être respectées pour que les fonctionnalités du produit final soient respectées.

Dans l'espace à deux ou trois dimensions dans lequel est décrit le schéma, la cohérence géométrique et topologique du schéma doit être vérifiée. Voici quelques-unes des contraintes possibles de ce type :

1. **contrainte d'intersection** : deux objets du schéma ne peuvent pas s'intersecter.

2. **contrainte d'adjacence** : deux objets doivent, ou ne doivent pas, être adjacents d'un point de vue géométrique.

3. **contrainte topologique directe** : certains objets doivent ou ne doivent pas être liés par une relation topologique. Par exemple, l'insertion d'une porte dans un mur ne peut se faire que d'une seule manière.

4. **contrainte topologique indirecte** : des objets peuvent être reliés d'une façon topologique à cause de leur fonction propre, ou pour des raisons dépendant de l'expertise dans le domaine. Par exemple, il faut s'assurer que les arrivées d'electricité se font selon certains critères bien définis; elles ne peuvent pas toutes être regroupées en un seul lieu.

5. **contrainte morphologique** : certains objets ne peuvent être déformés, et d'autres devront l'être suivant de critères dépendant du projet et du domaine. Si l'on change la largeur d'une piéce, les murs de celle-ci devront se déformer dans une direction, mais ceci ne sera possible que si le matériau utilisé le permet.

Certaine de ces contraintes, comme les contraintes d'intersection, d'adjacence, et de morphologie, sont assez naturelles pour le concepteur, car elles font directement référence à l'aspect visuel du schéma. Les contraintes topologiques sont moins naturelles, parce que leur influence est ressentie indirectement sur la géométrie du schéma. Cependant, les experts du domaine reconnaissent et identifient très vite de telles contraintes, au contraire des novices.

Les contraintes fonctionnelles et/ou sémantiques sont issues de l'obligation de la part des

composants de tenir correctemnet et efficacement leur fonction dans la totalité du projet. Elles dépendent essentiellement des fonctions des composants dans le domaine de CAO choisi. Dans un schéma électrique par exemple, il ne doit pas exister de court-circuit, tout élément du circuit doit être alimenté. Chacune de ces contraintes dépend de l'objet composant manipulé. C'est parfois un groupement de composants qui créera une contrainte particulière sur l'un d'eux.

Il existe encore d'autres contraintes influant sur une partie ou la totalité des composants du projet : par exemple l'état de l'art dans un domaine peut impliquer l'indisponibilité de certains composants dans un domaine de compétence très voisin (par exemple électronique et électrotechnique), des contraintes de fabrication, ... Nous ne pouvons bien sûr pas toutes les connaître.

III.2.2.4 Contraintes fonctionnelles

Les contraintes fonctionnelles jouent un rôle spécial [Andre 86, Trousse 89]. Si dans beaucoup de cas elles ne font intervenir qu'une partie localisée du schéma, elles peuvent cependant intervenir parfois sur la totalié du projet et donc sur la topographie du schéma. De telles contraintes peuvent aussi avoir des effets de bord à retardement ou pernicieux. Les contraintes fonctionnelles sont donc à traiter avec beaucoup de précaution. Elles sont de plus difficiles à traiter, de par leurs effets de bord et du fait qu'elles dépendent principalement de l'expertise dans le domaine.

Mais avant tout autre type de contraintes, les premières que l'on doit traiter sont les contraintes géométriques, d'autant plus que les contraintes sémantiques ou fonctionnelles ont en fait une implication indirecte sur les aspects géométriques du projet, et donc sur les contraintes de même nature. C'est pourquoi nous ne sommes préocuppés dans notre travail que des contraintes géométriques entre composants, en se limitant d'ailleurs aux contraintes de non recouvrement entre composants.

III.2.2.5 Au confluent de l'interface homme-machine et des systèmes de satisfaction de contraintes

Le système d'aide à la conception que nous envisageons fait donc appel à une interface utilisateur évoluée, de manière à pouvoir manipuler directement les composants, et à un système de satisfacttion de contraintes dont la majorité sont en fait de nature géométriques.

En matière d'interface homme-machine, les techniques modernes font largement appel au concept de base d'acteur interactif [Coutaz 89, Herrmann & Hill 89], encore appelés interacteurs dans certains cas [Linton & al 89, Garnet & al 90]. Le principe de base de la construction d'interface est de composer des interacteurs pris parmi un ensemble prédéfini et de les organiser au moment de l'exécution du programme sous forme d'une hiérarchie d'instances de ceux-ci. Certains types d'interacteurs suffisamment évolués sont capables de placer seuls (au sens de l'aménagement spatial en deux dimensions) les interacteurs qu'ils dirigent suivant des règles prédéfinies et cablées dans des algorithmes.

Si certains d'entre eux sont capables de manipuler des contraintes spécifiées par le programmeur comme c'est le cas pour AVIS [Beaudoin-laffon & al 90], THINGLAB [Borning 86], PERIDOT [Myers 88], GARNET [Myers & al 90], INTERFACE BUILDER [Webster 89] par un moyen interactif ou non, ce qui est déjà une capacité importante pour pouvoir réaliser facilement des interfaces à manipulation directe [Shneiderman 87] intégrant des contraintes géométriques entre composants, Ils n'en sont pas pour autant capables de tenir compte des contraintes que pourrait induire les manipulations d'un utilisateur final ou des contraintes sémantiques dépendantes de l'objet manipulé. En effet la nature et le type des contraintes à satisfaire dans un système d'aide à la conception tel que nous le proposons change dès que l'utilisateur agit sur un composant de l'interface (si nous modélisons les composants de l'objet conçu comme un composant d'interface). Cette nature éminemment dynamique des contraintes ne peut pas actuellement être prise en compte facilement par les modèles d'interface homme-machine.

De ce point de vue, le modèle d'interface que nous proposons est capable de prendre en compte la sémantique du composant d'interface manipulé.

D'autre part le problème du placement de composants sur une carte de circuit imprimés se modélise assez naturellement par un ensemble de contraintes à satisfaire et les recherches actuelles [Andre 86, Baykan & Fox 87, Du Verdier 91, Orchampt 87] montrent leur possiblités pour fabriquer des systèmes plus capables de prendre en compte la globalité du problème et d'en trouver une solution satisfaisante.

Néanmoins nous pensons que de tels systèmes sont trop lourds et trop peu adaptés à un traitement interactif et local du problème, parce qu'il font appel à la totalité des données du problème de placement d'une part, et d'autre part parce qu'ils restent actuellement trop coûteux en temps d'exécution.

Le type de logiciel que nous proposons ici n'est d'ailleurs pas incompatible avec des systèmes prenant en compte la totalité du problème d'aménagement spatial, il en est plutôt complémentaire. Comme la majorité des systèmes de placement automatique proposent des solutions qui doivent être finalement remaniées en partie par le concepteur, l'interface que nous proposons serait bien adaptée au travail de modification final car c'est à ce moment que la complexité de l'objet conçu est la plus grande et que les zones libres de l'espace sont les moins nombreuses. Comme les modification finales provoquent des conflits géométriques plus fréquents, notre système peut se révéler d'une aide précieuse à ce moment.

En conclusion nous proposons un éditeur de schématique intégrant les fonctions d'un éditeur classique en CAO, mais capable de satisfaire des contraintes de natures diverses sur les composants constituants de l'objet conçu.

III.2.2.6 Domaines d'application visés

Les remarques que nous avons faites précédemment sont valables pour les trois domaines de CAO les plus courants : la CAO électronique, la CAO en architecture et bâtiment, la CAO

en mécanique.

En fait, ce type d'interface disposant d'une aide à la conception pourrait être développé pour n'importe quel domaine de CAO, mais nos remarques ne seraient sans doute pas toutes valables. Néanmoins, dans toutes les applications où il existe de la schématique, beaucoup des remarques que nous avons faites restent correctes.

III.2.2.7 Remarques sur l'implantation du système

Fonctionnement piloté par les données

Dans ce type d'interface, ce sont les objets composants ajoutés au schéma, ou déplacés ou copiés par l'utilisateur qui vont provoquer le déclenchement d'une règle, qui elle-même va faire appel aux objets du contexte général ou local et modifier l'état, voire même l'existence d'autres composants. On a donc un fonctionnement de l'aide qui est piloté par les données, plus exactement par les événements reçus par chaque composant.

Par exemple, le déplacement d'un composant va provoquer de la part de celui-ci une vérification de contrainte de non intersection avec tous les composants se trouvant dans son voisinage. On peut considérer que ces voisins recevront des messages de vérification de cette contrainte de non-intersection. Il en serait de même pour les contraintes d'adjacence.

Ce comportement piloté par les événements enregistrés par les composants nous a fait opter pour le choix d'un langage à objet, associé à des mécanismes de contrôle distribués.

Architecture logicielle d'une telle interface

La figure III.3 représente schématiquement l'architecture logicielle courante d'une interface offrant de l'aide à la conception.

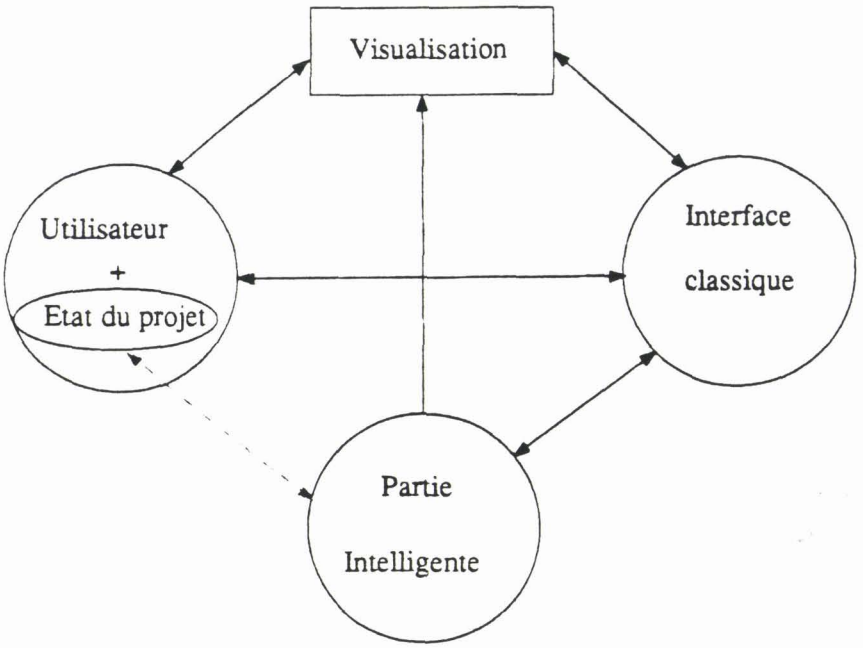


Fig III.3 architecture simplifiée d'un interface offrant une aide à la conception

dans ce modèle, la partie *Visualisation* est une entité finalement assez passive, qui montre des images repésentant le modèle de l'objet conçu et qui est dirigée par la partie *Interface classique* ainsi que par la *Partie intelligente*. Celle-ci agit directement sur la partie *Interface classique* en démarrant des commandes usuelles d'interface, ainsi que sur la partie Visualisation, pour présenter à l'utilisateur les résultats d'analyse ou les conseils. Elle est en relation indirecte avec l'état du projet, puisqu'elle est capable de modifier l'état des composants utilisés. L'*Utilisateur* joue quant à lui son rôle habituel de concepteur, en ayant de plus la possiblité de consulter ou de se servir de l'aide apporté par la partie intelligente.

Cette représentation montre les relations entre les composants de l'interface lors du fonctionnement de celle-ci. Elle ne préjuge en rien de la façon de construire une telle interface et des techniques IA utilisées.

Il y a donc trois acteurs principaux qui interagissent entre eux : l'utilisateur du système (*a priori* un expert du domaine de CAO envisagé), une partie d'interface tout à fait usuelle pour un système de CAO (offrant des fonctionnalités classiques à l'utilisateur), et une partie que nous avons citée comme " intelligente", car dotée de capacités de raisonnement à un degré plus ou moins grand.

Ce qui diffère essentiellement des autres interfaces de SCAO, c'est le fait que la partie intelligente est en permanence active, elle analyse et réagit si nécessaire à toutes les actions de l'utilisateur. Pour cela elle a des relations avec l'interface et avec l'état courant du projet.

III.2.3 Description élémentaire de l'interface

III.2.3.1 But de l'interface présentée

Le but principal de l'interface que nous avons réalisée est d'apporter une aide de conception à l'utilisateur. Nous ne nous sommes pas attachés à apporter une aide faisant intervenir une expertise riche. Nous avons en fait intégré à cette aide une expertise sommaire permettant de résoudre des conflits géométriques entre composants.

Une conséquence importante de ce travail a été l'étude d'un modèle de représentation des composants permettant de résoudre localement et de manière distribuée les conflits (Cf. III.3).

Nous avons vu la nécessité d'intégrer dans un tel système des connaissances expertes du domaine, de manière à pouvoir évaluer un état du produit en cours de conception Un autre but important de cette réalisation a été de trouver et de faire fonctionner un modèle particulier de représentation des connaissances pour pouvoir faire fonctionner l'interface normalement et pour résoudre les conflits dynamiquement.

D'autre part, la société METADESIGN désirait apporter un plus dans son logiciel de CAO électronique METADESIGN-CAE. Une étude du mode d'utilisation du logiciel fit remarquer la quantité très importante d'interactions routinières impliquées par une modification locale de composants sur la carte. Il fut alors envisagé de réaliser un système d'aide pour éviter cette quantité importante d'interactions nécessaire pour déplacer les composants après modification. C'est ce que doit réaliser cette interface au moyen d'un environnment de prototypage qui nous a permis d'implanter tous les concepts importants du modèle : le système Smalltalk.

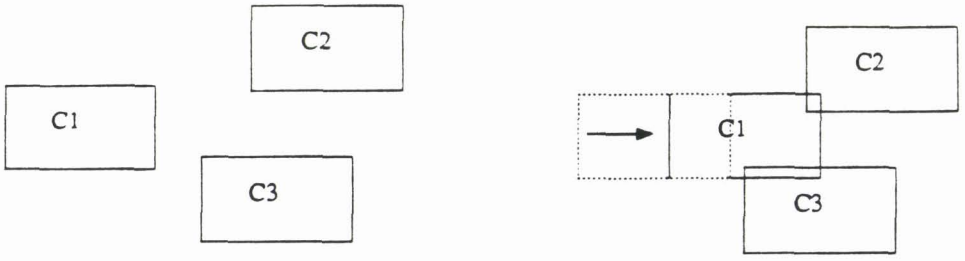
Enfin, il était prévu une réalisation en grandeur réelle sur le système de CAO électronique, qui devait être écrite en langage ADA, ce qui a impliqué certains choix pour la réalisation du système, en particulier l'organisation et le nombre des classes implantées. Nous y reviendrons en fin de ce chapitre.

III.2.3.2 Domaine applicatif choisi

Le domaine de conception choisi est donc la conception de cartes de circuits imprimés. Plus précisement, il concerne le placement de composants et la modification de ce placement sur une telle carte.

Le sous-ensemble étudié concerne uniquement le placement de composants rectangulaires dans un espace à deux dimensions. Le système réalisé doit être capable de résoudre les conflits de recouvrement géométrique lorsque l'utilisateur en provoque en déplacant l'un des composants existant à un moment donné de la conception.

La figure III.4 précise exactement ce qu'est un conflit géométrique entre composants.



Avant déplacement de C1 vers la droite

Après déplacement de C1 vers la droite, il y a conflit entre C1 et C2, entre C1 et C3

Fig III.4 Un conflit géométrique entre composants, après déplacement d'un composant par l'utilisateur

De plus, nous avons volontairement ignoré les connexions entre composants, toujours pour des raisons de simplification du problème.

III.2.3.3 Configuration matérielle choisie

Nous avons travaillé dans un environnement de machine compatible PC, munie d'une carte graphique de définition moyenne, et d'une souris standard sur ce genre de machine. Le logiciel utilisé pour le développement est une implantation du langage SMALLTALK, à savoir SMALLTALK/V conçu et vendu par la société américaine DIGITALK [SMALLTALK/V 87].

III.2.3.4 Fonctions de l'interface

L'interface seule dispose des commandes usuelles de manipulation du projet. La figure III.5 montre une image du système à un instant donné; on remarquera plus particulièrement la zone de messages qui contient des informations sur le composant sélectionné. Les commandes sont organisées en trois groupes :

1. Le premier groupe concerne la création d'un composant et des fonctionnalités sur l'ensemble du projet. La création de composant (**créer**) fait partie de cette catégorie de commandes car dans tout système de CAO, on ne crée pas un composant, mais on l'instancie à partir de la base de composants du système. Ainsi nous avons considéré que cette instanciation faisait partie des actions concernant le projet en entier. Cependant, il est clair que la création d'un composant nécessite également son placement dans l'espace et donc peut provoquer des conflits géométriques.
   Les autres fonctions de ce groupe sont :

   a. la vérification de la cohérence du projet (nous reviendrons sur ce point par la suite)(**cohérence**),
   b. le pas de grille affiché et permettant un placement précis et simple des composants dans l'espace (**grille**),
   c. le choix d'un composant existant (**choisir**), pour ensuite le manipuler,
   d. le rafraichissement de la fenêtre d'édition (**rafraichir**).

2. Le deuxième groupe de commandes concerne la manipulation de l'enveloppe du rectangle définissant le composant, c'est à dire changer la position du coin haut gauche d'un composant (**coin**), ou les deux coins (**enveloppe**).

3. Enfin le dernier groupe de commandes concerne des opérations plus spécifiques d'une action sur un composant. Il s'agit du déplacement d'un composant (**déplacer**) dans une direction horizontale ou verticale, du déport d'un composant dans une direction quelconque (**déporter**), de la destruction d'un composant (**détruire**).

Le lecteur trouvera en Annexe III-1 la description complète des fonctions de l'interface..

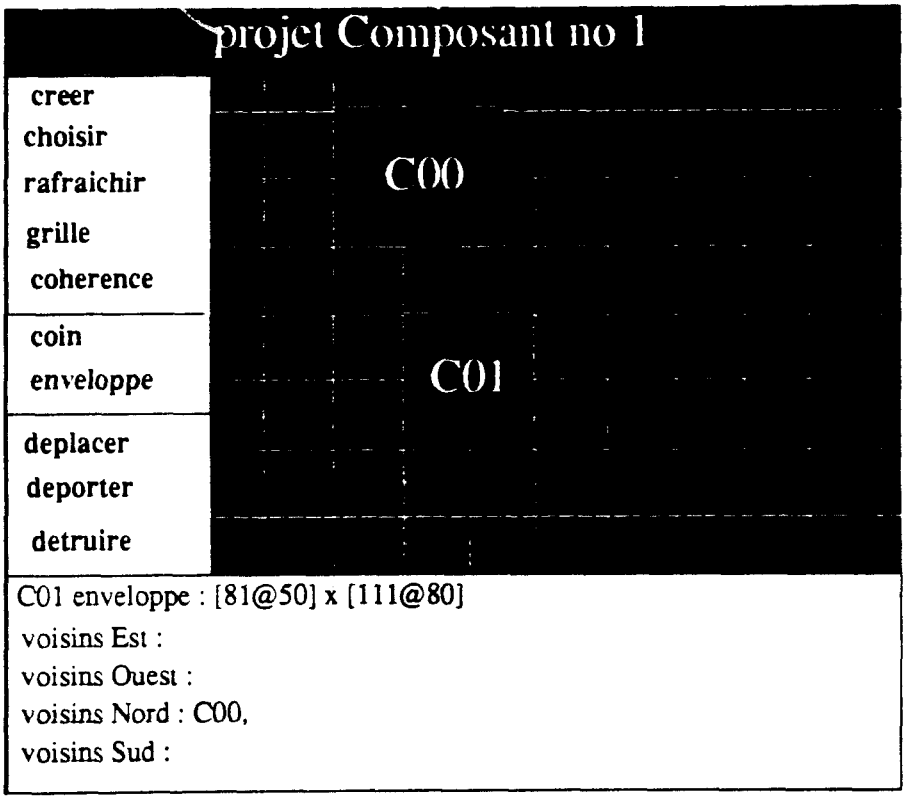


Fig III.5

III.2.3.5 Contenu de l'aide à la conception fournie

Pour réaliser cette interface, nous avons fortement simplifié le problème pour plusieurs raisons :

- pour traiter le problème au complet, il faut être un expert en placement-routage,
- complexité du problème. Si nous avions voulu résoudre complètement le placement des composants et le routage des connexions, nous aurions dû faire intervenir à la fois les problèmes de topologie dûs au schéma, mais aussi les contraintes fonctionnelles et sémantiques du domaine,
- but du travail : il s'agissait de fournir une aide simple à la conception, pour éviter tout le travail routinier effectué régulièrement par l'expert en placement-routage lors d'une modification.
- réaliser une première implémentation d'un modèle de résolution particulier où chaque composant est actif lors de la résolution (nous y reviendrons plus loin).

Finalement, nous ne manipulons que des composants rectangulaires de taille et de géométrie variables. Aucune réalisation sérieuse n'a été faite en ce qui concerne les connexions entre composants, bien que nous ayons en partie étudié le problème.

L'aide fournie se résume donc de la manière suivante : pour tout conflit entre un composant manipulé par l'utilisateur et d'autres composants existant dans l'état courant du projet, tenter de le résoudre en proposant interactivement une solution à l'utilisateur.

III.2.3.6 Avantages et inconvénients de cette aide

Malgré les impératifs qui nous ont poussé à simplifier de cette manière le problème, nous pouvons quand même citer quelques avantages et inconvénients de cette aide simplifiée.

Avantages

1. l'utilisateur peut travailler comme il le ferait avec une interface classique et peut activer la fonction d'aide quand il le désire. Dans ce cas, le système le prévient en cas d'incohérence ou de problème fonctionnel, et lui demande si il désire qu'on lui propose une solution. En bref, un tel système doté de fonctions semi-automatiques ne remet pas en cause l'utilisation habituelle des interfaces,
2. la simplification du problème permet une uniformisation du comportement des composants, et rend donc plus aisée la réalisation du système.

3. la réalisation est envisageable dans un environnement logiciel de faible capacité. En particulier, nous avons pu réaliser ce travail dans un environnement de prototypage SMALLTALK sur micro-ordinateur.

4. dans bien des cas, une résolution locale est possible. De ce fait, le danger de voir la résolution du conflit se propager à l'ensemble du projet est minime.

Inconvénients

1. Malgrè les simplifications supposées sur le problème (traitement des recouvrements de composants rectangulaires, la prise de décision risque d'être coûteuse en temps de calcul. Ceci sera d'autant plus vrai si l'on veut intégrer au mécanisme de résolution la prise en compte des points de vue fonctionnels et sémantiques concernant le domaine d'application. Pour résumer, le système risque de réagir lentement et donc de ralentir quand même l'interaction.

2. La simplification excessive (notamment au niveau de la seule prise en compte des contraintes géométriques) peut rendre caduque l'intérêt de l'aide apportée. En effet, l'expert peut ne jamais être satisfait des solutions proposées par le système, car dans certains cas les solutions proposées seront en contradiction avec les contraintes fonctionnelles et sémantiques du projet. Néanmoins pour être capable de prendre en compte des contraintes dépendantes du domaine d'application, il faut déjà être capable de résoudre les problèmes les plus simples. D'autre part nous sommes convaincus qu'une grande partie des cas d'aménagement à résoudre pourra être résolue de manière locale. Dans cette optique, l'aide offerte au concepteur sera opérationnelle et satisfaisante.

3. Dans beaucoup de cas routiniers (cas sinples et fréquents), une telle approche est satisfaisante. Cependant, on n'est pas toujours sûr de garantir une solution qui satisfasse pleinement le concepteur, si l'on n'explore pas toutes les solutions possibles.

III.3. Modélisation distribuée du problème : le modèle MAPS

III.3.1 Introduction

Nous présentons dans cette section le modèle distribué que nous avons conçu et implémenté pour déterminer les "relations sociales" de nature géométrique entre les composants (que nous appellerons indifféremment agent ou composant dans la suite du document) et en maintenir la cohérence lors d'une modification sur un composant.

MAPS signifie "**M**ulti-**A**gent model for Interactive **P**lacement **S**ystems", et comme son nom l'indique fait appel aux idées de base suivante :

1. modèle distribué basé sur la notion d' "acteur",
2. système interactif offrant une interface à manipulation directe,
3. placement de composant (aménagement spatial),
4. communication par envoi de message exclusivement.

Dans cette section, nous décrivons le modèle implanté dans MAPS, sans décrire l'aspect interface qui reste classique du point de vue utilisateur.

Nous commencerons par donner les raisons de notre choix d'architecture distribuée, puis introduirons la notion de "voisinage" qui *socialise* en quelque sorte les composants. Nous décrirons ensuite le voisinage que nous avons choisi pour MAPS, d'abord d'une manière informelle en faisant ressortir les caractéristiques importantes, et ensuite de façon plus détaillée avec un formalisme spécifique. Enfin nous expliquerons les mécanismes internes de gestion des accointances d'un agent et de conservation de la cohérence des relations entre agents.

III.3.2 Le choix du modèle distribué pour le raisonnement

Nous avons choisi d'implanter un moteur distribué pour la résolution des conflits. Nous expliquons ici les raisons de ce choix.

III.3.2.1 Trois approches possibles de modélisation du problème

Approche centralisée classique On peut vouloir résoudre le problème en utilisant les techniques classiques d'intelligence artificielle. Par classique, nous entendons un moteur de résolution centralisé qui agit en temps que maître. C'est par exemple le cas dans TROPIC [Descottes 81, Latombe 77, Tsang 87].

Dans cet esprit, la base de faits est représentée par l'état du projet en cours de modification et par le plan d'actions en cours d'élaboration. La base de connaissance est indépendante, elle permet d'évaluer et de corriger le plan, de provoquer des retours-arrière si nécessaire. Les composants sont alors passifs vis-à-vis de la résolution du problème, leur état véritable n'est changé qu'en fin d'exécution du plan d'action élaboré par le générateur choisi. La génération du plan d'action pourra se faire en utilisant les techniques actuelles adaptées à la résolution spatiale.

Approche distribuée Dans une approche distribuée, nous aurons une société d'experts qui va tenter de résoudre le conflit. Ces experts peuvent être de nature différente, spécialisés dans la résolution, ou dans la critique de la solution, ou encore dans l'analyse et la déduction des communications nécessaires entre agents, c'est à dire le choix des agents qu'il est pertinent de lier à un autre agent.

Une autre approche possible (celle que nous avons choisie ici) est de partir de l'hypothèse que les agents doivent être tous placés au même niveau, qu'ils disposent de la connaissance de leur environnement local, c'est-à-dire des composants avec lesquels la communication sera pertinente, et que leur capacité de raisonnement sur le problème est la même pour tous. Dans ce paradigme, il devient naturel (sinon obligatoire) de voir chaque composant lié à un agent unique. Chaque agent est alors déterminé par son état et par rapport à ses accointances
En effet, si chaque agent avait une vue privilegiée sur plusieurs composants, il faudrait que celle-ci soit étendue à tous les composants pour que chaque agent ait une vision identique du monde. Ceci voudrait dire que chaque agent serait capable de résoudre seul la totalité du problème, d'une manière centralisée, ce qui ferait perdre tout l'intérêt du paradigme envisagé.

Il faut donc que chaque agent soit lié à un composant unique. Les agents communiquent alors entre eux au moyen de messages, ils se renseignent auprès des autres pour connaître diverses données dont ils n'ont pas connaissance, ou pour leur déléguer une partie de la résolution du problème. Ils disposent d'une connaissance locale du monde et travaillent à partir des mêmes connaissances expertes.

Dans ce cas, la base de faits, qui représente l'état du monde à un instant donné, est également distribuée parmi les agents. Les agents peuvent travailler en coopération volontaire ou en coopération conflictuelle. La communication entre agents se fait de manière asynchrone mais en respectant l'ordre d'envoi de messages vis-à-vis des réceptions. Quand à la génération de plan d'action, le contrôle de son exécution, et la critique du plan engendré, ils seront réalisés partiellement par chacun des agents, en coopération avec les agents avec lesquels le composant est en relation.

Une remarque importante à faire est que les agents "vivent" et "meurent" dynamiquement. En effet, s'il existe un agent par composant existant, et que l'utilisateur de l'interface peut créer et détruire des composants à volonté, le nombre d'agents va varier dans le même nombre que les composants.

Une autre remarque importante concerne le fait qu'il doit exister une entité disposant d'une vue globale de l'ensemble des agents. En effet, lors de la création dynamique d'un agent en même temps que du composant correspondant, il faut être capable de décrire la vue du monde local du composant. Pour cela, il faut interroger l'ensemble des agents pour savoir s'ils font partie de ce monde local. Il faut donc disposer d'une entité qui connaisse tous les agents existant à un instant et soit capable de les interroger.

Approche intermédiaire Il va s'agir ici d'une coopération à la résolution avec un nombre limité d'agents [Trousse 89]. La différence principale avec le modèle précédent est que les agents du modèle n'ont pas la même connaissance ou les mêmes compétences. Leur nombre est déterminé d'avance par leur type de spécialité.

Certains agents auront une connaissance globale du projet, par exemple pour sa topographie, ou pour la détermination des contextes locaux des composants, d'autres auront une connaissance relevant plus de l'expertise du domaine ou de la génération de plan d'actions.

La connaissance et la résolution seront donc dispersées entre plusieurs types d'agent. Dans un tel modèle, on ne peut pas préjuger d'avance de ce que seront la base de connaissance et la base de faits, on peut penser qu'on disposera en fait de plusieurs bases de faits et de plusieurs bases de connaissance, chacune d'entre elles étant dédiée à un type particulier d'agent. La génération de plan se fera par un agent unique.

III.3.2.2 A propos de l'intelligence artificielle distribuée

Les systèmes d'intelligence artificielle distribués sont une branche parallèle de l'intelligence artificielle. Ils visent à décrire et faire coopérer des société d'experts. Quelques paragraphes sur des points clés du domaine de l'intelligence artificielle distribuée nous permettrons de faire un rapide tour d'horizon du domaine. Nous conseillons la lecture du rapport de O.BOISSIER [Boissier 90], des travaux de J.Ferber [Ferber90], ou encore de L.Buisine[Buisine 89],pour disposer d'une vue plus approfondie du domaine ainsi que d'une bibliographie riche dans ce domaine..

Définition

Pendant de nombreuses années les chercheurs en intelligence artificielle n'ont essayé de modéliser qu'un seul agent intelligent, selon le paradigme de NEWELL [Newell 62]. Mais le monde réel est le lieu privilégié d'interactions entre ces des agents intelligents : pour satisfaire des contraintes temporelles, fonctionnelles, spatiales, pour répartir une connaissance entre differents spécialistes, pour confronter différents points de vue visant à résoudre un problème.

D'après O.BOISSIER [Boissier 90], la problématique de l'IAD (intelligence artificielle distribuée), nouveau domaine de l'intelligence artificielle, peut être énoncée ainsi : "*une société d'experts intervenant dans la résolution, chacun agissant soit en coopération avec les autres, soit en concurrence avec eux. Chacun d'eux n'a que des connaissances limitées sur son environnement, et sur les intentions ou actions des autres*".

Différents domaine d'application

Résolution distribuée de problèmes Elle concerne des situations où plusieurs experts coopèrent pour atteindre un seul but, identique pour tous. Chaque agent dispose de connaisssances différentes, et de quantités de ressources et de traitement variables [Demazeau 90, Ferber & Ghallab 88, Gleizes & al 89, Rosenschein 84].

Planification en univers multi-agents C'est typiquement une situation où les agents interagissent entre eux au travers de la modification d'un univers commun, entrainant ainsi la planification d'actions, leurs synchronisation. Le but est alors de régir les interactions entre agents.

Assistance mutuelle Au cours d'une assistance mutuelle [Worden & others 86], un expert propose une solution à un autre expert et celui-ci la critique. Le premier expert peut alors tenir compte de ces critiques pour améliorer sa solution, et peut ainsi recommencer le processus, ou valider cette solution.

Problèmes de l' IA distribuée

Au niveau de la société d'agents, on peut s'intéresser à un expert de manière microscopique, c'est-à-dire en tant qu'individu ayant une connaissance et des capacités de résolution limitées, ou à un niveau macroscopique, qui prend en compte la société d'expert complète.

interaction entre experts Les interactions entre experts peuvent être de type coopératif ou conflictuel [Buisine 89, Ferber & Ghallab 88]. La *coopération volontaire* d'un agent, ou comportement du "benevolent agent" [Smith 85], est une situation où tous les experts ont des buts identiques et non conflictuels; ils s'aident les uns les autres et échangent informations et ressources sans difficultés.

Dans une situation de *coopération conflictuelle* [Rosenschein 84, Ferber & Ghallab 88], les agents ne sont pas tous forcément prêts à s'entraider, à partager des informations et des ressources, et n'ont pas toujours les même désirs ou les mêmes buts.

Actuellement, seule l'hypothèse du "benevolent agent" a réellement été exploitée dans les systèmes d'IAD.

On peut classifier la nature des interactions en trois types : par échange de connaissance, par échange de travail, par partage de ressources.

Résolution et conduite de la coopération Le processus de résolution peut se faire en décomposant le problème initial, en distribuant les sous-problèmes aux différents experts, en faisant résoudre des sous-problèmes par des agents uniques, en faisant une synthèse des résultats par aggrégation des sous-résultats.

Deux techniques de coopération sont couramment utilisées : par partage de résultats, et par partage de tâches. On rencontre cependant deux types de problème pour ces modes de coopération : trouver une bonne répartition de la charge et un bon processus de direction de la recherche.

La structuration entre les experts peut être de type *hétérarchique*, cas où il n'y a pas de lien précis entre les experts (c'est le cas des langages d'acteurs), de type *hiérarchique* comme dans Hearsay-II[Erman & al 80, 81], de type *marché* où les liens entre experts s'organisent dynamiquement [Buisine 89, Demazeau 90, Ferber & Ghallab 88] comme c'est le cas ici.

Trois modèles typiques

Le modèle acteur Le modèle Acteur a été introduit par C.HEWITT [Hewitt 71] au travers du langage Planner. Un acteur est une entité informatique non décomposable, qui communique avec d'autres acteurs par envoi de messages unidirectionnels, c'est à dire sans attente de réponse. La transmission entre les acteurs est asynchrone et non déterministe.

A la réception d'un message, un acteur ne peut que le transmettre à ses connaissances, changer son état interne ou créer de nouveaux acteurs. Il peut également déléguer une tâche ou des résultats par envoi de message. L'intérêt des acteurs dans un système parallèle est de disposer d'une bonne modularité de comportement des acteurs, de localiser la connaissance, et de séparer leur programmation de leur utilisation.

ABCL Abcl est un descendant du modèle acteur [Dang 86]; il est employé en programmation concurrente objet et peut être appliqué à la résolution distribuée de problèmes, à la planification, à la conception de systèmes temps réel.

Dans ce modèle, les objets ont les même propriétés que les acteurs. Mais chaque objet dispose d'une mémoire locale qu'il peut modifier et dont le contenu représente l'état de l'objet. l'ordre temporel des réceptions de message est le même que celui de l'envoi. Les types de message peuvent être distingués par la rapidité de livraison et par l'attente ou pas d'un message de réception. Un objet peut du coup être endormi, actif ou en attente.

Des mécanismes de synchronisation et des instructions dédiées au parallélisme sont disponibles.

Le modèle du contract net protocol Ce modèle [Smith 80] est basé sur le principe de l'offre et de la demande dans une société d'agents intelligents. L'interaction se fait grâce à

une négociation entre experts, et le problème est décomposé en un réseau de tâches, où tous les noeuds utilisent le même langage de description des tâches.

Les négociations entre les parties intéréssées se décomposent en trois phases : un échange d'information dans les deux sens, cette information est évaluée séparément par chaque partie, l'accord de communication est obtenu par sélection mutuelle. Chaque noeud peut avoir un rôle de contractant, c'est-à-dire qu'il est responsable de l'exécution de la tâche, ou de manager, auquel cas il a pour responsabilité de superviser l'exécution du travail. Les noeuds peuvent à la fois être contractant et manager.

Planification en univers multi-agents

D'après Boissier [Boissier 90], certains des problèmes de l'IA distribuée se retrouvent sous la forme suivante dans les contextes multi-agents :

- chaque agent doit être capable de planifier et d'inférer des buts à la place d'autres agents. Il faut donc connaître les plans des autres agents,
- chaque doit être capable de s'adresser à d'autres agents pertinents,
- il faut pouvoir créer de nouveaux buts en cas de situation bloquante,
- résolution de problèmes d'interaction au travers de règles de comportement,
- les agents doivent disposer non seulement d'une représentation du monde et des autres agents, mais également de ce que les autres agents savent d'eux mêmes,
- la communication entre agents est importante et il faut aussi prévoir de la planifier,
- il faut être capable de planifier dans des cas où des actions non prévues par l'agent se sont produites,
- la planification en univers multi-agents dégrade rapidement les performances d'un système purement distribué, il faut développer des techniques particulières.

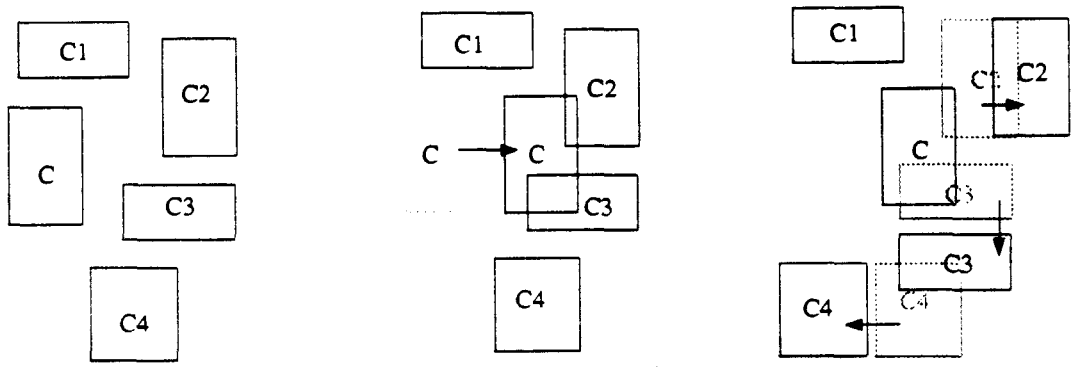
Dans [Corkill 79], l'utilisation du générateur de plans hiérarchiques NOAH [Sacerdoti 75] se sert au mieux des possibilités de celui-ci dans un univers distribué : la génération du plan global peut se faire localement sur chacun des agents, et sur chaque site est implanté un générateur de plans; par contre la prise en compte des interdépendances se déroule dans une phase de critique, et se fait à un niveau plus ou moins global; l'exécution du plan produit peut se faire de manière parallèle, car NOAH est un générateur non linéaire; enfin l'espace de recherche est restreint par la nature hiérarchique du plan engendré par NOAH.

Dans Genplan [Bessiere 83], on détermine d'abord les agents pertinents pour l'agent

qui planifie, on engendre un plan correspondant dépendant de la tâche à atteindre et de la connaisssance de l'agent, on termine par une critique des plans qui se pratique de manière locale. Par contre, la conduite de l'exécution du plan n'est pas faite localement par chaque agent, contrairement au cas précédent.

III.3.2.3 Remarques sur le problème des conflits géométriques entre composants

Explicitons tout d'abord un exemple complet de résolution de conflits géométriques entre composants par le dessin suivant :



Situation initiale sans conflit

C est déplacé vers la droite, il est alors en conflit avec C2 et C3

Une solution possible au conflit précédent

Fig III.6 Une situation conflictuelle résolue

Dans cet exemple, où le composant C joue le rôle d'initiateur du conflit, on part d'une situation où il n'y a aucun recouvrement entre les composants (situation initiale). Si l'utilisateur déplace C vers la droite de la quantité spécifiée sur le schéma, on arrive alors à une situation conflictuelle où C entre en recouvrement avec C2 et C3.

Dans la partie droite du dessin, on a figuré le résultat possible d'un mécanisme intelligent de résolution de recouvrement géométrique. Le composant C2 s'est déplacé vers la droite sans rencontrer d'autre composant qui aurait pu engendrer un autre conflit. Par contre, le composant C3, pour pouvoir être déplacé vers le bas devait engendrer un conflit avec C4, ce qui implique un déplacement inattendu de C4, qui ne devait pas a priori être touché par les recouvrements existant entre C, C2 et C3.

Ce problème peut ressembler à première vue au jeu classique du taquin à neuf cases [Nilsson 83, Rich 83]. Cependant, dans notre cas, l'espace de recherche n'est pas limité puisque les composant peuvent être replacés dans un espace a priori illimité et sans contrainte de positionnement à des endroits connus d'avance.

En fait, il s'agit ici d'un problème de "*planification*" : on part d'une situation initiale, caractérisée par deux conflits géométriques, et il faut trouver un plan d'actions menant vers une solution, c'est-à-dire aboutir à une nouvelle situation où il n'y aura plus de conflit. On doit donc effectuer une résolution *spatiale*, et trouver un plan d'actions pour effectuer cette résolution.

Pour résoudre ce type de problème, on peut donc faire appel aux techniques connues comme le générateur de plans d'actions NOAH [Sacerdoti 75, 77], TROPIC [Latombe 77], ARGOS-II [Farreny 80]. On remarquera cependant que dans ce type de problème, la génération de plan n'est pas linéaire, car d'une part il n'existe pas qu'une seule solution au problème, et d'autre part il peut exister plusieurs chemins pour aboutir à la même solution. Il vaudrait donc mieux adopter un générateur de plan d'actions non linéaire de type NOAH

Cependant, le problème étant purement topologique et spatial, il est vraiment naturel de voir en chaque composant un agent expert capable de raisonner sur son environnement local. C'est ce que nous avons fait dans ce travail, alors que nous aurions pu utiliser les techniques de planning existantes comme celle qu'il existe dans NOAH. Nous reviendrons en détail sur le contenu du modèle choisi dans un prochain paragraphe.

Nous allons maintenant décrire succintement quelques possibiltés de modélisation de la résolution.

III.3.2.4 Aspects géométriques et topologiques du placement de composants

géométrie des composants manipulés

En schématique logique de circuit imprimé, on manipule essentiellement des symboles normalisés représentant des composants de base, du genre de ceux indiqué sur la figure III.7

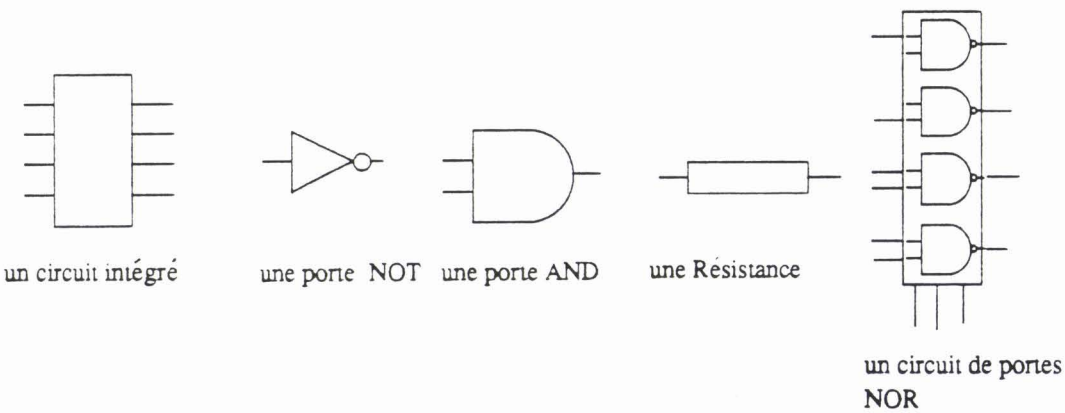


Fig III.7 quelques symboles courants en schématique électronique de circuit logique

En plus de ces symboles, les connexions sont representées de manière filaire, avec des changements de direction à 45 ou 90 degrés.

Par contre, dans la phase de placement-routage, les seules entités géométriques manipulées sont le rectangle (entouré d'une série de points de connexion), et le trou de perçage representé par un symbole normalisé de petite taille. Nous ferons ici abstraction des trous de perçage.

On remarquera que tout symbole peut être manipulé logiquement (du point de vue de son placement) comme un rectangle avec points de connexions, rectangle qui sera l'enveloppe du symbole complet. Ceci restreint néanmoins les possibilités de solution au placement puisque les connexions peuvent être représentées comme une suite de segments brisés de longueur variable dont l'orientation est horizontale, verticale, ou oblique à 45 degrés. Ainsi une portion de connexion peut logiquement passer à l'intérieur du rectangle enveloppe d'un symbole comme une porte NOT sans causer d'impossibilité ou de court-circuit. L'introduction d'un rectangle enveloppe interdit toutes les solutions de ce genre.

En placement — routage, il existe aussi les connexions (ou pistes) entre composants, que l'on trouve sous forme d'une ligne brisée d'épaisseur variable. D'autre part, nous ne tiendrons pas compte ici des problèmes relatifs aux connexions entre composants.

Topographie d'une carte

Les divers symboles représentant des composants sont répartis dans l'espace rectangulaire limité de la carte. C'est la fonctionnalité du circuit à concevoir qui va induire pour l'expert le placement et la répartition des composants sur la surface disponible.

Certains composants seront cependant souvent situés en des points précis de la carte (cas des alimentations), ou regroupés ensembles suivant leur fonction (cas des circuits de mémoire).

Aspect topologique

Du point de vue topologique, la règle primordiale est de ne jamais avoir de recouvrement de composants, que ce soient entre symboles, entre connexions, ou entre connexions et symboles. Si deux connexions se touchent ou se croisent, on provoquera un court-circuit. De même, on ne peut pas disposer deux symboles l'un sur l'autre. Parfois une connexion pourra "traverser" un symbole dans certaines conditions (seul passage possible pour une connexion)

Note : On peut considérer que l'espace dans lequel se trouve les composants est une partie d'un espace euclidien orienté à deux dimensions.

III.3.2.5 Aspects topologiques secondaires

Le problème à résoudre étant restreint aux seuls composants à placer correctement sur la carte, uniquement en vue de satisfaire les contraintes topologiques directes citées dans le paragraphe précédent, on doit cependant remarquer que certaines de ces contraintes sont indirectes ou plus liées à la carte elle-même qu'aux autres composants.

Il s'agit de contraintes sémantiques et fonctionnelles induisant des contraintes topologique fortes. En voici trois exemples.

1. Un *regroupement de composants* est parfois souhaitable. Les contraintes de positionnement relatif de ces composants deviennent alors prioritaires pour conserver la cohésion du groupe. Si un seul de ces composants doit être modifié ou déplacé à cause d'un autre composant externe au groupe, il faudra éviter de détruire la topographie courante du groupe, si cette contrainte est essentielle.

2. certains composants sont *incompatibles entre eux*. C'est à dire que leur bon fonctionnement serait remis en cause par leur proximité géométrique, car des effets électriques nuisibles peuvent alors détruire leur fonction. C'est particulièrement vrais pour les circuits d'alimentation qui peuvent par des effets électromagnétiques ou électriques perturber le fonctionnement d'autres composants. Ceci induira donc des contraintes topologiques d'éloignement entre ces composants.

3. Enfin, il existe des *lieux privilégiés*. En effet, la carte devant être en communication électrique avec le monde extérieur, le positionnement de circuits comme les alimentations se fait en tenant compte de l'appareil dans lequel doit être intégrée la carte. Cela est aussi vrai pour des composants d'encombrement physique important qui imposent de disposer d'une grande place pour pouvoir être placés.

Ces trois exemples montrent l'importance prise par les contraintes fonctionnelles sur la conception de la carte. Le respect de ces contraintes fait en réalité partie de l'expertise ou savoir-faire spécifiques du domaine d'application et diffèrents du savoir-faire nécessaire pour résoudre le problème initial. Pour les respecter, il faut être capable de faire des retours—arrière lors de la génération des plans d'action. Ce type de contrainte joue un rôle particulier lors de l'évaluation du plan d'action généré, et ceci est d'autant plus vrai que la carte est complexe.

Notre problème n'étant pas d'inclure l'expertise en électronique d'un placeur-routeur, nous avons également ignoré ce genre de contraintes.

III.3.2.6 Le modèle choisi dans cette interface

Dans cette interface, nous avons choisi de concevoir un système totalement distribué, nous en précisons ici les raisons.

D'une part, nous avons dit que nous simplifions le problème le plus possible, de manière à disposer d'une efficacité raisonnable dans les temps de réaction de l'interface. Ainsi, la minimalisation du problème impliquait que nous ne nous intéresserions que très peu à l'expertise en électronique nécessaire pour traiter le problème dans sa totalité. Alors l'approche intermédiaire présentée au paragraphe III.3.2.1 ne pouvait être choisie, car elle aurait forcément nécessiter de prendre en compte des aspects relevant beaucoup de l'expertise du domaine.

Le choix d'un modèle complètement centralisé était possible, auquel cas nous pouvions résoudre partiellement le problème, et aborder seulement la question de la résolution des conflits géométriques. Cependant ce modèle ne tire pas parti de la modularité des composants,

et implique un traitement global du problème. C'est un expert centralisé qui construit les plans d'actions, détermine les composants pertinents d'un autre composant, évalue le plan d'action choisi, et le réalise. C'est pourquoi nous n'avons pas choisi ce modèle.

Enfin le modèle décentralisé s'adapte bien au raisonnement naturel pour un problème d'aménagement spatial en cas de modification de l'aménagement : lorsqu'un ensemble de composants est déjà placé, et que l'on veut en modifier la topographie par déplacement ou changement des caractéristiques d'un composant, on analyse naturellement l'environnement local de ce composant avant d'effectuer le changement en question. Ce type de raisonnement permet de travailler efficacement au niveau local. Au niveau global, une analyse globale du problème et une connaissance experte de ce type de problème s'avère nécessaire.

Nous avons donc choisi un modèle totalement distribué pour :

1. tirer parti de la modularité naturelle des composants.

2. tirer parti de l'aspect souvent local des conflits.

3. profiter de la grande modularité implicite du problème pour satisfaire plus aisément les contraintes au niveau local des composants.

4. simuler de plus près et plus facilement l'approche naturel du raisonnement lors de modifications locales.

5. tenter d'obtenir une efficacité raisonnable au niveau de la résolution du fait de sa localité.

Le modèle choisi est donc distribué et chaque composant agit en tant qu'expert dans un univers multi-agents. Tous les composants disposent de la même connaissance, de la même expertise et se répartissent la tâche de résolution grâce à une coopération volontaire. Il tire parti de la modularité du modèle acteur qui s'adapte bien à la modularité des composants, et il localise la résolution là où se produisent les conflits. Il adopte aussi certaines propriétés du modèle *Abcl*, comme le synchronisme dans l'ordre d'émission et de réception de messages, et le fait que les objets disposent d'une mémoire locale. Du modèle du *contract net protocol*, il reprend les aspects de contractant et de manager.

Nous n'avons néanmoins résolu qu'une petite partie des problèmes, en nous attachant seulement à proposer un mécanisme de résolution, sans que celui-ci ne génère à proprement parler de plan d'action, et en nous limitant seulement aux aspects géométriques et topologiques directs du problème.

Le plan d'action est fabriqué en même temps qu'une partie du problème est résolue, et il est disséminé dans les résolutions partielles (satisfaction des contraintes) et successives des composants. Les actions et décisions situées au niveau local des composants vont en fait construire progressivement un plan d'action global fournissant une solution au problème.

Avant de présenter la partie concernant la résolution, il nous faut d'abord présenter le modèle algorithmique distribué permettant de gérer et de déterminer les agents pertinents.

III.3.3 Le concept de voisinage d'un composant

Nous introduisons ici la notion de *voisinage de composant*, son utilité pour notre problème, ce que doivent ou devraient être les propriétés de ce voisinage, et ses rapports avec la planification dans un univers multi-agent.

III.3.3.1 Définition informelle

Dans la figure III.8, nous voyons un groupe de composants disposés dans l'espace, groupe dans lequel il n'y a pas de conflit géométrique. Pour le composant C, nous appellerons voisinage de C l'ensemble des composants du groupe pour lesquels *la relation géométrique de voisinage* entre eux et le composant C est effective. Il reste à choisir une relation géométrique intéressante pour disposer d'un voisinage ayant de bonne propriétés.

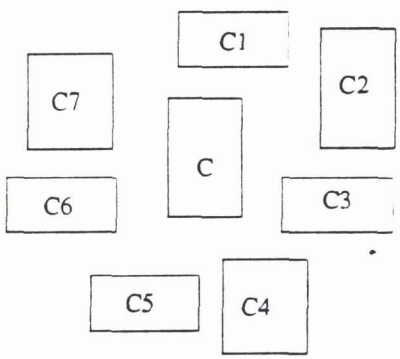


Fig III.8 notion de voisinage

Ainsi, si la relation géométrique est : "être le plus proche composant situé à l'Est, à l'Ouest, au Nord, ou au Sud ", le voisinage de C dans cette configuration topographique est l'ensemble des composants {C1, C3, C4, C6, C7}.

Le voisinage d'un composant C suivant une relation géométrique ρ est l'ensemble des composants de l'espace Ci tels que (C ρ Ci) est vraie.

On définit ainsi un ensemble de composants de l'espace liés au composant C. C'est en quelque sorte un contexte local, un monde "visible" pour C que l'on définit ainsi.

III.3.3.2 Danger d'un voisinage inadapté

Il est très important de disposer d'une bonne notion de contexte local pour un composant, en voici les raisons illustrées par la figure qui suit.

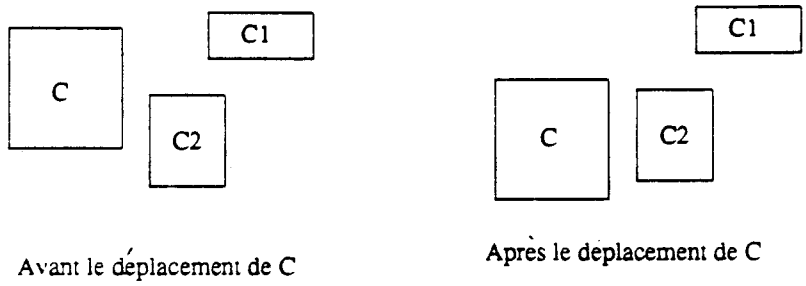


Fig III.9 déplacement et contexte local

Sur la figure III.9, le composant C se déplace vers le bas jusqu'à être situé en dessous de C1. Dans un univers multi-agents, lorsque C se déplace, il doit prévenir les autres agents de son déplacement.

Si nous reprenons le même voisinage que dans la section II.3.3.1, le composant C sera incapable de prévenir C1 de son changement de position. Le composant situé le plus près de C à l'Est est C2. Donc, sur ce schéma initial, le voisinage de C suivant cette relation de voisinage est {C2}. Par contre, le voisinage de C1 est {C}, puisque C est le composant le plus proche de C1 situé à l'ouest. Or, si l'on déplace C vers le bas jusqu'à une ordonnée inférieure à C1, comme c'est le cas dans l'exemple, C1 n'aura plus de voisins à l'ouest puisque son seul voisin C aura disparu. Mais C n'ayant pas enregistré C1 comme voisin ne préviendra pas ce dernier d'un changement de situation, et C1 ne sera donc jamais prévenu du fait que C s'est déplacé et ne fait plus partie de son voisinage ouest. Du coup, le voisinage enregistré pour C1 sera en désaccord avec la situation effective après déplacement de C.

On voit donc ici l'importance de disposer d'un voisinage cohérent et adapté à l'univers distribué dans lequel nous travaillons.

III.3.3.3 Rapport avec la planification en univers multi-agents

Nous avons vu que dans un système d'IA distribué, une part importante du travail de résolution est la communication entre experts. Grâce à cette communication, ils peuvent coopérer entre eux en se déléguant des tâches, en échangeant des informations de diverses natures, en contrôlant mutuellement leur part de résolution.

Dans la planification en univers multi-agents, ces capacités doivent être renforcées car les agents doivent déterminer seuls les agents avec lequels ils doivent collaborer, c'est à dire *déterminer leurs agents pertinents.*

Dans un univers comme celui sur lequel nous travaillons, la part de résolution directement liée à la notion de conflit géométrique nécessite également de modéliser les agents pertinents pour un agent quelconque. Ainsi un voisinage est en fait un ensemble d'agents pertinents de l'espace pour une relation géométrique particulière.

Mais tous les voisinages induits par différentes relations géométriques ne sont pas tous équivalents, et peuvent même être complètement inadaptés àla résolution de conflits géométriques. Il faut donc déterminer ce que doivent être les bonnes propriétés géométriques d'un voisinage pour le cas de la résolution de conflits géométriques. On pourra inversement remarquer qu'un bon voisinage pour la résolution de conflits géométriques n'est pas forcément adapté á la résolution de conflits dûs aux contraintes fonctionnelles.

III.3.3.4 Propriété fondamentale du voisinage : la symétrie

Le voisinage définissant l'ensemble des agents pertinents au point de vue topologique, nous pouvons remarquer que si un composant C1 est pertinent pour un composant C2 pour résoudre un conflit, alors il est normal que C2 soit pertinent pour C1 pour résoudre ce même conflit. En effet, C1 et C2 doivent tous deux participer à la résolution du conflit qui les concerne tous les deux. Cette symétrie du caractère de pertinence vient du problème topologique à résoudre, comme nous l'avons précisé au paragraphe III.3.3.2.

Remarquons de plus que la symétrie de la relation de voisinage permet de distribuer la totalité des relations topologiques entre composants. *C'est grâce à la symétrie que le contexte local d'un composant peut être cohérent avec le projet global.*

La symétrie permet de plus de remettre à jour facilement les voisinages des composants, de calculer simplement les voisinages pour un ensemble de composants fixés au départ, elle permet enfin un changement dynamique des interdépendances topologiques entre les composants.

Mais c'est surtout de part la possibilité de répartir de manière distribuée les relations topologiques entre composants d'une façon cohérente avec le projet global, que la symétrie est la propriété essentielle que doit respecter un voisinage.

III.3.3.5 Unicité des voisins

Supposons que nous ayons défini un voisinage tel que celui schématisé sur la figure III.10, c'est à dire un voisinage où il y aurait quatre directions privilégiées pour déterminer des voisins (l'horizontale, la première bissectrice, la verticale, la deuxième bissectrice) et huit sens de scrutation, chaque sens correspondant au *"regard"* du composant vers son monde environnant. Alors le composant C1 est voisin de C non seulement pour la direction verticale, mais également pour la direction de la première bissectrice. De même C2 est non seulement voisin de C pour la direction horizontale, mais également pour la direction de la première bissectrice.

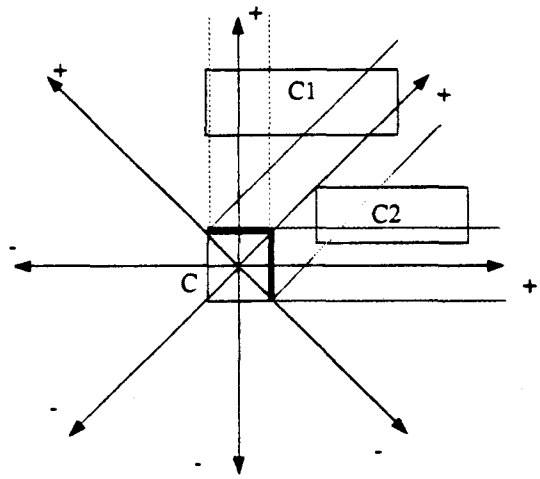


Fig III.10 une relation de voisinage ne respectant pas la propriété d'unicité

Lorsqu'une modification sera apportée sur C, celle-ci devra être répercutée pour les deux occurences de voisin C1 du composant C. D'autre part, lorsque C va déléguer une tâche de résolution à C1, il devra le faire pour les deux occurences de voisin C1 ou faire un choix entre ces deux occurences pour la délégation. Ceci est important car l'occurence de voisin C1 suivant la verticale ne se comportera sans doute pas de la même manière que l'autre occurence suivant la première bissectrice. Si il y a un processus de recalcul ou de délegation en chaîne, la complexité de l'arbre de recherche sera multipliée d'autant.

On risque de provoquer ainsi une explosion combinatoire qui ralentira terriblement les mécanismes de résolution et qui fera perdre tout son intérêt à une telle interface. De plus, le type de voisinage va influencer la résolution qui ne sera plus de qualité équivalente pour toutes les directions. On risque aussi d'avoir des choix menant à une solution correcte et d'autres menant à une impasse où une solution de mauvaise qualité pour le même problème topologique à résoudre.

La propriété d'unicité est donc indispensable pour garantir à la fois une solution dont la qualité n'est pas variable, ainsi que pour conserver une efficacité minimale au processus de

résolution en minimisant les communications et les délégations de tâches au sein du réseau d'agents. Elle simplifie de plus grandement les algorithmes de contrôle ou de résolution.

III.3.3.6 Propriétés intéressantes pour un voisinage

Isotropie de la relation de voisinage

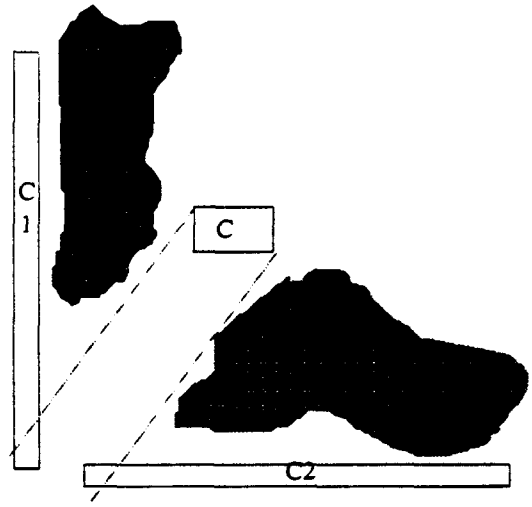
Définition : une relation de voisinage sera qualifiée d' *isotropique* si elle conserve ses propriétés pour toutes les directions de l'espace.

Nous avons défini au paragraphe III.3.3.1 un voisinage qui était tel que la relation géométrique considérée prenait en compte les quatre points cardinaux. Ainsi C3 était un voisin Est, et C4 un voisin Sud. Le fait de modifier la configuration locale de ces composants va impliquer de redéterminer l'ensemble des agents pertinents pour chaque composant du groupe.

Si la relation géométrique utilisée pour définir le voisinage ne fait aucune distinction de direction, redéterminer l'ensemble des agents pertinents ne sera pas fait à l'aide d'un algorithme spécialisé adapté à la direction ou au voisinage choisi, mais basé uniquement sur la relation géométrique sans avoir à tenir compte des directions. On utilisera un algorithme unique pour toutes les directions. Dans le cas inverse, toutes les opérations qui impliquent une communication d'un agent avec un autre nécessiteront une prise en compte de la direction, soit dans l'algorithme lui-même, soit en traitant chaque direction de manière équivalente, mais en mémorisant les voisins d'une manière qui tienne compte de cette direction.

Ainsi une relation de voisinage isotropique minimisera les traitements sur les voisins par uniformisation de ceux-ci, ainsi que par une prise compte équivalente de tous les voisins. Le choix de la relation géométrique utilisée pour la détermination du voisinage va donc influer directement sur le coût des communications entre agents et sur l'efficacité globale du système. On a donc tout intérêt à choisir une relation géométrique isotropique.

L'exemple suivant montre l'interêt de l'isotropie pour un déplacement de composant.



On voit sur ce schéma que si la relation géométrique est isotrope, on pourra envisager pour C un déplacement oblique figuré par les traits hachurés, jusqu'à rencontrer C1 ou C2, sans recourir à des méthodes sophistiquées de recalcul de voisinage, ni passer par des méthodes itératives de simulation de déplacement oblique par déplacements horizontaux et verticaux. En fait, l'isotropie permet de gérer des déplacements de composant plus efficaces sans provoquer de conflit géométrique, et ceci dans toutes les directions.

A l'inverse, si seulement quatre directions comme dans l'exemple précédent peuvent être gérées par le voisinage, le déplacement figuré sur ce schéma ne pourra être obtenu que par une succession de petits déplacments verticaux ou horizontaux. On ne pourra donc pas gérer interactivement un tel déplacement de composant..

Voisinage et espace topologique

Une relation très intéressante pour le voisinage est la qualité topologique de la relation. En effet, si la relation géométrque induit une topologie sur l'espace dans lequel nous travaillons, nous disposerons de toute la technique mathématique pour fabriquer les algorithmes de détermination et modification du contexte local des composants, grâce aux propriétés topologiques de cet espace.

En conclusion de cette partie, nous pouvons dire qu'un voisinage idéal devrait utiliser une relation géométrique isotrope et symétrique, et que chaque voisin trouvé devrait être unique (ne pas être comptabiliser deux fois comme voisin. Ceci induirait une topologie complète dans l'espace, c'est-à-dire un ensemble de propriétés bien connues et dont on pourrait tirer parti.

Nous présentons ici le voisinage que nous avons choisi pour résoudre le problème de la détermination des agents pertinents, introduisons le formalisme qui sera utilisé pour toute la suite du document concernant les concepts logiques et mathématiques utilisés, et décrivons les propriétés de ce voisinage.

III.3.4 Définition informelle du voisinage choisi

A l'aide du schéma suivant, nous donnons une définition informelle de la notion de voisin choisie.

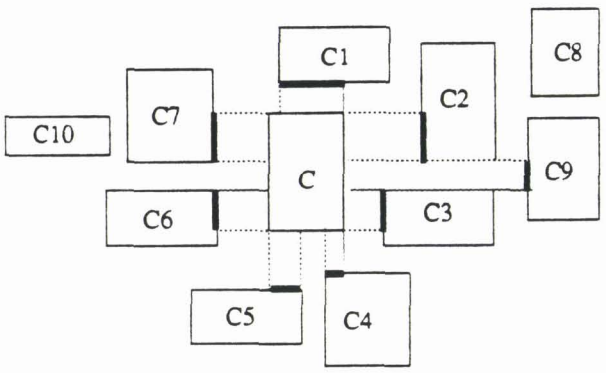


Fig III.11

Sur cette figure, le composant C a comme voisins : C2, C9, C3, C6, C7, C1, C4, C5. Le voisinage est distribué suivant quatre orientations correspondant aux quatre points cardinaux.

Définition : Un composant C' est voisin d'un composant C dans une orientation Δ parmi Nord, Sud, Ouest, Est s'il existe une portion de C qui peut être reliée à C' par un segment porté par l'orientation Δ sans traverser aucun autre composant existant dans l'espace.

Ainsi, C2 et C3 sont des voisins à l'Est de C car on peut tracer un segment horizontal allant de C à C2 ou C3 sans rencontrer d'obstacle, c'est à dire d'autre composant. La portion de C2 et de C3 joignable a été matérialisée par un bord gras sur la figure. De même C9 est un voisin Est de C car on peut tracer un segment horizontal le reliant à C sans rencontrer d'obstacle. Sur la figure, nous avons matérialisé en gras (sur les voisins) les bords que l'on peut joindre à C.

La relation géométrique utilisée peut être définie comme : " *le plus proche composant visible de C suivant l'orientation choisie*". On voit qu'il y aura dans ce cas quatre type de voisinage : à l'Est, à l'Ouest, au Nord, au Sud.

En fait, la relation choisie joue sur le caractère symétrique nécessaire de la relation de voisinage, et conserve en même temps l'aspect de proximité ou d'éloignement des composants entre eux, par la longueur du segment reliant deux composants voisins. Ce sont bien évidemment en priorité sur ses voisins les plus proche qu'un composant effectuera un raisonnement , une délégation, ou encore négociera une action ou un déplacement. Le fait de disposer de voisins plus "lointains" est un effet de la symétrie de la relation et permet de plus une annalyse plus riche de la situation.

III.3.5 Composant visible et composant masqué

Dans cette définition du voisinage choisi, le paradigme important est la notion de visibilité d'un composant pour un autre. Nous avons voulu organiser notre société d'experts pour qu'ils se "voient" et se "parlent" comme le feraient des humains. La restriction de cette visibilité étant située au niveau des quatre directions précitées. L'apport principal de cette notion de visibilité pour notre modèle est en fait d'impliquer automatiquement la symétrie de la relation de voisinage

Définition informelle : Un composant R est visible pour un autre composant C, s'il existe au moins un segment de droite horizontal ou vertical joignant des bords opposés de R et C.

Un composant peut être invisible à un autre composant pour deux raisons possibles : soit il est géométriquement impossible de tracer un segment horizontal ou vertical joignant les bords opposés des deux composants, soit il existe un ou plusieurs autre composant faisant obstacle au tracé du segment. Dans ce cas les deux composants sont invisibles l'un à l'autre parce qu'ils sont masqués par un autre composant.

Les composants masqués jouent un rôle important lors d'une modification du schéma; en effet un composant initialement masqué peut devenir visible à un autre composant par une action sur le(s) composant(s) masquant situé(s) entre eux. Le modèle doit être capable de prendre en compte les conséquences de tels événements indirects.

La figure suivante montre plus clairement cette notion de visibilité à l'aide d'un exemple spécifiique.

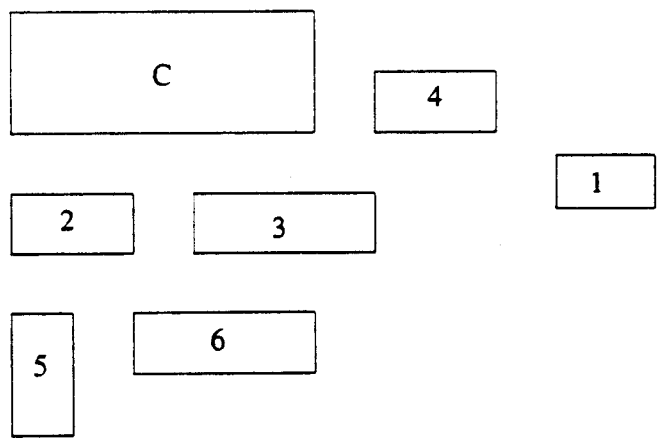


Fig III.12 Visibilité d'un composant. Dans ce cas particulier, les composants 2, 3, 6, et 4 sont visibles pour C, par contre le composant 5 est masqué à C par le composant 2, et le composant 1 est invisible pour C.

Sur cette figure, on voit que 5 est invisible à C et réciproquement, mais que si 2 se déplace par exemple, 5 et C pourront redevenir visible l'un à l'autre. Par contre 1 ne sera

jamais visible à C, sauf si 1 ou C se déplace verticalement pour arriver dans une position où ils devienent visibles l'un à l'autre.

Remarquons que les composants pertinents pour un composant donné sont ceux qu'il peut "voir".

III.3.6 Description détaillée du voisinage choisi

Nous décrivons ici le formalisme choisi pour décrire les objets mathématiques que nous utiliserons. Mais nous devons tout d'abord signaler que nous travaillons dans le plan cartésien, avec un repère orthonormé direct (O, x, y), au sens mathématique habituel.

III.3.6.1 Description géométrique des composants

Un composant, ou encore agent si nous nous référons à la terminologie habituelle en IA distribuée, est un rectangle dont les quatre coins sont identifiés par : **bg** pour "bas gauche", **hd** pour "haut droit", **hg** pour "haut gauche", **bd** pour bas droit. La figure III.13 résume cette définition sur un schéma.

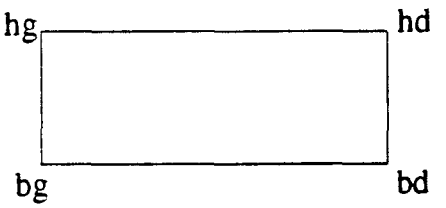


Fig III.13 définition d'un composant dans cette application

Chaque coin du composant est donc un point de l'espace repérable par ses coordonnées cartésiennes.

III.3.6.2 Direction et sens de déplacement

Les composants sont supposés être uniquement déplacés suivant une direction horizontale ou verticale. Comme il n'y a que deux directions, chacune est symbolisée par sa lettre initiale en majuscule. Ainsi, la direction horizontale sera notée H, et la direction verticale sera notée V.

Les deux directions prises en compte sont considérées comme opposées. La direction opposée d'une direction D est notée $\bar{D}$. Toute direction D appartient au couple { H, V}. Si une direction D vaut H alors $\bar{D}$ vaudra V, et inversement.

Pour chaque direction, on dispose de deux sens les orientant :

- le sens +, qui suit le sens des vecteurs du repère orthonormé direct (↑, →).
- le sens −, qui suit le sens opposé des vecteurs du repère orthonormé direct (↓, →).

Tout sens S appartient à { −, +). Le sens opposé à un sens S est noté $\bar{S}$. Si S=+, alors $\bar{S}$=-, et inversement.

Orientation DS : C'est la donnée d'un couple (D, S).

beaucoup de désignations d'entités concernant les composants font référence à un couple (D, S). Ainsi l'Est peut être matérialisé par le couple (H, +). Les couples (D, S) seront souvent notés en indice de l'entité concernée. On notera l'orientation DS par "(D, S)" ou "DS" indifféremment.

Exemple : H+, (V, —).

III.3.6.3 Coordonnées de points et composants

Nous définissons ici deux notions de coordonnées, relatives à un point et à un composant, qui seront utilisées par la suite dans notre formalisme. Ces notions ne seront utiles que pour une partie restreinte de l'exposé, essentiellement pour fournir un formalisme de base pour des notions plus abstraites.

Coordonnée D d'un point P

C'est la coordonnée du point P, relativement à la direction D. Elle sera notée $O_D(P)$.

$$O_V(P) = P.y$$

$$O_H(P) = P.x$$

Coordonnée DS d'un composant C

C'est la valeur représentant l'extrémité du composant C, relativement à la direction D et au sens S. Elle sera notée $O_{DS}(C)$. Par exemple, la coordonnée H+ d'un composant correspond à la valeur en x du bord droit du composant. On a exactement :

$$O_{H-}(C) = bg.x = hg.x$$

$$O_{H+}(C) = bd.x = hd.x$$

$$O_{V-}(C) = bg.y = bd.y$$

$$O_{V+}(C) = hg.y = hd.y$$

III.3.6.4 Bord et côté de composant

Nous définissons ici formellement ce qu'est un bord et un côté de composant, Ces deux notions sont très importantes puisque ce sont elles qui vont permettre de pouvoir situer un composant par rapport au monde extérieur.

Bord D d'un composant

Le bord D d'un composant C est l'intervalle fermé $[O_{D-}(C), O_{D+}(C)]$. Il sera noté $b_D(C)$.Par exemple, le bord H d'un composant C est l'intervalle $[O_{H-}(C), O_{H+}(C)]$ = [bg.x, bd.x]. la figure suivante résume simplement cela. Cette définition correspond bien à l'idée intuitive du bord horizontal ou vertical d'un composant.

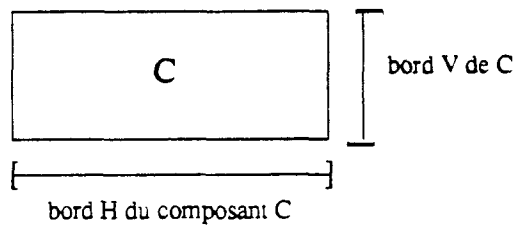


Fig III.14 les deux bords H et V d'un composant

Remarque : Le bord D d'un composant n'est pas unique. Tout composant a deux bords H ([bg.x, bd.x] ou [hg.x, hd.x] et deux bords V ([bg.y, bd.y] ou [hg.y, hd.y]).

Côté DS d'un composant

Le côté DS d'un composant C est la partie du plan non recouverte par C, relativement à la direction D et au sens S. Il sera noté $c_{DS}(C)$. Nous pouvons exprimer cela plus formellement par la relation suivante :

$$c_{DS}(C) = \left\{ p \in \mathcal{P} / \begin{array}{l} O_{DS}(C) < O_D(p) \text{ si } S = + \\ \text{ou} \\ O_{DS}(C) > O_D(p) \text{ si } S = - \end{array} \right\}$$

Où *P* représente le plans cartésien à deux dimensions.

exemple : $c_{H+}(C) = \left\{ p \in \mathcal{P} / \; O_{H+}(C) < O_H(p) \right\} = \{ p \in \mathcal{P} / \; hd.x < p.x \}$. C'est donc le demi-plan de l'espace situé à l'Est de C. La figure suivante montre cela clairement.

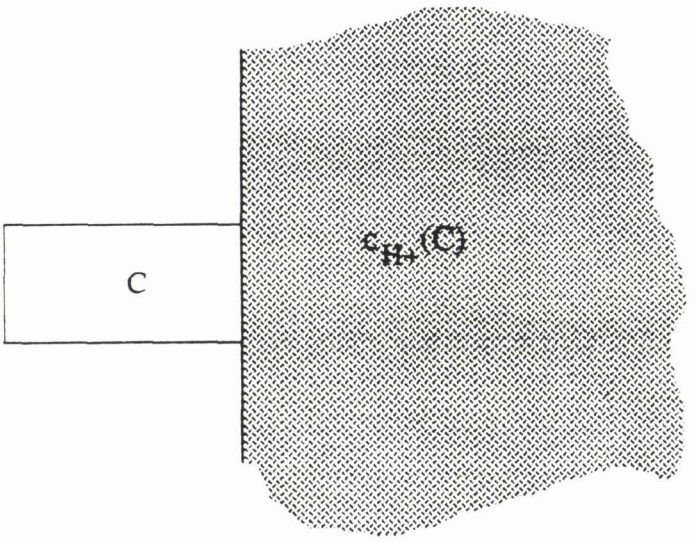


Fig III.15 Le côté H+ d'un composant quelconque

cette notion de côté DS d'un composant permet un découpage de l'espace en quatre demi-plans, et assurera l'unicité de la propriété d'être voisin d'un composant. Un composant C1 ne pourra être voisin d'un autre composant que pour une seule orientation (D,S).

Bord DS d'un composant

Définition : Le bord DS d'un composant C est le bord D de ce composant qui se trouve sur la frontière ouverte du demi-plan $c_{DS}(C)$.

Exemple : Sur la figure III.15, le bord H+ de C est celui qui se trouve sur la frontière ouverte de $c_{H+}(C)$.

Remarque : le bord DS d'un composant est unique, contrairement au bord D.

III.3.6.5 Intersections de composant

Nous définissons ici trois notions qui concernent le recouvrement de bords de composants, pour une direction donnée.

Intersection D de deux composants

C'est l'intersection des bords $\bar{D}$ des deux composants. L'intersection H concerne donc les bords V des deux composants. Elle sera notée $i_D(C1, C2)$, C1 et C2 étant les deux composants en question. Ceci peut être résumé simplement par la relation :

$$i_D(C1, C2) = b_{\bar{D}}(C1) \cap b_{\bar{D}}(C2).$$

La figure suivante montre simplement à quoi correspond cette notion. Il s'agit ici de définir clairement le recouvrement horizontal ou vertical entre deux composants. Si les deux composants ne se recouvrent pas, au moins l'une des deux intersections $i_H(C1, C2)$ ou $i_V(C1, C2)$ sera vide.

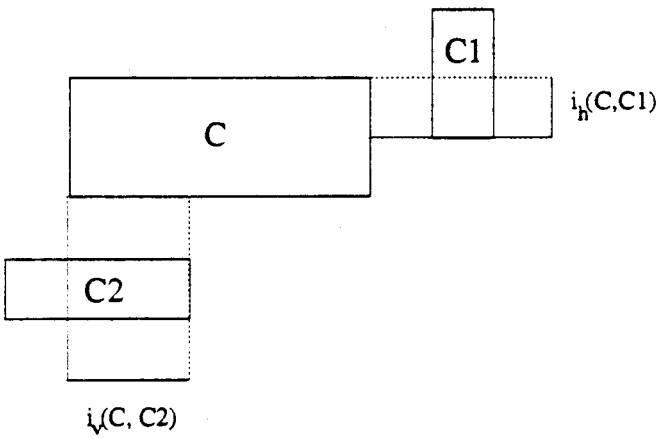


Fig III.16 intersections verticale et horizontale entre deux composants

L'ensemble Intersection D d'un composant C

L'Intersection D d'un composant C est l'ensemble des composants R tels que leur intersection D avec C est non vide. Cet ensemble sera noté $I_D(C)$. Ceci peut être résumé simplement par la définition mathématique suivante :

$$I_D(C) = \{R / R \neq C \wedge i_D(C, R) \neq \emptyset\}$$

Exemple : l'intersection horizontale d'un composant est l'ensemble des composants tels que les intervalles représentés par leurs bords verticaux respectifs se recouvrent.

Remarque : les deux ensembles $I_H(C)$ et $I_V(C)$ sont disjoints. En effet, si aucun des composants de l'espace n'intersecte C, on ne peut pas avoir à la fois $I_D(C, R) \neg=\emptyset$ et $I_{\bar{D}}(C, R) \neg=\emptyset$.

L'ensemble intersection D,S d'un composant C

Par rapport à la définition précédente, on rajoute une contrainte supplémentaire pour qu'un composant puisse appartenir à ce nouveau type d'ensemble : les composants appartenant à cet ensemble doivent tous se trouver du même côté du composant C.

Définition : L'intersection D,S d'un composant C est l'ensemble des composants R qui appartiennent à $I_D(C)$ et qui se trouvent dans le demi-plan défini par le coté DS de C, soit $C_{DS}(C)$.

On peut résumer cela par la relation suivante :

$$I_{DS}(C) = \left\{ R \in I_D(C) / \begin{array}{l} O_{D\bar{S}}(R) < O_{DS}(C) \text{ si } S = + \\ \text{ou} \\ O_{D\bar{S}}(R) > O_{DS}(C) \text{ si } S = - \end{array} \right\}$$

Par exemple, $I_{H+}(C) = \left\{ R \in I_H(C) / O_{H-}(R) > O_{H+}(C) \right\}$

Remarque : Pour un composant C, un composant quelconque R ne peut appartenir au plus qu'à un seul des quatres ensembles $I_{H+}(C)$, $I_{H-}(C)$, $I_{V+}(C)$, $I_{V-}(C)$. En effet, si un composant R appartient à $I_{DS}(C)$, il est forcément situé dans le demi-plan $C_{DS}(C)$. On a donc $I_D(C, R) \neg= \emptyset$. Pour que R appartienne à $I_{\bar{D}S}(C)$, il faut que $I_{\bar{D}}(C, R) \neg= \emptyset$ également. Ce qui est impossible car il faudrait que R et C s'intersectent.

De même, si R est situé dans le demi-plan $C_{DS}(C)$, il ne peut pas se trouver dans l'autre demi-plan $C_{D\bar{S}}(C)$, pusique $C_{DS}(C)$ et $C_{D\bar{S}}(C)$ sont disjoints. Donc R ne peut pas appartenir à la fois à $I_{DS}(C)$ et à $I_{D\bar{S}}(C)$.

Donc R ne peut appartenir qu'à un seul des quatre ensembles cités précedemment.

La notion d'intersection D,S d'un composant C est très importante. Grace à elle, on filtrera très vite l'ensemble des composants qui sont candidats pour le voisinage de C. En effet, pour chaque orientation cardinale Est, Ouest, Nord et Sud, il existe un couple (direction, sens) qui symbolise cette orientation. Pour obtenir les voisins à l'Est de C, il suffira d'abord de déterminer $I_{H+}(C)$, puis de déterminer dans cet ensemble les composants qui seront effectivement voisins de C. La définiton de l'intersection D,S permet en fait d'implanter des algorithmes efficaces de détermination et de modification de voisins.

III.3.6.6 Distance D,S entre deux composants

La distance D,S entre deux composants va nous permettre d'ordonner l'ensemble des voisins pour chaque composant. La définition informelle du voisinage que nous avons donnée au paragraphe III.3.4 implique que les voisins d'un composant peuvent se trouver à une distance variable de celui-ci. Or, pour résoudre des problèmes de conflit géométrique inter-composants, tous les agents pertinents d'un composant n'ont pas la même importance du point de vue géométrique: l'influence d'un agent pertinent décroit en sens inverse de sa distance par rapport à l'agent dont on considère le voisinage. La distance que nous définissons ici va nous permettre d'ordonner l'ensemble des voisins selon cette idée intuitive d'influence en relation avec la distance.

Définition : Etant donnés deux composants A et B, la distance orientée D,S entre A et B, notée $d_{DS}(A,B)$, est $d_{DS}(A,B) = | O_{D\hat{S}}(B) - O_{D\hat{S}}(A) |$.

Exemple :
A est tel que bg.x = 1, bd.x = 5, hg.y = 4, bg.y = 1.
B est tel que bg.x = 6, bd.x = 8, hg.y = 8, bg.y = 6..
$d_{H+}(A,B) = | O_{H-}(B) - O_{H-}(A) | = | bg.x(B) - bg.x(A) | = | 6 - 1 | = 5.$

d_{DS} est une distance

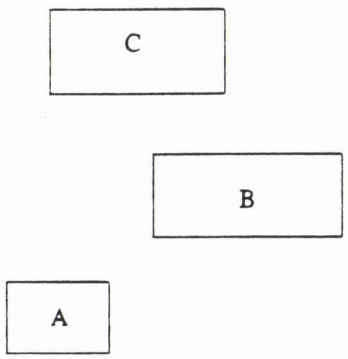
- $d_{DS}(A,B)$ est supérieur ou égal à 0, pour tous composants A et B, puisque c'est la valeur absolue d'une différence.

- $d_{DS}(A,A) = | O_{D\hat{S}}(A) - O_{D\hat{S}}(A) | = 0$.

- $d_{DS}(A,B) = | O_{D\hat{S}}(B) - O_{D\hat{S}}(A) | = | O_{D\hat{S}}(A) - O_{D\hat{S}}(B) | = d_{DS}(B,A)$. Par définition de la valeur absolue.

- soient A,B,C trois composants, l'inégalité $d_{DS}(A,C) \leq d_{DS}(A,B) + d_{DS}(B,C)$ est elle toujours vérifiée ?

Cas 1 : les deux composants B et C se trouvent d'un même coté $_{DS}$ de A, comme c'est le cas par exemple sur la figure suivante.



Alors $d_{DS}(A,B) + d_{DS}(B,C) = | O_{D\hat{S}}(B) - O_{D\hat{S}}(A) | + | O_{D\hat{S}}(C) - O_{D\hat{S}}(B) |$.

Comme B et C sont du même coté $_{DS}$ de A, on a :

soit $O_{D\hat{S}}(C) > O_{D\hat{S}}(B) > O_{D\hat{S}}(A)$

$| O_{D\hat{S}}(B) - O_{D\hat{S}}(A) | = O_{D\hat{S}}(B) - O_{D\hat{S}}(A)$ **et**

$| O_{D\hat{S}}(C) - O_{D\hat{S}}(B) | = O_{D\hat{S}}(C) - O_{D\hat{S}}(B)$

D'où

$d_{DS}(A,B) + d_{DS}(B,C) = O_{D\hat{S}}(B) - O_{D\hat{S}}(A) + O_{D\hat{S}}(C) - O_{D\hat{S}}(B) =$

$O_{D\hat{S}}(C) - O_{D\hat{S}}(A) = | O_{D\hat{S}}(C) - O_{D\hat{S}}(A) | = d_{DS}(A,C)$.

l'inégalité est donc bien vérifiée dans ce cas.

ou bien $O_{D\hat{S}}(A) > O_{D\hat{S}}(B) > O_{D\hat{S}}(C)$

$| O_{D\hat{S}}(B) - O_{D\hat{S}}(A) | = O_{D\hat{S}}(A) - O_{D\hat{S}}(B)$ **et**

$| O_{D\hat{S}}(C) - O_{D\hat{S}}(B) | = O_{D\hat{S}}(B) - O_{D\hat{S}}(C)$

d'où

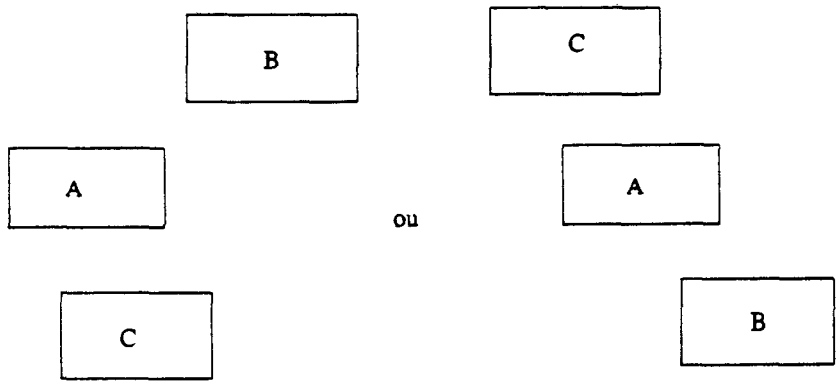
$d_{DS}(A,B) + d_{DS}(B,C) = O_{D\hat{S}}(A) - O_{D\hat{S}}(B) + O_{D\hat{S}}(B) - O_{D\hat{S}}(C) =$

$O_{D\hat{S}}(A) - O_{D\hat{S}}(C) = | O_{D\hat{S}}(C) - O_{D\hat{S}}(A) | = d_{DS}(C,A) = d_{DS}(A, C)$.

l'inégalité est donc également vérifiée dans ce cas.

Si B et C sont situés du même côté de A, l'inégalité est vérifiée.

Cas 2 : les deux composants B et C se trouvent de part et d'autre du composant A, comme c'est le cas sur la figure suivante :



Alors $d_{DS}(B,C) > d_{DS}(C,A)$ *et* $d_{DS}(B,C) > d_{DS}(B,A)$.

Donc

$$d_{DS}(A,B) + d_{DS}(B,C) > d_{DS}(A,B) + d_{DS}(A,C) > d_{DS}(A,C)$$

l'inégalité est donc également vérifiée si B et C sont situés de part et d'autre de A.

En conclusion, d_{DS} est donc bien une distance entre composants, suivant une orientation (D, S).

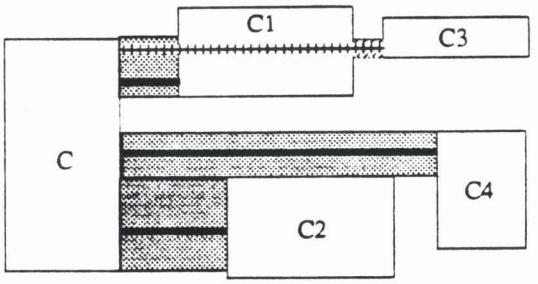
Cette distance va donc induire une topologie pour chaque orientation DS d'un composant. Pour chaque composant, on aura donc un cylindre topologique associé à chacune des orientations Est, Ouest, Nord, Sud. Mais chacun de ces espaces topologiques sera indépendant des autres. D'où la nécessité d'élaborer des algorithmes complexes de modifications de voisinages.

III.3.6.7 Visibilité DS d'un composant

Nous décrivons ici formellement ce que nous avons appelé précédemment la visibilité d'un composant.

Définition : Un composant R est visible à C pour une orientation DS si et seulement si on peut tracer un segment reliant le bord DS de C au bord D$\bar{S}$ de R, parallèle à la direction D et sans traverser aucun autre composant de l'espace.

exemple :



Sur cette figure, les composants visibles à C dans l'orientation (H, +), c'est à dire à l'Est de C sont : C1, C2, C4, car on peut tracer un segment horizontal du bord H+ de C à chacun des bords H— de C1, C2, C4 sans traverser aucun autre composant. Par contre, tout segment horizontal reliant le bord H+ de C au bord H- de C3 traverse obligatoirement un bord H de C1. Donc C3 n'est pas visible à C pour l'orientation H+.

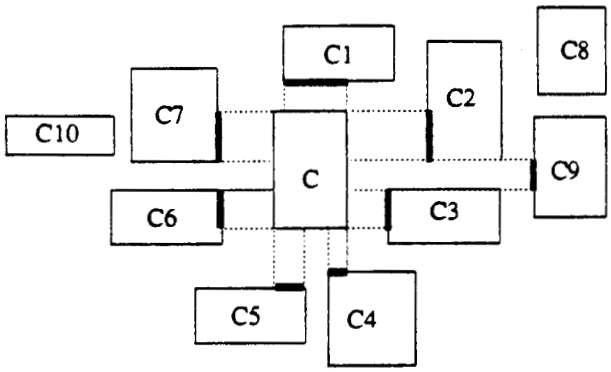
Remarque : un composant R peut être invisible à C pour l'orientation DS pour deux raisons : soit il n'appartient pas à $I_{DS}(C)$, soit il est *masqué* par un ou plusieurs autres composants, c'est-à-dire que ce ou ces derniers composants sont ceux qui sont traversés par tout segment parallèle à la direction D qui relie le bord DS de C au bord $D\bar{S}$ de R.

III.3.6.8 Voisinage d'un composant

Nous donnons ici une définition plus formelle du voisinage d'un composant et introduisons le mécanisme de construction récurrente du voisinage.

Définition : Etant donné un composant C, un composant R est voisin de C pour une direction D et un sens S, si et seulement si R appartient à $I_{DS}(C)$ et R est visible à C pour l'orientation DS.

Exemple : Sur la figure suivante, C2 et C9 sont voisins de C pour la direction *H* et le sens +, car ils appartiennent tous deux à $I_{H+}(C)$ et sont tous deux visibles à C pour l'orientation (H, +). Par contre, C10 appartient bien à $I_{H-}(C)$ mais n'est pas visible à C pour l'orientation (H, —), car il est masqué par C7. Donc C10 n'est pas un voisin de C pour l'orientation (H, —), ni pour les autres orientations d'ailleurs, puisqu'il est situé du côté $C_{H-}(C)$ et ne peut donc appartenir aux autres intersections $I_{DS}(C)$. On a la situation symétrique pour le composant C8



Remarque : Cette définition est redondante avec la notion de visibilité DS d'un composant. En effet, tout composant visible à un composant C, d'après la définition que nous avons donnée au paragraphe précédent, est un voisin de C.Néanmoins la formulation que nous donnons ici va nous permettre de construire plus facilement les algorithmes de création et de manipulation de voisinage, à partir de l'intersection DS du composant C et de la distance entre deux composants.

Définition : Le voisinage DS d'un composant C est l'ensemble des composants R qui sont voisins de C pour la direction D et le sens S. Ce voisinage est noté $V_{DS}(C)$.

Définition : Le voisinage d'un composant C est l'union des quatres voisinages $V_{DS}(C)$ obtenus en faisant varier la direction D et le sens S.

On a donc $V(C) = V_{H+}(C) \cup V_{H-}(C) \cup V_{V+}(C) \cup V_{V-}(C)$.

Sur la figure précédente, on a :

$V_{H+}(C) = \{ C2, C3, C9 \}$.

$V_{H-}(C) = \{ C6, C7 \}$

$V_{V+}(C) = \{ C1 \}$

$V_{V-}(C) = \{ C4, C5 \}$

$V(C) = \{ \{ C2, C3, C9 \}, \{ C6, C7 \}, \{ C1 \}, \{ C4, C5 \} \}$

Symétrie de la relation de voisinage

Supposons que R soit un voisin DS de C. Alors on peut tracer au moins un segment parallèle à la direction D du bord DS de C au bord DS̄ de R. Mais on peut également tracer ce même segment du bord DS̄ de R au bord DS de C. Donc C est visible à R pour l'orientation DS̄. Donc C est un voisin DS̄ de R.

Ceci signifie que la relation de voisinage choisie ici est symétrique, au sens près choisi sur l'orientation DS. Si R appartient à $V_{DS}(C)$, alors C appartient à $V_{DS}(R)$.

Construction récurrente du voisinage DS

Supposons que $I_{DS}(C)$ soit ordonné par ordre croissant des distances $d_{DS}(R, C)$, R étant un élément de $I_{DS}(C)$. Nous indiçons $I_{DS}(C)$ suivant le numéro d'ordre de chacun des composants R.

On a $[\ I_{DS}(C)]_{\ 1}$ qui est forcément visible à C pour l'orientation DS puisque comme c'est le composant le plus proche de C, aucun autre composant ne peut le masquer. C'est donc un voisin DS de C, nous le noterons v1.

Supposons que jusqu'à l'indice n (n > 1), la propriété suivante soit vérifiée :

$$\forall n > 1,\ I_D\left(v_n, C\right) \not\subset \bigcup_{1 \le k < n} \left(I_D\left(v_k, C\right)\right) \qquad \textbf{(E1)}$$

v indiquant que le composant considéré est un voisin DS de C.

Il existe un indice l > 1 tel que $v_n = [\ I_{DS}(C)]_{\ l}$. Considérons maintenant le composant $r_{l+1} = [\ I_{DS}(C)]_{\ l+1}$.

Alors soit $I_D(r_{l+1}, C) \not\subset \left(\bigcup_{1 \le k < n} (I_D(v_k, C))\right) \cup I_D(v_n, C)$, et il existe donc une petite partie de r_{l+1} qui n'est masquée par aucun composant de $I_{DS}(C)$ d'indice inférieur à l+1. Dans ce cas r_{l+1} est visible à C pour l'orientation DS, et c'est donc un voisin DS de C, qui aura comme indice n+1.

Ou sinon $I_D(r_{l+1}, C) \subset \left(\bigcup_{1 \le k < n} (I_D(v_k, C))\right) \cup I_D(v_n, C)$, et donc r_{l+1} est complètement masqué à C par les voisins de C d'indice inférieur ou égal à n. Donc r_{l+1} n'est pas visible à C est n'est donc pas voisin DS de C. On peut alors réitérer ce raisonnement pour les successeurs de r_{l+1}, jusqu'à épuisement de l'ensemble $I_{DS}(C)$, auquel cas on ne trouvera pas de voisins DS de C pour les indices supérieurs à l+1; ou bien il existera un indice m tel que r_m ne vérifie pas la dernière inclusion. Dans ce cas $r_m$sera le voisin v_{n+1} de C pour l'orientation DS et on aura bien $I_D(r_m, C) \not\subset \bigcup_{1 \le k < (n+1)} (I_D(v_k, C))$, et on pourra á nouveau itérer le processus sur le reste des composants r_i jusqu'à épuisement de $I_D(C)$.

On peut donc construire de cette manière l'ensemble $V_{DS}(C)$ par récurrence sur l'ensemble $I_{DS}(C)$. Cette construction sera reprise plus en détail lors de la description de l'algorithme de création de voisinage pour un composant.

Remarque : Cette construction par récurrence n'est en fait qu'un moyen de déterminer la visibilité d'un composant d'une manière incrémentale. Elle implique néanmoins la gestion

d'unions d'intersections d'intervalles de R. Ceci nous amènera à construire un objet spécialisé pour ce genre de manipulation.

III.3.6.9 Propriétés et restrictions de ce voisinage.

Si nous évaluons maintenant le voisinage que nous avons défini ici, nous constatons que :

1. La relation géométrique de voisinage n'est pas isotrope. Ceci est évident du fait que le voisinage est constitué d'une union de quatre sous-voisinages spécialisés et indépendants entr'eux, au moins pour les directions opposées. Nous n'avons en fait pas trouvé de relation géométrique permettant de construire une topologie complète sur l'ensemble des composants de l'espace. Néanmoins, pour chacune des quatres orientations considérées et manipulables dans ce modèle, nous disposons d'un espace topologique pour le voisinage suivant cette orientation. Ceci nous permet d'ordonner et de construire simplement chaque voisinage orienté d'un composant.

2. La relation est *presque* symétrique. Nous avons vu que si R appartient à $V_{DS}(C)$, alors C appartient à $V_{D\hat{S}}(R)$. Ceci va simplifier grandement les algorithmes de modification de voisinage puisqu'à chaque fois qu'une modification sera enregistrée sur un voisinage $V_{DS}(C)$, elle pourra être immédiatement répercutée sur le voisinage $V_{D\hat{S}}(R)$, R désignant un composant concerné par la modification de $V_{DS}(C)$. A titre d'exemple, si un composant devient un nouveau voisin DS de C, alors C est un nouveau voisin $D\hat{S}$ de R. Cependant, on ne pourra rien déduire concernant des directions opposées, et quoique généralisés aux quatres orientations DS, les algorithmes de modifications de voisinage traiteront chaque orientation séparèment.

3. Il y a unicité des voisins suivant les quatre orientations. Si un composant R est voisin de C, il ne peut l'être que pour une seule orientation. Ceci va également simplifier grandement les algorithmes de gestion de voisinage, et les rendre d'autant plus efficaces.

III.3.6.10 Conclusion

Bien que notre voisinage ne dispose pas des propriétés d'un voisinage idéal, il en a quelques-unes essentielles, qui vont nous permettre tout au moins de fabriquer des algorithmes de gestion de voisinage rigoureux et efficaces.

Le formalisme que nous avons décrit ici sera repris dans la suite du texte, ceci permettant de disposer d'un vocabulaire précis concernant les entités manipulées.

III.3.7 Création et gestion distribuée de voisinage

Dans cette partie nous considérons qu'il n'y a jamais de conflits géométriques entre composants, c'est à dire que quelles que soient les opérations ou manipulations effectuées sur les composants, celles-ci ne provoquent jamais de situations conflictuelles.

Nous décrivons ici l'influence des manipulations d'un composant sur son voisinage et sur celui de ses propres voisins. Il s'agit de catégoriser l'influence de ces manipulations sur le voisinage d'un composant.

Puis nous proposons une série de trois algorithmes permettant de gérer la totalité des modifications possibles d'un voisinage, par suite d'une opération géométrique sur un composant. Ces algorithmes sont prévus pour fonctionner de manière distribuée.

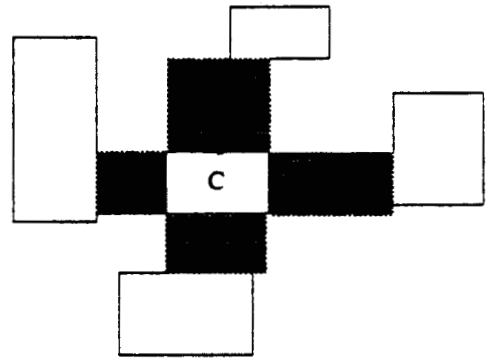
III.3.7.1 Opérations fondamentales applicables sur les composants

Si l'on ne tient pas compte du domaine d'application de schématique, on peut relever quatre opérations fondamentales applicables sur les composants :

1. la *création de composant*. Pour l'utilisateur d'un système de schématique, elle consiste à choisir un composant de la base du domaine d'application et à le placer dans son schéma. Pour notre système, il s'agit de l'introduction d'un nouveau composant dans l'espace. Il faut que ce composant puisse communiquer avec ses voisins; il faut donc déterminer son voisinage local, sachant qu'il existe déjà d'autres composants dont il faut tenir compte. Pour déterminer le voisinage du composant créé, il faut communiquer avec ces composants.

2. le *déplacement de composant*. Pour l'utilisateur, il s'agit d'une opération interactive courante dans tous les systèmes d'édition graphique. Pour notre système, la sémantique de cette opération est la suivante : la même que pour l'utilisateur, à ceci près que la modification de voisinage du composant déplacé, et de ses anciens et nouveaux voisins (ceux d'avant et d'après son déplacement) est gérée par les algorithmes spécialisés de notre système.

Nous avons vu dans le paragraphe III.3.3.6 concernant l'isotropie de la relation de voisinage, que cette propriété permet de gérer des déplacements dans toutes les directions. Au contraire, comme notre voisinage n'est pas construit à partir d'une relation isotropique, nous ne pourrons pas gérer ici des déplacements dans toutes les directions possibles. Le seul type de déplacement que nous pouvons prendre en compte simplement avec notre modèle est suivant une direction horizontale ou verticale, car les directions gérées par la relation de voisinage sont seulement horizontales ou verticales. La figure qui suit montre l'éventail des déplacements possibles pour un composant C dans son contexte local



Ceci veut dire que nous devons construire des algorithmes de modification de voisinage adaptés aux seuls déplacements horizontaux ou verticaux. La prise en compte d'autres types de déplacement pourrait également être prise en compte, mais au prix d'une grande complexité des algorithmes,et ne serait pas adapté au point de vue utilisateur (par exemple succession de déplacement Horizontaux et Verticaux pour simuler un déplacement oblique.

3. la *déportation de composant*. Cette opération consiste à prendre un composant et à le replacer loin de son contexte initial. Pour l'utilisateur, il s'agit également d'un déplacement. Cependant, pour notre système, la déportation se caractérise par l'impossibilité du modèle à gérer ce type de déplacement. En effet, le voisinage ne prenant en compte que quatre directions, les déplacements suivant une droite oblique ne peuvent traités simplement, car on ne sait pas simuler ces déplacements aisément. D'autre part, un déplacement sur une direction horizontale ou verticale qui fait "sortir" le composant de son contexte local initial empêche la mise à jour du voisinage du composant. La figure suivante montre un exemple de déportation.

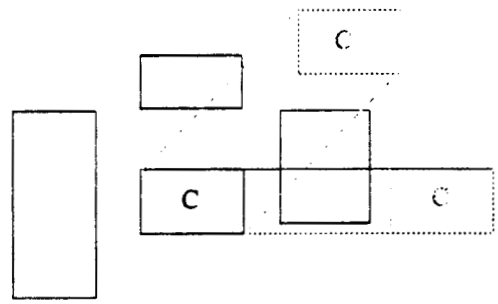


Fig III.17 Un exemple de déportation du composant C dans notre modèle .Un déplacement en oblique est une déportation car le voisinage considéré ne gére que que des directions horizontales et verticales. D'autre part, le déplacement horizontal provoque la traversée d'un autre composant, ce qui ne peut être pris en compte sans un recalcul complet du voisinage de C. Les deux déplacements symbolisés en pointillé sur le dessin sont en fait des déportations dans notre système.

Si la déportation parait identique au déplacement d'un point de vue utilisateur, elle est bien diffèrente du déplacement pour un composant pris en tant qu'agent. La déportation implique en fait une destruction de l'ancien contexte de l'agent et un recalcul du nouveau contexte local.

4. le *changement de taille d'un composant.* Cette opération est déclenchée par l'utilisateur lorsqu'il change un attribut du composant ayant rapport avec sa taille (par exemple un composant de même nature mais homothétiquement deux fois plus grand), ou bien quand il agit directement sur la forme du composant (la longueur d'un mur par exemple). Dans notre modèle, nous traitons cette opération comme la destruction de l'ancien composant et la création d'un nouveau.

III.3.7.2 Influence des quatre opérations fondamentales sur le voisinage

Pour chacune de ces opérations décrites précédemment, nous précisons les implications directes sur le contexte local du composant manipulé.

Création ou déportation d'un composant

Lorsqu'on effectue une telle opération, le composant doit être placé dans un endroit où l'on puisse l'y mettre. Il peut donc masquer des composants à d'autres composants. Par contre, il ne peut pas dévoiler de nouveaux composants. Dans ce cas, il ne peut donc y avoir que *disparition de voisins.* La figure suivante en montre un exemple.

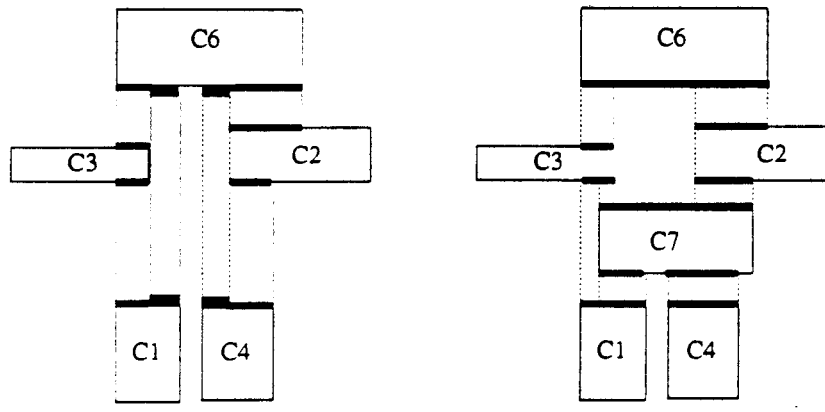


Fig III.18 Influence de la création de composant sur les voisinages.

Avant le placement de C7, on a : $V_{V-}(C6)$ = { C3, C2, C1, C4 }, $V_{V-}(C2)$ = { C4}, $V_{V+}(C1)$ = { C3, C6 } et $V_{V+}(C4)$ = { C2, C6 }. L'introduction de C7 dans ce contexte change les voisinages de C1, C2, C4, C6 : elle a pour effet de masquer des composants.

En effet, après placement de C7, on a : $V_{V-}(C6)$ = { C3, C2, C7 }, $V_{V-}(C2)$ = { C7}, $V_{V+}(C1)$ = { C3, C7 } et $V_{V+}(C4)$ = { C7 }.

Des voisins ont disparu pour chacun des composants C1, C2, C4, C6. C1 ne "voit" plus C6 car C7 le lui masque, C2 ne voit plus C4 car C7 le lui masque, C4 ne voit plus ni C2 ni

C6 car C7 les lui masque, et enfin C6 ne voit plus ni C1 ni C4 car C7 les lui masque. En fait, on constate que l'introduction de C7 , si celui-ci devient un nouveau voisin de C1, C2, C4, C6, a aussi pour effet de leur cacher des voisins, et il y donc disparition de voisins.

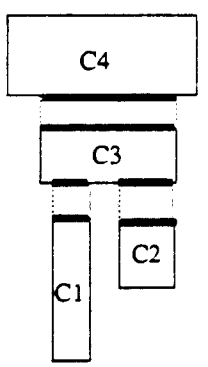
Il faut noter que la création d'un composant C ne peut que cacher des voisins aux composants du contexte local de C (voisinage de C), elle ne peut pas en faire apparaître d'autres car C ne fait que contraindre un peu plus le " champ de vision" des composants de son voisinage.

Régle : Pour tout composant C nouvellement créé, les composants R appartenant à $V_{DS}(C)$ peuvent perdre des voisins dans $V_{D\hat{S}}(C)$.

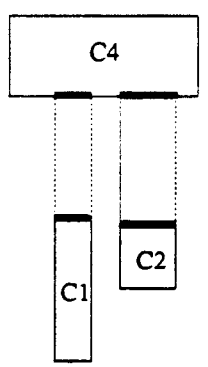
Il faut donc que chaque composant dispose d'un algorithme de suppression d'anciens voisins, que nous appelerons ici simplement un algorithme de disparition de voisins.

Destruction ou déportation de composant

La destruction est l'opération symétrique de la création, c'est à dire que le composant détruit va libérer une zone de l'espace et donc élargir le champ de vision de ses voisins. L'influence sur les voisins du composant va donc consister en une *apparition de voisins.* Lors d'une déportation de composant , on a d'abord une destruction de composant, d'où une influence identique lors de cette première phase. La figure suivante montre un exemple de destruction de composant. .



Voisinages avant destruction de C3



Voisinages après destruction de C3

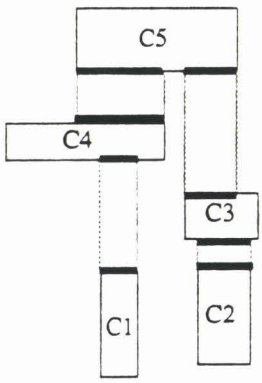
Avant que C3 ne soit détruit, on a $V_{V+}(C1) = V_{V+}(C2) = \{ C3 \}$, $V_{V+}(C3) = \{ C4 \}$, $V_{V-}(C4) = \{ C3 \}$, $V_{V-}(C3) = \{ C1, C2\}$. La destruction de C3 rend le composant C4 visible à C1 et C2 et inversement. On a alors $V_{V+}(C1) = \{ C4 \} = V_{V+}(C2)$ et inversement $V_{V-}(C4) = \{C1, C2\}$. Le composant C3 disparait bien sur des voisinages de C1, C2, C4, mais le voisinage de ces derniers s'enrichit de nouveaux agents.

Régle :Pour tout composant R appartenant à $V_{DS}(C)$, la destruction de C peut faire apparaître de nouveaux composants dans $V_{DS}(R)$.

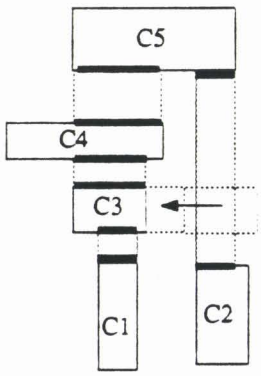
Il faut donc que chaque composant dispose aussi d'un algorithme de détection de nouveaux voisins, que nous appelerons ici simplement un algorithme d'apparition de voisins.

Déplacement de composant

L'opération de déplacement de composant est plus complexe à gérer, car d'anciens voisins vont disparaître et de nouveaux voisins vont apparaître. La figure suivante illustre ce phénomène.



Voisinages V avant déplacement de C3



Voisinages V après déplacement de C3

En se déplacant à l'ouest, C3 a perdu ses deux voisins C2 et C5, mais a gagné les composants C1 et C4 comme nouveaux voisins. Par contre C2 a gagné C5 comme nouveau voisin, et réciproquement, et C4 et C1 ne sont plus voisins l'un de l'autre car ils sont masqués par C3.

Lors d'un déplacement d'un composant C, les voisins de C dans sa position initiale peuvent avoir de nouveaux voisins, il y aura donc apparition de voisins, et les nouveaux voisins de C peuvent avoir des voisins qui seront supprimés.

Régle : Quand un composant C se déplace dans une direction D, on ales trois posibilités suivantes :

1. les anciens voisins R de C appartenant à $V_{D^-}(C)$ (resp. $V_{D^+}(C)$), V désignant l'ancien voisinage de C, peuvent avoir de nouveaux voisins $V_{D^+}(C)$ (resp. $V_{D^-}(C)$).

2. Les nouveaux voisins R de C appartenant à $V'_{D^-}(C)$ (resp. $V'_{D^+}(C)$), V' désignant le nouveau voisinage de C, peuvent avoir certains de leurs voisins V_{D^+} (resp. V_{D^-}) qui disparaissent.

3. Les anciens voisins de C qui sont restés voisins de C peuvent subir les deux types d'influence précédente.

Les composants doivent donc être capables lors d'un déplacement de gérer correctement les apparitions et les disparitions de voisins.

III.3.7.3 Comportement général d'un composant lors d'une modification

C'est le composant qui a subi une modification directe de son contexte (à la suite d'une création, d'une destruction, d'un déplacement, d'une déportation) qui va déclencher le recalcul distribué des voisinages des composants appartenant au voisinage du composant modifié. En envoyant des messages à ses voisins, anciens voisins et nouveaux voisins, ils les informe qu'ils doivent mettre à jour leur propre voisinage. Notons d'autre part que le composant modifié connait exactement le type d'influence de la modification qu'il subit sur ses propres voisins.

Les algorithmes de modification pour un composant modifié sont toujours de la forme :

1. [facultatif] Je crée mon voisinage local (ou je crée un nouveau voisinage local sans détruire l'ancien)

2. J'informe mes voisins (nouveaux si nécéssaire) qu'une modification d'une certaine catégorie est intervenue :

   création (placement) ou déportation — disparition de voisins
   destruction ou déportation — apparition de voisins
   déplacement — apparition et disparition de voisins

   J'informe également mes anciens voisins de la modification intervenue.

3. Quand tous mes voisins (anciens et nouveaux) ont terminé leur mise à jour, je valide mon voisinage (le nouveau a priori), et je détruis l'ancien.

La partie numérotée 1 est facultative en ce sens qu'elle ne sera pas exécutée lors d'une destruction de composant. Les mises à jour ont l'avantage de n'être à faire que sur les voisins (anciens et/ou nouveaux) du composant modifié. Ceci est dû au fait que la relation de voisinage est symétrique.

Le composant modifié précise à ses voisins, par le type de message envoyé, le type d'algorithme qu'il doit effectuer pour mettre à jour son voisinage.

III.3.7.4 Comportements spécialisés des composants lors d'une modification

placement (création) d'un composant

Lorsqu'un composant C est créé ou placé (lors d'une déportation), il effectue les actions suivantes :

1. Je calcule mon nouveau voisinage, grâce à l'algorithme de calcul de voisinage (que nous décrivons plus loin).

2. J'l informe tous mes voisins R appartenant à $V_{DS}(C)$ que leur voisinage $V_{DŜ}(R)$ risque d'être modifié par suite de disparitions de voisins et qu'ils doivent donc procéder à une remise à jour de ce voisinage.

Destruction d'un composant

Lorsqu'un composant C se détruit (soit par suite d'une véritable destruction ou d'une déportation), il effectue les actions suivantes :

1. Il informe tous ses voisins R appartenant à $V_{DS}(C)$ qu'une modification de leur voisinage $V_{DŜ}(R)$ est possible et qu'ils doivent rechercher de nouveaux voisins dans $V_{DŜ}(C)$, par suite d'une éventuelle apparition de voisins. Il les informe également qu'ils doivent détruire toute référence à C dans leur voisinage $V_{DŜ}(R)$.

2. Il s'auto-détruit.

Déplacement horizontal ou vertical d'un composant

Lorsqu'un composant C se déplace horizontalement ou verticalement, il effectue la série d'opérations ou de messages suivante :

1. Il calcule son nouveau voisinage (appelé *NouveauxVoisins)* sans détruire l'ancien, il ne doit le faire que dans la direction opposée à son sens de déplacement.

2. Il détermine alors trois nouveaux ensembles de voisins :
   a. Les voisins qui sont restés voisins (nous appellerons cet ensemble de voisins "VoisinsInchangés"). On a :
   $$VoisinsInchangés = AnciensVoisins \cap NouveauxVoisins$$
   b. Les voisins qui ont disparu (que nous appelerons ici "VoisinsDisparus"). On a :
   $$VoisinsDisparus = AnciensVoisins \ominus VoisinsInchangés$$
   c. Les voisins qui sont apparus (que nous appelerons ici "VoisinsApparus"). On a :
   $$VoisinsApparus = NouveauxVoisins \ominus VoisinsInchangés$$

3. Il informe tous les composants R de *VoisinsDisparus* et appartenant à $V_{DS}(C)$ qu'ils doivent rechercher de nouveaux voisins $V_{DŜ}$, car une apparition de voisins est possible pour eux. Il les informe également qu'il n'est plus un de leurs voisins $V_{DŜ}$.

4. Il informe tous les composants R de *VoisinsApparus* et appartenant à $V_{DS}(C)$ qu'ils doivent déterminer si certains de leurs voisins $V_{DŜ}$ n'ont pas disparu, car une disparition de voisins est possible pour eux. C devient bien sûr pour eux un nouveau voisin $V_{DŜ}$.

5. Il informe tous les composants R appartenant à *VoisinsInchangés* et appartenant à $V_{DS}(C)$ que :
   a. Ils doivent rechercher de nouveaux voisins V_{DS}, car une apparition de voisins est possible pour eux.
   b. Ils doivent déterminer si certains de leurs voisins V_{DS} n'ont pas disparu, car une disparition de voisins est possible pour eux.

6. Il détruit son ancien voisinage et conserve le nouveau.

Déportation d'un composant

La déportation de composant est traitée comme une destruction suivie d'une création de composant. Lors de cette opération, les comportements de création et de destruction de composants existent tous deux.

Changement de taille

Le comportement est le même que pour la déportation de composant, on traite d'abord la destruction du composant, puis une création de composant d'une taille différente.

Remarques :

1. Lors d'une création ou d'une destruction de composant C, toutes les orientations DS possibles de voisinage de C sont concernées par l'apparition ou la disparition de voisins.

2. Lors d'un déplacement de direction D pour un composant C, seuls les voisins R de C appartenant à $V_{\bar{D}^-}(C)$ et $V_{\bar{D}^+}(C)$ sont concernés par une modification éventuelle de leur voisinage par apparition ou disparition de voisins. Ceci est dû au fait que la relation de voisinage n'est pas isotropique. Une relation isotropique aurait nécéssité d'examiner tous les voisins de C.

3. Ces comportements et les algorithmes décrits ci-après ne sont valables que parce que l'on fait l'hypothèse que les modifications apportées ne provoquent jamais de conflits géométriques. Cette éventualité fait l'objet de la parti de résolution intelligente décrite plus loin.

4. L'interface utilisateur a un comportement particulier du à l'existence des voisinages de composants. En effet, certaines actions de l'utilisateur sur un composant C sont automatiquement contraintes au voisinage de C. Lors d'un déplacement de C, celui-ci ne peut se faire que dans l'espace disponible jusqu'à rencontrer le voisin le plus proche de C dans cette direction. De même, le changement de taille d'un composant ne peut se faire que dans l'espace laissé complétement libre par les voisins les plus proches de C.

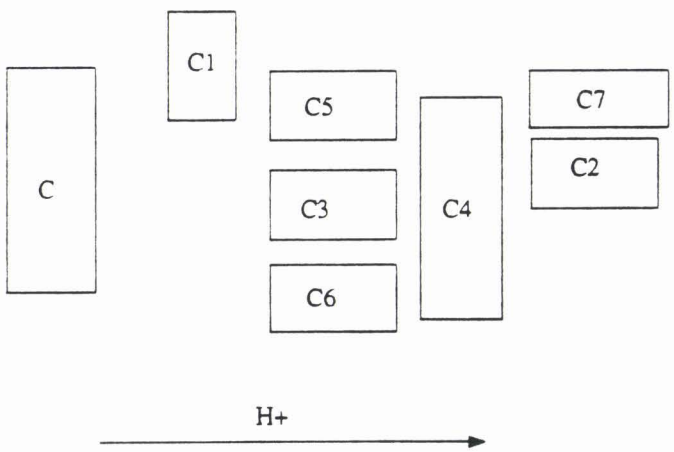
III.3.7.5 Algorithme de création de voisinage

Nous décrivons ici l'algorithme de création de voisinage qui est exécuté par un composant quand il se crée. Cet algorithme est basé sur la construction par récurrence réalisée à partir de l'équation 1 décrite dans le paragraphe III.3.6.8.

Introduction des notations

Pour avoir une description syntaxique moins lourde de l'algorithme, nous reprenons et simplifions les notations utilisées dans le paragraphe III.3.6.8.

Rappelons tout d'abord que l'ensemble $I_{DS}(C)$ est ordonné par ordre croissant des distances $d_{DS}(C)$ de ses éléments, et que si deux éléments R1 et R2 sont situés à la même distance $d_{DS}(C)$, ils sont alors rangés par ordre croissant des ordonnées $O_{\vec{D}+}(C)$. Ceci est illustré par la figure suivante.



Sur ce dessin, on a $I_{H+}(C)$ = { C1, C6, C3, C5, C4, C2, C7 }.

Nous avons vu que $[I_{DS}(C)]_1 = [V_{DS}(C)]_1 = v1$.

Quel que soit i, Mi = I_D(vi, C). Et $\forall i > 1,\ v_i = [V_{DS}(C)]_i$

L'équation 1 s'écrit alors : $\forall n > 1,\ M_n \not\subset \bigcup_{i=1}^{n-1} M_i$

Algorithme de calcul du voisinage d'un composant

L'algorithme se déroule en deux phases : la construction des ensembles $I_{DS}(C)$, pour les quatre orientations DS, puis la construction des quatre voisinages $V_{DS}(C)$, pour les quatre orientations DS possibles.

```
    /co fabrication et calcul des 4 intersections DS co/
— Création des 4 ensembles I(H-), I(H+), I(V-), I(V+).
pour tous les composants R de la base du projet :
     /co Fabrication des I_V co/
     — ? Y a-t-il intersection verticale entre C et R (I_V(R, C) <> 0) ?
     si oui (I_V(R, C) <> 0)
      si R est du côté V+ de C
        — mémoriser sa référence dans I_V+(C),
           suivant l'ordre croissant des distances d_V+(R, C),
           et en cas d'équidistance suivant les coordonnées croissantes H+(R).
      sinon (R est du côté V- de C)
        — mémoriser la référence de R dans I_V-(C),
           suivant l'ordre croissant des distances d_V-(R, C),
           et en cas d'équidistance suivant les coordonnées croissantes H+(R).
      fsi


        /co Fabrication des I_H co/
     sinon (I_V(R, C) = 0)
      ? Y a-t-il intersection horizontale entre C et R (I_H(R, C) <> 0) ?
      si oui (I_H(R, C) <> 0)
       si R est du côté H+ de C
        — mémoriser sa référence dans I_H+(C),
           suivant l'ordre croissant des distances d_H+(R, C),
           et en cas d'équidistance suivant les coordonnées croissantes V+(R).
       sinon (R est du côté H- de C)
        — mémoriser la référence de R dans I_H-(C),
           suivant l'ordre croissant des distances d_H-(R, C),
           et en cas d'équidistance suivant les coordonnées croissantes V+(R).
       fsi
      fsi
     fsi
fpour

    /co construction des quatres voisinages DS de C co/
— création des 4 ensembles V(H, —), V(H, +), V(V, —), V(V, +).
Pour les deux directions H et V : (D)
  Pour les deux sens — et + : (S)
      si I_DS(C) <> 0
       — mémoriser la référence à [I_DS(C) ]_1 comme le voisin [V_DS(C) ]_1
       — initialiser la partie invisible à C avec I_D([ I_DS(C)]_1, C).
       — initialiser i à 1 (ind = indice de parcours de I_DS(C) )


          — tantque partie invisible à C ⊅ b_D(C) et i < CARD( I_DS(C) )
          — calcul de M_i.
          si M_i ⊄ partie invisible à C
           — partie invisible à C  — partie invisible à C ⋃ M_i.
           — mémoriser la référence à [I_DS(C)]_i comme voisin DS de C.
          fsi
          — incrémenter i.
        ftant que
       fsi
       —détruire I_DS(C).
  fpour
fpour.
```

Remarques

La partie calculant les intersections DS du composant C est assez mécanique. Pour chaque composant de la base de projet, on détermine si son intersection verticale ou horizontale avec C est vide ou non. Si elle n'est pas vide, ce composant appartient à l'intersection DS de C.

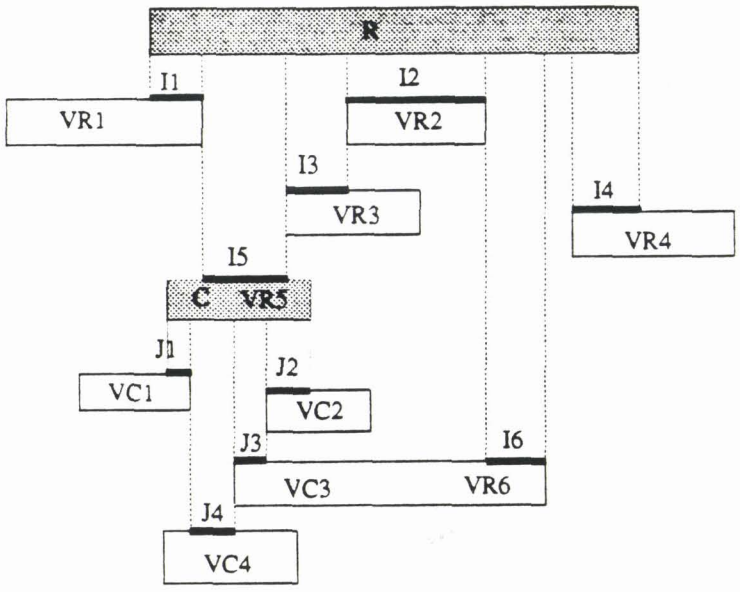
Par contre, pour la construction des quatrs voisinages DS de C, on manipule les intervalles déterminés par les $I_{DS}(C)$. En particulier, on teste si un bord est inclus dans une union d'intervalles (*"partie invisible à C* $\not\supset$ $b_D(C)$" et " $M_i \not\subset$ *partie invisible à C"*), et on calcule aussi des unions d'intervalles (" *partie invisible à C* $\bigcup M_i$ "). Ces mécanismes de calcul et de manipulation d'ensembles d'intervalles doivent être implémentés dans le code de l'interface ou être disponibles dans l'environnement logiciel utilisé. Dans notre cas nous avons créé une classe d'objet spécialisé : l'union d'intervalles. Lors de la description de l'implantation du système, nous rentrerons en détail dans les fonctionnalités de cet objet.

III.3.7.6 Algorithme d'apparition de voisins

Afin de faciliter la compréhension de cet algorithme, nous commencerons par décrire un exemple assez complet de cas d'apparition de voisins. Nous précisons ensuite le contexte dans lequel est exécuté cet algorithme, nous décrivons ensuite la méthode de détermination de nouveaux voisins, et enfin nous donnons le code source informel de l'algorithme.

Un exemple démonstratif

La figure suivante montre le voisinage V- d'un composant R. Nous supposerons que le composant VR5 est détruit par l'utilisateur. Le but est de déterminer les nouveaux voisins V- de R qui sont devenus visibles à R par suite de la destruction de VR5.



Sur cette figure, les voisins V- de R sont notés VR_i, et les voisins V- de C sont notés VC_i. On notera que le composant VR6, qui est à la fois voisin V- de R et de C, est noté différemment suivant qu'on le désigne en tant que voisin de C ou de R. Néanmoins, il n'existe qu'un seul composant qui corresponde dans la base du projet. VC3 et VR6 ne sont que des synonymes.

Rappelons que l'on peut avoir apparition de voisins dans $V_{V-}(R)$ si C est détruit ou déplacé. Les nouveaux voisins possibles de R sont ceux de $V_{V-}(C)$ qui n'appartiennent pas à $V_{V-}(R)$.

Intuitivement, on voit que la destruction de C va provoquer l'apparition de deux nouveaux voisins : VC2 et VC4. En effet, VC1 va rester masqué à R par VR1, et VC3 = VR6 est déjà voisin de R. Seuls VC2 et VC4 seront démasqués lors de cette destruction de composant.

Après destruction de C, on a :
— la zone libérée par C et visible pour R est I5,

— $VC_1 \notin V_{V-}(R)$, car J1 $\in \bigcup_{i=1}^{4} M_i(R)$ (J1 est complétement inclus dans I1).

— $VC_2 \in V_{V-}(R)$, car J2 $\notin \bigcup_{i=1}^{4} M_i(R)$. (*on a* $I5 \cap J2 \neq \emptyset$ *et* $I3 \cap J2 \neq \emptyset$)..

— $VC_3 \in V_{V-}(R)$, on le savait déjà car VC3 = VR6.

—$VC_4 \in V_{V-}(R)$. *car* $J4 \notin \left(\left(\bigcup_{i=1}^{4} M_i(R) \right) \cup \left(\bigcup_{j=1}^{3} M_j(C) \right) \right)$ *et* $(J4 \cap I5 \neq \emptyset)$.donc VC4 est visible à R.

— la zone masquée à R aprés que C soit détruit est :

$$\text{zoneMasquée} = \left(\left(\bigcup_{i=1}^{4} M_i(R) \right) \cup \left(\bigcup_{j=1}^{3} M_j(C) \right) \right).$$

Le principe de détermination de nouveaux voisins parmi $V_{V^-}(C)$ est le suivant : en utilisant la relation de récurrence qui est à la base de la construction de voisinage, on calcule la zone libérée par C (un intervalle), et on détermine progressivement les voisins V- de C dont le bord V intersecte cette zone.

principe de détermination des nouveaux voisins

Lorsqu'un composant C est détruit ou déplacé, pour un composant R qui est son voisin DS, il libère un intervalle $\bar{D}$ (que nous appellerons ici *zoneLibérée*) et certains composants de $V_{D\hat{S}}(C)$ peuvent donc être visibles pour R.

Seuls les composants de $V_{D\hat{S}}(C)$ qui n'appartiennent pas à $V_{D\hat{S}}(R)$ doivent être considérés.

Pour qu'un composant de $V_{D\hat{S}}(C)$ devienne un nouveau voisin $D\hat{S}$ de R, il faut et il suffit qu'il recouvre une partie de la zoneLibérée par C et que cette partie ne soit pas déjà complètement recouverte par les prédecesseurs de ce composant (suivant l'ordre croissant de la distance $d_{D\hat{S}}$).

En appelant $A = \left(\bigcup_{i=1}^{\text{indice}(C)-1} I_{\bar{D}}([V_{D\hat{S}}(R)]_i, R) \right)$,

la zoneLibérée vaut : $\text{zoneLiberee} = (A \cup I_{\bar{D}}(C, R)) - A.$

Chaque fois qu'un composant v de $V_{D\hat{S}}(C)$ est ajouté aux $V_{D\hat{S}}(R)$, on recalcule la zoneLibérée qui devient $\text{zoneLiberee} = \text{zoneLiberee} - I_{\bar{D}}(v, R)$.

Si une destruction de C est prévue, on détruit sa référence dans $V_{D\hat{S}(R)}$.

Le composant C détruit ou déplacé envoie à tous ses voisins V_{DS} le message suivant :

- — je suis le composant : C,
- — je suis situé dans votre direction : D,
- — je suis pour vous dans le sens : $\bar{S}$,
- — notre intersection D est actuellement : $I_{\bar{D}}(C, R)$
- — mon voisinage V_{DS} est : $V_{D\hat{S}}(C)$,
- — autodestruction : *vrai* ou *faux*.

Algorithme d'apparition de voisins

Le composant R qui reçoit le message précédent exécute l'algorithme d'apparition de voisins suivant :

— je sélectionne mon voisinage $V_{D\hat{S}}$;
— je crée et initialise un ensemble A à ∅,
si $V_{D\hat{S}}(C) = \emptyset$,
— détruire la référence à C dans $V_{D\hat{S}}(R)$ si nécessaire;
sinon
/co construire le recouvrement de R avant C co/
pour tous les $v \in V_{D\hat{S}}(R)$ et du côté $D\hat{S}$ de C
$A \leftarrow A \cup I_D(v, R)$;
fpour
/co déterminer la zoneLibérée co/
— zoneLibérée ← $(A \cup I_D(C,R)) - A$;
si autodestruction vraie
— détruire la référence à C dans $V_{D\hat{S}}(R)$;
fsi

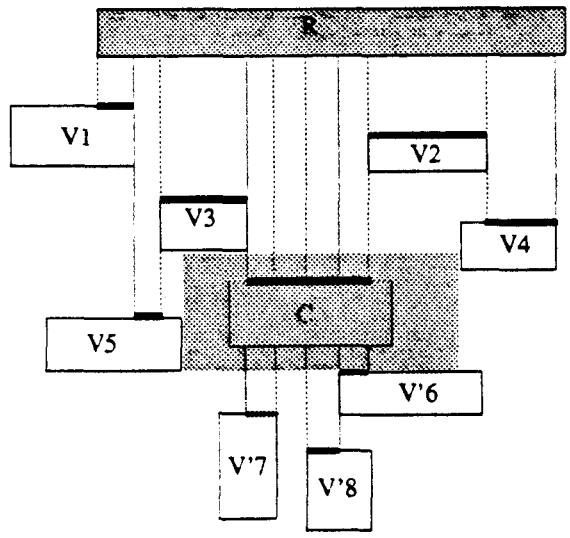
pour tous les $v'' \in V_{D\hat{S}}(C)$
si $\text{zoneLiberee} \cap b_{\tilde{D}}(v'') \neq \emptyset$
— ajouter v" à $V_{D\hat{S}}(R)$ dans le bon ordre
— zoneLiberee ← zoneLiberee — (zoneLiberee $\cap$ $b_D(v'')$;
fsi
fpour
fsi

III.3.7.7 Algorithme de disparition de voisins

Nous décrivons ici l'algorithme de disparition de voisins, en présentant d'abord un exemple assez complet et démonstratif, en introduisant les idées principales sur lesquelles l'algorithme est construit, en expliquant très informellement le fonctionnment, et enfin en présentant un texte source informel.

Un exemple démonstratif

La figure suivante montre le voisinage V- d'un composant R. Nous supposerons que le composant C est créé (placé) par l'utilisateur. Le but est de déterminer les voisins V- de R qui disparaitront quand C est cr´éé.



Le schéma représente le voisinage V- du composant R avant et après création du composant C. Avant le placement de C, les voisins V- de R sont les composants V1 à V8. Après création de C, les trois composant V6 à V8 ont disparu. La zone grisée entourant C représente l'espace dans lequel le concepteur désire placer le nouveau composant.

Si l'utilisateur place un composant dans cette zone sans créer de conflit géométrique, les voisins V1 à V4 restent voisins car ils sont moins éloignés de R que C. De même, le composant V5 reste voisin car I_H(V5, C) n'est pas vide, donc il ne peut être masqué par C que si C le recouvre en partie, donc en provoquant un conflit géométrique.

Par contre, V6 à V8 peuvent disparaitre car ils seront voisins V- de C. En effet, la zone de R qui leur est masquée par C peut en fait leur rendre R complètement invisible.

Finalement, nous pouvons déduire de la position de C par rapport aux voisins initiaux de R ce qu'il peut advenir du voisinage V- de R après création de C.

Fondements de l'algorithme

On suppose que le composant C est créé ou déplacé sans provoquer de conflit géométrique. R devient alors un de ses nouveaux voisins DS. Donc C est un voisin $D\hat{S}$ de R.

Le placement de C peut provoquer une disparition de voisins $D\hat{S}$ de R. Les trois règles de base pour déterminer si un voisin $D\hat{S}$ de R disparait sont :

1. *les voisins* $v \in V_{D\hat{S}}(R)$ *qui sont du côté DS de C ne peuvent pas disparaître.* En effet, ces composants étant plus proche de R que C, le placement de ce composant ne peut pas les masquer à R.

2. *les voisins* $v \in V_{D\hat{S}}(R)$ *qui sont tels que* $I_{\bar{D}}(v, C)$ *est non vide ne peuvent pas non plus disparaître.* En effet, puisque C ne peut être placé en provoquant un conflit, et que les deux composants C et v sont placés l'un à côté de l'autre dans la direction $\bar{D}$, le composant v consituera en fait une barrière de placement pour C. Par conséquent C ne peut pas masquer le composant v à R.

3. *les voisins* $v \in V_{D\hat{S}}(R)$ *qui sont du côté* $D\hat{S}$ *de C disparaîtront de* $V_{D\hat{S}}(R)$ *si l'équation 1 n'est plus vérifiée pour v.* En effet, si v est du côté $D\hat{S}$ de C, alors C est un voisin plus proche de R que V (C est un prédécesseur de v dans $V_{D\hat{S}}(R)$, et son indice dans $V_{D\hat{S}}(R)$ est inférieur à celui de v). Ainsi v ne sera plus un voisin $D^{-}\hat{S}$ de R si l'on a plus $I_D(v, R) \not\subset \bigcup_{i=1}^{j} M_i$, j étant l'indice de v dans $V_{D\hat{S}}(R)$.

L'algorithme est basé essentiellement sur ces trois régles.

Déroulement de l'algorithme

Grâce aux trois règles énoncées précédemment, l'algorithme est très simple et se déroule de la manière suivante :

1. construire par récurrence l'union des ensembles Mi, pour i allant de 1 à l'indice précédent celui de C dans $V_{D\hat{S}}(R)$.

2. Pour tous les voisins $D\hat{S}$ de R qui sont en intersection $\bar{D}$ avec C, continuer de construire l'union des ensembles Mi

3. Pour tous les voisins $D\hat{S}$ de R qui sont du côté $D\hat{S}$ de C, déterminer s'ils vérifient l'équation 1. S'ils la vérifient, ils restent voisins $D\hat{S}$ de R, sinon ils sont supprimés de $V_{D\hat{S}}(R)$. Pendant le même temps, on continue de construire l'union des ensembles Mi jusqu'à recouvrement total du bord $\bar{D}$ de R par cette union.

NOTE : le composant R exécute l'algorithme de disparition du voisinage sur réception d'un message spécialisé de C :

— je suis : C,
— je suis dans votre direction : D,
— je suis dans votre sens : $\bar{S}$,
— notre intersection est : $I_D(C,R)$.

Code source informel de l'algorithme

Sur réception du message précédent, R réalise l'algorithme suivant :

si $V_{D\bar{S}}(R)$ est vide
— ajouter C comme nouveau voisin $D\bar{S}$ de R.
sinon
— création de A /co A est destiné à représenter l'union des Mi co/
pour tous les $v \in V_{D\bar{S}}(R)$ **et** $\in C_{DS}(C)$
$A \leftarrow A \bigcup I_D(v,R)$
fpour

pour tous les $v \in V_{D\bar{S}}(R)$ **et** tels que $I_{\bar{D}}(v,C) \neq \emptyset$
$A \leftarrow A \bigcup I_D(v,R)$
fpour

— ajouter C à $V_{D\bar{S}}(R)$ /co si nécessaire et au bon indice co/
pourt ous les $v \in V_{D\bar{S}}(R)$ **et** $\in C_{D\bar{S}}(C)$
si $I_D(v, R) \subset A$
— détruire la référence à v dans $V_{D\bar{S}}(R)$.
sinon
— $A \leftarrow A \bigcup I_D(v,R)$
fsi
fpour
fsi

III.3.7.8 Conclusion

Le voisinage que nous avons choisi nous permet de construire des algorithmes de gestion de l'ensemble des agents pertinents d'un composant.

Nous devons cependant insister sur les particularités suivantes :

1. Nous avons toujours supposé dans cette partie que les manipulations de l'utilisateur n'impliquaient aucun conflit géométrique entre composants à l'issue de l'action. C'est cette règle qui nous permet d'avoir toujours les propriétés du voisinage respectées. Dès qu'il y a un conflit géométrique entre composants, les propriétés du voisinage ne sont plus respectées, en particulier la symétrie de la relation de voisinage. Cette dernière propriété est capitale puisque c'est grâce à elle que nous avons pu élaborer les algorithmes de gestion du voisinage.

L'introduction de conflits géométriques fait l'objet de la partie ultérieure de résolution distribuée. Lors de cette résolution, les algorithmes précédents seront utilisés à des étapes intermédiaires pour modifier les voisinages de composant.

2. Les composants peuvent travailler en tant que maitre, en réalisant toutes les tâches de gestion du voisinage seul (c'est le cas pour la création de composant où la seule délégation consiste à informer un nouveau voisin qu'il doit rajouter ce composant dans son voisinage).

Les composants peuvent aussi travailler en délégation, en réalisant la mise à jour de son propre voisinage su message spécialisé d'un composant. C'est ce qui se passe lorsque l'utilisateur détruit ou déplace un composant. Celui—ci gère lui-même son voisinage, puis délègue à chacun de ses voisins leur propre mise à jour de voisinage.

3. Lorsque l'utilisateur effectue un déplacement de composant, celui ci travaille à la fois sur ses voisins avant modification et après modification. Ceci complique la gestion de cette opération, par rapport à un traitement où l'on détruirait le composant et on le placerait ailleurs, mais rend d'un autre côté cette gestion de voisinage plus efficace.

Finalement, à partir de ces outils, on peut créer une interface où chaque composant dispose d'une connaissance de son contexte local, où chaque action interactive peut être dirigée suivant le contexte du composant manipulé, mais à condition de ne jamais provoquer de conflit géométrique lors des manipulations de composant par l'utilisateur.

Dans la partie suivante, nous décrivons une implémentation d'une telle interface, en détaillant peu les aspects strictement techniques.

III.3.8 Conclusion

Le modèle MAPS reprend les modèles totalement décentralisés que l'on peut trouver dans la littérature concernant l'IA distribuée [Buisine 89, Demazeau 90, Ferber 89], qui convient bien à l'aspect *microscopique* des agents à modéliser dans notre cas. Il est d'autre part bien adapté à une satisfaction de contraintes, un à raisonnement et à un traitement local pour les conflits géométriques lors d'un problème d'aménagment spatial. Il convient donc bien au type d'aide à la conception que nous souhaitons offrir à l'utilisateur.

Dans la partie que nous venons de décrire, nous ne nous sommes intéressé qu'à l'aspect "vie sociale" des agents en proposant un modèle et des mécanismes permettant la gestion distribuée des accointances des agents (agents pertinents). Il s'agit de faire prendre conscience à un agent de son entourage d'autres agents. Les agents que nous traitons ici sont *microscopiques*; Ils ne disposent pas d'une connaissance sophistiqué et sont basés sur les mêmes principes de d'existence et de fonctionnement.

Nous avons montré l'importance de la relation de voisinage et les bonnes qualités de celui-ci pour disposer d'un modèle riche concernant la modification géométrique. C'est des qualités de cette relation que dépendra la richesse de modélisation de l'environnement local du composant, et donc la puissance à gérer des modifications ne créant pas de conflits.

Nous avons d'autre part élaboré des algorithmes de gestion de voisinage indépendants du voisinage choisi (exception faite des quatres directions cablées dans les algorithmes et structures de données).

Nous réussissons ici à gérer la base de composants existant de manière distribuée tout en conservant la cohérence des relations existantes entre composants, et c'est pourquoi chaque composant est capable de gérer indépendamment des autres son propre voisinage. Néanmoins il semble indispensable de disposer d'une entité connaissant la totalité des agents existants, pour pouvoir déterminer l'ensemble des agents pertinents lors de la création d'un composant, d'une modification de forme ou encore d'une déportation.

Cependant, le modèle MAPS tel que nous l'avons décrit ici est incapable de résoudre les conflits géométriques qui sont susceptibles d'être occasionnés par la modification locale du schéma par l'utilisateur final. Nous présenterons dans la section III.5 deux algorithmes de résolution de conflit géométrique adaptés au modèle MAPS. Mais nous présentons d'abord la réalisation de l'interface intégrant le modèle MAPS.

III.4. Implémentation de l'interface

Nous allons ici décrire l'implémentation que nous avons réalisée pour cette interface, sachant que dans cette partie on supposera que l'utilisateur ne produit jamais de conflit entre composants en les manipulant.

III.4.1 L'environnement de développement logiciel

Le modèle que nous avons choisi étant du type multi-agents, il est naturel de penser à implémenter les composants comme des acteurs. Pour cela, on peut utiliser un langage spécialisé (comme PLASMA, MEHRING, KRL ...). D'autre part, des langages de frames permettraient d'implémenter les délégations et connexions entre composants grâce à des techniques implicites d'attachement procédural. On peut aussi représenter ces fonctionnalités impliquées par le modèle d'une manière explicite avec le langage utilisé.

D'autre part, comme ce modèle était destiné à être traduit dans un environnement logiciel utilisant le langage ADA, il était important de minimiser l'effort de traduction nécessaire pour passer du langage spécialisé du prototype au langage ADA.

Il faut également disposer d'un environnement supportant l'interaction graphique (représentation graphique des composants à l'écran, fenêtre spécialisée d'édition graphique, ...). Et il faut également que les performances du système graphique soient suffisantes pour évaluer réellement l'intérêt interactif de l'efficacité de l'interface.

Enfin, vu l'aspect distribué du fonctionnement de l'interface, et le fait que cette implémentation n'était qu'un prototype de ce qui devait être fait de manière industrielle, il fallait que l'environnement que nous utilisions dispose de capacités de prototypage importantes.

Pour ces raisons, nous avons choisi de développer notre prototype d'interface dans l'environnement logiciel fourni avec les implémentations du langage SMALLTALK. Il remplit dans notre cas beaucoup des conditions citées précédemment : un bon environnement de développement et de prototypage de logiciel, une bonne boite à outils pour le développement d'interfaces graphiques interactives, un bon modèle objet permettant de fabriquer simplement les composants, une bibliothèque de composants logiciels réutilisables importante.

Nous avons donc développé notre interface dans l'environnment SMALLTALK, en essayant de réaliser du logiciel facilement portable dans un autre langage classique avec lequel on puisse facilement construire des types abstraits, ceci afin de faciliter le portage de l'environnement orienté objet de SMALLTALK à ce langage.

III.4.2 Architecture logicielle de l'implémentation

Nous présentons ici les idées qui nous ont amenés à l'architecture logicielle implémentée pour cette interface. nous montrons la nécessité des diverses classes constituant la totalité du logiciel.

III.4.2.1 Implémentation des agents dans l'interface

Nous avons vu dans les paragraphes précédents que nous voulions modéliser les composants comme des acteurs (ou agents) d'un système multi-agents. Le modèle que nous avons élaboré permet à chaque agent de déterminer et de communiquer avec l'ensemble de ses agents *pertinents.*

L'implémentation de ce modèle se fait simplement et naturellement en SMALLTALK. Nous avons créé une classe *ComposantRectangle* qui reprend les fonctionnalités de nos agents du modèle. Les composants seront des instances de cette classe.

Chaque objet de la classe ComposantRectangle dispose d'un très petit nombre d'attributs permettant de le décrire et de le manipuler complètement dans notre modèle. Ces attributs sont : l'identificateur du composant dans le projet en cours (*nom*), le rectangle enveloppe de ce composant (*enveloppe*), les voisins du composant (*voisins*), et le projet auquel ce composant est rattaché (*projet*).

Les instances de la classe ComposantRectangle disposent d'une série de méthodes implémentant les algorithmes de création de voisinage, d'apparition de voisins, de disparition de voisins que nous avons décrits précédemment et qui permettent de gérer le voisinage d'un composant.

D'autres méthodes d'instance de ComposantRectangle concernent les actions directes de l'utilisateur sur un composant : il s'agit essentiellement du placement d'un composant (création), du déplacement d'un composant, de la destruction d'un composant.

Il existe enfin d'autres méthodes d'instance pour cette classe, qui sont locales au composant (accessibles uniquement par une instance de cette classe), ou bien liées à l'implémentation de la classe elle-même.

Nous reviendrons un peu plus en détail sur cette classe dans un prochain paragraphe.

III.4.2.2 La gestion du projet et de l'interface

Les composants ne peuvent exister seuls, sans être rattachés à un projet. Pour pouvoir communiquer avec d'autres composants, il faut qu'il existe au moins un objet qui mémorise la liste des composants existant dans le projet.

D'autre part, la partie interface utilisateur doit permettre d'accéder à n'importe quel composant de la base du projet et de le manipuler grâce aux commandes de l'interface.

La classe *ProjetComposant* sert ces objectifs. Chaque instance de cette classe est en fait une application complète implémentant notre système. Elle contient alors la base du projet (*baseProjet*), qui est l'ensemble des composants existant à un moment donné dans le projet, et des attributs permettant de réaliser et de manipuler l'interface utilisateur.

Les méthodes existant dans cette classe permettent de manipuler la base du projet, en ajoutant ou en supprimant des composants de la base; d'autres méthodes implémentent directement les commandes de l'utilisateur. Il existe également des méthodes permettant de modifier directement l'image graphique du projet dans la fenêtre d'édition (rafraichissement du dessin, écho graphique de déplacement de composant, ...). Enfin il existe une méthode qui permet de tester explicitement si deux composants s'intersectent.

L'interface est implémentée suivant le modèle "Model-View_Controller" disponible et préconisé dans le système Smalltalk [Krasner & Pope 88, Preux 87]. Nous décrivons plus en détail la classe ProjetComposant dans un paragraphe qui suit.

III.4.2.3 Opérateurs de manipulation d'ensembles

Nous avons vu que notre modèle manipule d'une part ce que nous avons appelé les "bords" de composant, c'est à dire les intervalles réels et fermés représentant l'ensemble des valeurs existant entre les deux côtés horizontaux ou verticaux d'un composant. D'autre part, pour déterminer le voisinage d'un composant ou le modifier, nous faisons un usage intensif d'intersections de bords de composants et d'opérations ensemblistes sur ces intersections (des unions d'intersections de bords de composant, des intersections et des différences entre de tels ensembles ...). Comme il n'existe aucune classe dans l'environnement Smalltalk permettant de traiter de tels objets, nous avons dû créer une classe spécialisée qui permet de faire les traitements ensemblistes sur des intervalles fermés. Cette classe est appelée *UniondIntervalles*.

La classe UniondIntervalles permet en fait de réaliser les opérations logiques concernant les intersections D de composant, ainsi que de construire des ensembles par récurrence, au cours de l'exécution des algorithmes de gestion de voisinage (création de voisinage, apparition de voisins, disparition de voisins)

Si nous reprenons par exemple l'équation **E1** :

$$\forall n > 1,\ I_D\left(v_n, C\right) \not\subset \bigcup_{1 \le k < n} \left(I_D\left(v_k, C\right)\right)$$

la partie droite se réécrit simplement en : $\bigcup_{1 \leq k \leq n} (b_{\tilde{D}}(v_k) \cap b_{\tilde{D}}(C))$. Le calcul sera réalisé au moyen d'une série d'opérations d'intersection entre bords de composants (qui sont des intervalles), et d'un calcul d'union construite par récurrence entre tous ces intervalles.

Il faut donc pouvoir manipuler à la fois les intervalles et les unions d'intervalles pour réaliser la gestion et la manipulation de voisinage. Les intervalles sont implémentés comme des instances de la classe *Point* de SMALLTALK. Les coordonnées de ces points contiennent les valeurs extrêmes de l'intervalle représenté.

Nous avons par contre complétement défini et implanté une nouvelle classe *UniondInter-valles* pour gérer les ensembles d'intervalle. Notons d'ailleurs qu'il ne faut pas confondre les instances de cette classe avec un ensemble ordinaire d'intervalles. Les méthodes d'instances implantées dans cette classe concernent les opérations logiques ou de calculs élémentaires nécessaires :

1. opérateurs d'union : avec un intervalle ou une autre union

2. opérateurs d'intersection : avec un intervalle ou une autre union

3. différence logique : d'un intervalle ou d'une union

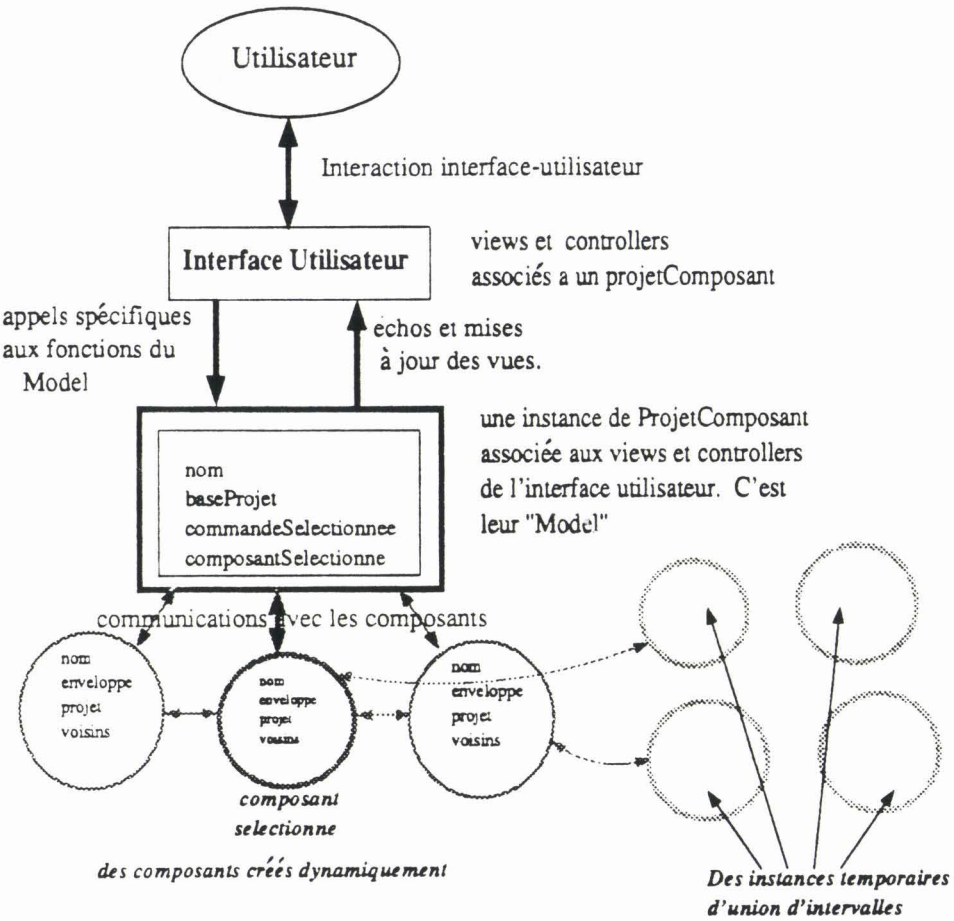
4. prédicats logiques : contenir un intervalle ou une union

5. prédicats de calcul : intervalle d'une union contenant un intervalle donne', intervalles d'une union intersectant un intervalle donné ...

Tous ces opérateurs et ces prédicats sont nécessaires et suffisants pour implémenter les algorithmes de gestion de voisinage décrits précédemment.

III.4.2.4 Schéma de l'interface en cours de fonctionnement

L'utilisateur crée un nouveau projet ainsi que son interface en envoyant le message "*new interface*" à la classe ProjetComposant. A un instant donné, en supposant qu'il y ait déjà *N* composants créés dans ce projet, la structure logicielle du programme peut être schématisée de la manière suivante :



Toutes les instances de composants ou d'union d'intervalles sont créées dynamiquement. Lorsque l'utilisateur crée un nouveau projet, il crée une instance de la classe ProjetComposant, ainsi que l'interface associée selon le paradigme Model-View-Controller.

L'interface est alors opérationnelle pour créer de nouveaux composants et les manipuler. Les commandes sont interprétées par l'interface et sont directement transmises au composant sélectionné. Ce composant connaît ses voisins ou les détermine dans certains cas (création, déportation, déplacement, ...). Chaque fois qu'une action lui est demandée (transmise par l'interface), il agit seul ou en collaboration avec ses agents pertinents (voisins) grâce à des messages spécialisés. S'il a besoin de déterminer de nouveaux agents pertinents, il s'adressera au projet dont il est issu pour "converser" avec les autres composants existant dans le projet.

Enfin les instances d'UniondIntervalles sont uniquement créées temporairement quand elles sont nécessaires, c'est à dire pendant l'exécution des algorithmes de gestion de voisinage.

Il est important de rappeler que l'utilisateur interagit presque directement avec le composant sélectionné. Les composants communiquent entre eux et s'identifient grâce à leur voisinage. Le projet dans lequel est créé un composant permet de gérer l'interface utilisateur (View et

Controller du paradigme MVC de Smalltalk) et contient la base du projet (c'est à dire les composants couramment existant dans le projet).

Il faut donc retenir que ce système, en cours d'utilisation, ne fait que fournir une interface à l'utilisateur pour interagir directement avec des agents indépendants. Le fonctionnement est donc piloté par les actions demandées aux composants, sauf pour la création de nouveaux composants.

III.4.3 Description des classes

Pour décrire les trois classes qui composent notre implémentation, nous ne rentrerons pas dans les détails de l'implémentation elle-même. Une description approfondie du modèle ayant été déjà faite auparavant, il ne nous semble pas nécessaire de décrire exhaustivement l'implémentation. Quand il le faudra nous expliciterons d'une manière détaillée certaines particularités (c'est le cas de la classe *UniondIntervalles*.

Chaque classe est décrite d'une manière assez conventionnelle en Smalltalk :

- Présentation de la structure de l'objet : son nom de classe, sa superclasse, ses attributs, les variables de classes (attributs de la classe), et enfin les variables partagées.

- On explicite ensuite la signification de chacun de ces attributs de classe et d'instance et leur utilité.

- Nous décrivons ensuite les méthodes de classe et les méthodes d'instance. Le sélecteur de chaque méthode est présenté avec une courte description de ce que fait la méthode associée. Par ailleurs le code complet de l'implémentation est disponible en annexe.

La description de la classe *UniondINtervalles* est un peu différente car les instances de cette classe sont issues d'un modèle théorique de représentation des zones visibles, libérées et masquées que l'on utilise dans les algorithmes de gestion de voisinage. Nous avons donc décrit précisément et assez exhaustivement cette classe.

III.4.4 La classe UniondIntervalles

Comme nous l'avons dit précédemment, la création de cette classe était nécessaire pour implémenter les algorithmes de gestion du voisinage d'un composant. Elle sert surtout à effectuer les calcul d'union et d'intersection qui permettent d'obtenir les ensembles M_n, A , et *zoneLiberee* (Cf III.3.7.5. et III.3.7.6.).

Notations : la classe *UniondIntervalles* contenant et permettant de manipuler des intervalles fermés de R, nous précisons ici quelques notations que nous avons adoptées.

Soit I un intervalle fermé de l'ensemble des réels R : I = [a, b]. Nous désignerons les bornes inférieures a et supérieures b de I par inf(I) et sup(I) respectivement.

Nous désignerons le plus grand de deux réels a et b par max(a, b); nous désignerons le plus petit de deux réels a et b par min(a, b).

Un intervalle I est inclus dans un intervalle J si inf(I) ≥ inf(J) et sup(I) ≤ sup(J).

Nous décrivons ici l'implémentation de la classe **UniondIntervalles**.

III.4.4.1 Définition et description des instances d'uniondIntervalles

Définition : Une uniondIntervalles est une collection (un groupe) d'intervalles fermés et disjoints de la droite réelle R, ordonnée par ordre croissant de position des intervalles sur cette droite.

Soit u une union d'intervalles. Si nous appelons les intervalles de u : I1, I2, ... In, les notations (I1, I2, ... In) et I1 U I2 U ...U In désignerons l'un et l'autre l'uniondIntervalles u. Tous les intervalles de u étant disjoints, les deux notations sont équivalentes et ne peuvent prêter à confusion.

Exemple :

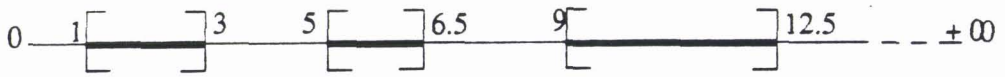


Fig III.1 un exemple d'union d'intervalles

Sur cette figure est représentée une union d'intervalles u, constituée des intervalles suivants : [1, 3], [5, 6.5] , [9, 12.5]. Nous pourrons également désigner u de la manière suivante : u = [1, 3] U [5, 6.5] U [9, 12.5], ou encore u = ([1, 3], [5, 6.5] , [9, 12.5]).

Cette représentation est très adaptée à l'idée de visibilité d'un composant telle que nous l'avons définie, de zone masquée d'un composant. La figure suivante va nous permettre d'illustrer l'utilisation et l'intérêt des uniondIntervalles.

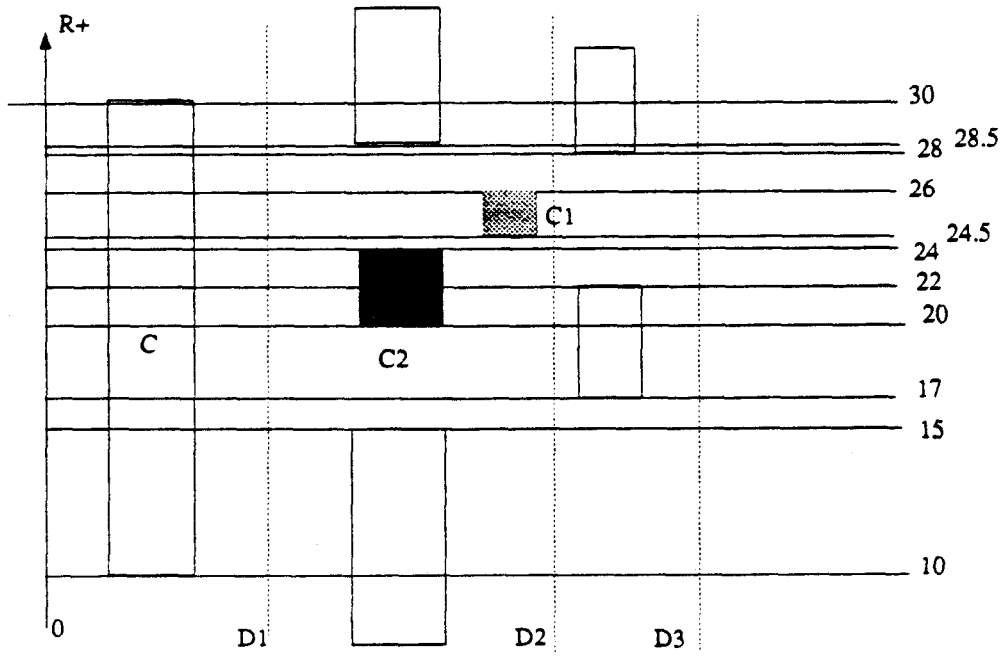


Fig III.2

Nous pouvons représenter les *zoneMasquées* à C dans chacune des zones partagées par les droite D1, D2 , D3 :

1. A gauche de la droite D1 : la *zoneMasquée* = {}..

2. entre les droites D1 et D2, le composant C2 étant présent, et le composant C1 n'existant pas :

   zoneMasquée = [10, 15] U [20, 24] U [28.5, 30].

Si ensuite le composant C1 est rajouté, on aura :

   zoneMasquée = [10, 15] U [20, 24] U [24.5, 26] U [28.5, 30].

Si ensuite le composant C2 est détruit, on aura :

   zoneMasquée = [10, 15] U [24.5, 26] U [28.5, 30].

3. Aprés la droite D3, le composant C1 n'existant pas :

   zoneMasquée = [10, 15] U [17, 24] U [28, 30].

Les composants situés à droite de C et représentés sur cette figure font partie de $I_{H+}(C)$. Ils sont donc rangés par ordre croissant des distances $d_{H+}(C)$. Quand le composant C va déterminer l'ensemble de ses voisins H+, il va calculer par récurrence l'union des M_i telle que nous l'avons définie dans les algorithmes de gestion de voisinage. Cette union correspond à la *zoneMasquée* , calculée successivement à gauche de D1, entre D1 et D2, entre D2 et D3, puis enfin à droite de D3.

On remarque d'autre part que l'ajout du composant C1 rajoute un élément de plus dans la *zoneMasquée* (l'intervalle [24, 26.5]) et que la destruction de C2 implique pour la zone-Masquée le remplacement des intervalles [20, 24] et [24.5, 26] par un seul intervalle [24.5, 26]. La classe UniondIntervalles doit permettre de gérer ce genre de transformation. Mais dans tous les cas, les ensembles dont nous avons besoin pour réaliser les algorithmes de gestion de voisinage peuvent être représentés par une uniondIntervalles.

La classe UniondIntervalles semble donc bien adaptée à la représentation des ensembles manipulés dans les algorithmes de gestion de voisinage, à condition de définir correctement les opérateurs nécessaires à la manipulation de ses instances.

Nous avons implémenté trois types d'opérateurs dans la classe uniondIntervalles, opérateurs dont la nécessité est justifiée par le fonctionnement des algorithmes de gestion de voisinage : ce sont les opérateurs d'union, d'intersection, et de différence entre uniondIntervalles. Ces opérateurs sont bien sûr implémentés sous forme de méthodes d'instance de la classe UniondIntervalles, nous les décrivons dans les paragraphes qui suivent.

Après avoir situé dans la hiérarchie Smalltalk la classe *UniondIntervalles*, nous introduirons les relations logiques d'appartenance et d'inclusion, puis ensuite nous décrirons les opérateurs de manipulation d'uniondIntervalles.

III.4.4.2 position de UniondIntervalles dans la hiérarchie Smalltalk

La position de la classe *UniondIntervalles* dans la hiérarchie de classes du système Smalltalk [Goldberg & Robson 83, SMALLTALK/V 87]est représenté dans la figure suivante :

En SMALLTALK-80 :

Object
  Collection
    SequenceableCollection
      OrderedCollection
        UniondIntervalles

En SMALLTLK/V :

Object
  Collection
    IndexedCollection
      OrderedCollection
        UniondIntervalles

Les objets de la classe *Collection* représentent des groupes d'objets quelconques appelés *éléments*. Par exemple les tableaux sont des collections d'objets de même nature.

Les collections peuvent être ordonnées ou non. Par exemple les chaînes de caractères représentées par la classe *String* sont ordonnées; par contre les instances de la classe *Dictionary* ne le sont pas, de même pour les ensembles (instances de *Set*). La classe *SequenceableCollection* (Smalltalk-80) ou encore *IndexedCollection* (Smalltalk/V) est la racine de tous les classes de collections ordonnées. Les éléments des collections instances d'une sous-classe de *SequenceableCollection* peuvent être accédés de manière externe à l'aide d'indices.

Certaines sous-classes de *SequenceableCollection* représentent des objets de taille fixe (comme les tableaux et les chaînes de caractères), mais aussi des objets de taille variable. C'est la classe *OrderedCollection* qui permet de manipuler ces derniers objets. Dans ces classes d'objets, c'est la séquence et la manière par laquelle les éléments sont ajoutés ou retranchés de l'instance qui détermine l'ordre des éléments de l'instance. La classe *OrderedCollection* pourrait par exemple être très simplement spécialisée pour donner une sous-classe *Pile* ou *File.*

La Classe *UniondIntervalles* a été implémentée comme sous-classe de *OrderedCollection.* D'une part, les ensembles construits par récurrence par les algorithmes de gestion du voisinage manipulent des unionsdIntervalles dont le nombre d'éléments varie au cours du déroulement de l'algorithme. D'autre part, pour pouvoir manipuler facilement les intervalles des uniondIntervalles, en particulier pour effectuer des opérations d'union, d'intersection, de différence, entre uniondIntervalles, il est important que tous ces intervalles soient ordonnés comme ils le sont sur la droite réelle. Pour toutes ces raisons nous avons choisi d'implémenter la classe *UniondIntervalles* comme sous-classe de la classe *OrderedCollection.* Ce n'est pas une spécialisation de *OrderedCollection*, mais plutôt une classe qui utilise les propriétés de sa sur-classe pour ses propres besoins.

III.4.4.3 Prédicats logiques

Nous dirons qu'un intervalle I *appartient* à une union u = (I1, I2, ..., In) s'il existe un indice k et un seul de l'intervalle entier [1, n] tel que I = I_k.

Nous dirons q'un intervalle I est *inclus* dans une union u = (I1, I2, ..., In) s'il existe un indice k et un seul de l'intervalle entier [1, n] tel que I est inclus dans I_k, c'est à dire si inf(I) ≥ inf(I_k) et sup(I) ≤ sup(I_k).

La méthode d'instance de *UniondIntervalles* implémentant le test d'inclusion d'un intervalle dans une uniondIntervalles a pour sélecteur :

```
contient : unInt
"inclusion - renvoie true si l'union receptrice du message
 contient unInt, et false sinon"
```

Nous dirons qu'une union u = (I1, I2, ..., In) est incluse dans une union v = (J1, J2, ..., Jm) si tous les intervalles de u sont inclus dans l'uniondIntervalles v, c'est-à-dire si pour tout intervalle I_k de u il existe un indice l de l'intervalle entier [1, m] tel que I_k est inclus dans J_l.

La méthode d'instance de *UniondIntervalles* implémentant le test d'inclusion d'une uniond-Intervalles dans une autre a pour sélecteur:

```
contientUnion : uneUnion
"inclusion - renvoie true si l'union receptrice du message
 contient une Union, false sinon."
```

utilitaires associés aux relations logiques

Pour simplifier le code des méthodes associées aux relations logiques citées précédemment et aux opérateurs logiques que nous verrons ensuite, nous avons implémenté deux méthodes privées (uniquement utilisées par les instances de UniondIntervalles).

Le contenant d'un Intervalle : c'est l'intervalle unique d'une uniondIntervalles qui contient un intervalle donné. le sélecteur de message associé à cette méthode est :

```
contenantDe : unInt
"prive - renvoie l'unique intervalle de l'union receptrice
du message qui contient unInt."
```

Les intersectants d'un Intervalle : ce sont tous les intervalles d'une uniondIntervalles dont l'intersection avec un intervalle donné n'est pas vide. Le sélecteur de message associé à cette méthode est :

```
intersectantsDe : unInt
"prive - renvoie l'uniondIntervalles constituée de tous
 les intervalles qui intersectent unInt."
```

III.4.4.4 Opérateurs d'union

Ces opérateurs permettent de réaliser l'union de deux uniondIntervalles ou d'une uniond-Intervalle avec un intervalle.

Union entre deux intervalles fermés

Rappelons tout d'abord le résultat d'une union entre deux intervalles fermés I et J quelconques de R :

- Si I est inclus dans J , alors I U J = J. Par exemple, [1, 5] U [2, 3] = [1, 5].
- Si I et J sont disjoints, alors I U J = l'union d'intervalles contenant les intervalles I et J. Par exemple, [1, 2] U [3, 5] = ([1,2], [3,5]).
- Si I et J s'intersectent (sup(I) >= inf(J) ou inf(I) <= sup(J)), alors I U J = [min(inf(I), inf(J)), max(sup(I), sup(J))]. Par exemple, [1, 3] U [2, 5] = [1, 5].

D'autre part, l'opération d'union entre deux intervalles est commutative : pour tous I et J intervalles fermés de R, on a I U J = J U I.

Opérateur d'union entre une uniondIntervalles et un intervalle.

Définition : Soit u une union d'Intervalles, et I un intervalle fermé de R, l'union de u et de I est la plus petite union contenant u et I.

l'uniondIntervalles v = u U I est telle que I est inclus dans v et u et inclus dans v.

Exemples :

- ([1, 3], [4, 5]) U [2, 3] = ([1, 3], [4, 5]).
- ([1, 3], [4, 5]) U [6, 7] = ([1, 3], [4, 5], [6, 7]).
- ([1, 3], [4, 5]) U [2, 4.5] = ([1, 5]).
- ([1, 2], [4, 5], [6, 7]) U [1, 3] = ([1, 3], [4, 5], [6, 7]).
- ([1, 2], [4, 5], [6, 7]) U [1.5, 5.5] = ([1, 5.5], [6, 7]).

Méthode de construction de u U I : l'opération d'union entre u et I a été implémentée de la manière suivante :

1. Si I est inclus dans u, alors u U I = u.
2. Si aucun intervalle de u n'intersecte I, alors l'opération u U I consiste à rajouter I dans u. u U I = (I1, I2, ..., $I_{n, I}$).
3. Si I intersecte un ou plusieurs intervalles de u, alors l'opération u U I consiste à déterminer l'intervalle englobant à la fois I ainsi que tous les intervalles de u intersectés par I, puis à remplacer dans u les intervalles intersectés par I par ce seul intervalle englobant.

La méthode d'instance de *UniondIntervalles* implémentant l'union entre une est :

```
++ unInt
" union - renvoie l'union entre l'union receptrice du message et l'intervalle unInt.
 Cette opération consiste a determiner la plus petite uniondIntervalles
contenant a la fois l'union receptrice et l'intervalle unInt."
```

Intuitivement, l'opération d'union revient à prendre le résultat de la superposition entre l'union d'intervalles et l'intervalle.

Opérateur d'union entre deux unions d'intervalles

C'est la généralisation de l'opération précédente à deux unions d'intervalles.

Définition : étant données deux unions d'intervalles u1 et u2, u1 U u2 est construit par récurrence de la manière suivante :

Exemples :

- ([1, 2], [6, 8]) U ([4, 5], [10, 12]) = ([1, 2], [4, 5], [6, 8], [10, 12]).

- ([1, 3], [7, 16], [19, 21]) U ([5, 9], [12, 15], [18, 23]) = ([1, 3], [5, 16], [18, 23]).

Méthode de construction de u U v : l'opération d'union entre deux uniondIntervalles a étée implémentée par récurrence de la manière suivante :

soit u2 = j1 U j2 U j3 U ... U jn. On définit V0 = u1, V1 = V0 U j1.
Alors, pour tout entier k, k > 1 et k <= n, on a Vk = V_{k-1} U jk.
Et Vn = (V_{n-1} U jn)= u1 U u2.

Le sélecteur de message associé à la méthode de calcul de l'union est le suivant :

```
+ uneUnion
"union - realise l'opération d'union entre deux uniondIntervalles.
Renvoie une uniondIntervalles contenant le resultat."
```

III.4.4.5 Opérateurs d'intersection

Ces opérateurs réalisent le calcul de l'intersection entre une uniondIntervalles et un intervalle, et entre deux uniondIntervalles

Intersection de deux intervalles fermés

L'intersection de deux intervalles fermés I et J de R est un intervalle fermé K qui est contenu dans les deux intervalles I et J. On a les résultats suivants :

- Si I est inclus dans J, alors I ⋂ J = I. Par exemple, [1, 5] ⋂ [2, 3] = [2, 3].

- Si I et J sont disjoints, alors I ⋂ J = ∅. Par exemple, [1, 2] ⋂ [5, 3] = ∅.

- si I et J s'intersectent, alors I ⋂ J = [max(inf(I), inf(J)), min(sup(I),sup(J))]. Par exemple, [1, 3] ⋂ [2, 5] = [2, 3].

- L'intersection de deux intervalles est commutative. I ⋂ J = J ⋂ I.

Intersection entre une uniondIntervalles et un intervalle

Définition : l'intersection entre une uniondIntervalles u et un intervalle I est la plus grande uniondIntervalles dont les éléments sont à la fois inclus dans u et dans I.

L'intersection v = u ⋂ I = (v1, v2, ..., vn) est telle que v est incluse dans u et pour tous les entiers k appartenant à l'intervalle entier [1, n] vk est inclus dans I.

Exemples :

- ([1, 3], [4, 5]) ⋂ [2, 3] = ([2, 3]).
- ([1, 3], [4, 5]) ⋂ [6, 7] = ( ).
- ([1, 3], [4, 5]) ⋂ [2, 4.5] = ([2, 3], [4, 4.5]).
- ([1, 2], [4, 5], [6, 7]) ⋂ [1, 3] = ([1, 2]).
- ([1, 2], [4, 5], [6, 7]) ⋂ [0, 5.5] = ([1, 2], [4, 5]).

Méthode de construction : la méthode d'instance implémentant l'intersection entre une uniondIntervalles et un intervalle est basée sur le mécanisme suivant :

soit u = (u1, u2, ..., un) une uniondIntervalles et I un intervalle fermé. Soit v = (v1, v2, ..., vm) l'intersection entre u et I. v est incluse dans u et I, donc

$$\forall k \in [1,m], \exists! j \in [1,n] \textbf{ tq } v_k \subset u_j \textbf{ et } v_k \subset I$$

Il existe donc j tel que $v_k \subset (u_j \bigcap I)$. Les éléments de u étant disjoints entre eux, on a :$\forall i$ et $j \in [1,n].(u_i \cap I) \bigcap (u_j \cap I) = \emptyset$,et donc l'uniondIntervalles $((u_1 \cap I),(u_2 \cap I),....(u_n \cap I))$ contient forcément v puisque tout élément de v est inclus dans un $(u_i \cap I)$. D'autre part, $((u_1 \cap I).(u_2 \cap I)....,(u_n \cap I))$ est incluse dans v puisque tout ses éléments sont à la fois contenus dans u et dans I. Donc v = $((u_1 \cap I),(u_2 \cap I),...,(u_n \cap I))$.

D'où pour calculer $(u \bigcap I)$, on calculera séquentiellement les $(u_i \bigcap I)$ et on les collationnera dans une uniondIntervalles initialement vide.

Le sélecteur de message associé à la méthode implémentant l'intersection dune uniondIntervalles avec un intervalle est :

```
** unInt
"intersection - renvoie l'intersection entre l'union receptrice
du message et unInt.
procede par determination des intersectants de unInt,
puis calcule l'intersection effective entre
ces intersectants et unInt."
```

Opérateur d'intersection de deux uniondIntervalles

C'est l'extension de l'opération précédente à deux uniondIntervalles.

Définition : Etant données deux uniondIntervalles u et v, l'intersection de ces deux uniondIntervalles est la plus grande uniondIntervalle qui est incluse dans u et dans v.

Exemples :

- ([1, 2], [6, 8]) ⋂ ([4, 5], [10, 12]) = ().
- ([1, 3], [7, 16], [19, 21]) ⋂ ([5, 9], [12, 15], [18, 23]) = ([7, 9], [12, 15], [19, 21]).

Méthode de construction de u ⋂ v : on effectue une itération sur les éléments de v pour obtenir $(u \cap v_1), (u \cap v_2), ..., (u \cap v_m)$. Toutes ces nouvelles uniondIntervalles sont disjointes entre elles, puisques les éléments de v sont disjoints entre eux. En collationnant les uniondIntervalles, on obtient donc u ⋂ v.

L'opérateur implémentant l'opération d'intersection a pour sélecteur de message :

```
* uneUnion
"intersection - renvoie l'uniondIntervalles representant l'intersection
entre l'union receptrice et uneUnion. procede par iteration
sur uneUnion."
```

III.4.4.6 Opérateurs de différence

Ces opérateurs réalisent la différence entre une uniondIntervalles et un intervalle, ainsi qu'entre deux uniondIntervalles. Il s'agit en fait du complémentaire d'un intervalle dans une uniondIntervalles ou d'une uniondIntervalles dans une autre uniondIntervalles. Nous l'avons appelé différence pour des raisons de simplicité d'écriture et pour conserver une notation identique à celle des algorithmes de gestion de voisinage.

Différence entre deux intervalles fermés

Nous définissons ici ce que nous appelons la différence entre deux intervalles fermés.

- Si I est inclus dans J, alors J — I = ([inf(J), inf(I)], [sup(I), sup(J)]. Par exemple, [2, 5] — [3, 4] = ([2, 3], [4, 5]).
- Si J est inclus dans I, alors J — I = ().Par exemple, [3, 4] — [2, 5] = ().
- Si I et J sont disjoints, alors J — I = (). Par exemple, [1, 2] — [3, 4] = ().
- Si I et J s'intersectent, alors deux cas peuvent se présenter : 1– si sup(I) >= inf(J), alors J — I = [sup(I), sup(J)]; 2 — si inf(I) <= sup (J), alors J — I = [inf(J), inf(I)].

remarquons que la différence de deux intervalles n'est pas commutative.

Opérateur de différence entre une uniondIntervalles et un intervalle

Définition : soit une uniondIntervalle u et un intervalle I. L'ensemble u — I est la plus grande uniondIntervalles qui soit incluse dans u mais disjointe de I.

La différence v = u — I est telle que v est inclus dans u et $v \cap I = \emptyset$.

Exemples :

- ([1, 3], [4, 5]) — [2, 3] = ([1, 2], [4, 5]).
- ([1, 3], [4, 5]) — [6, 7] = ([1, 3], [4, 5]).
- ([1, 3], [4, 5]) — [2, 4.5] = ([1, 2], [4.5, 5]).
- ([1, 2], [4, 5], [6, 7]) — [1, 3] = ([4, 5], [6, 7]).
- ([1, 2], [4, 5], [6, 7]) — [1.5, 5.5] = ([1, 1.5], [6, 7]).

Méthode de construction de u — I : on procède par itération sur u.

Soit u = (u1, u2, ..., un). Pour tout i de [1, n], on a (ui — I) qui est inclus dans ui. Donc pour tous i et j distincts, on a (ui — I) et (uj — I) qui sont disjoints, car ui et uj sont disjoints. Donc ((u1 — I), (u2 — I), ..., (un —I)) est une uniondIntervalles incluse dans u — I. D'autre part, tout élément vj de v appartient à u mais ne peut être inclus dans I. Donc $v_j \cap I = \emptyset$, et il existe un entier k compris entre 1 et n tel que vj est inclus dans uk — I. On a donc v = u - I = ((u1 — I), (u2 — I), ..., (un —I)). On obtiendra donc v en collationnant dans une uniondIntervalles initialement vide les ui — I.

L'opérateur implémentant cette opération a pour sélecteur :

```
\\ unInt
"difference - realise l'operation de difference entre l'union
receptrice et unInt.
procede par itération sur l'union receptrice et renvoie
l'uniondIntervalles resultante."
```

Opérateur de différence entre deux uniondIntervalles

C'est l'extension de l'opération précédente à une uniondIntervalles.

Définition : Etant données deux uniondIntervalles u et v, la différence u — v est la plus grande uniondIntervalles incluse dans u et disjointe de v.

Exemples :

- ([1, 2], [6, 8]) — ([1, 2], [6, 8]) = ().

- ([1, 2], [6, 8]) — ([0, 8], [10, 12]) = ().
  ([0, 8], [10, 12]) — ([1, 2], [6, 8]) = ([0,1], [2,6], [10, 12]).

- ([1, 2], [6, 8]) — ([4, 5], [10, 12]) = ([1, 2], [6, 8]).

- ([1, 3], [7, 16], [19, 21]) — ([5, 9], [12, 15], [18, 23]) = ([1, 3], [9, 12], [15, 16]).

Méthode de construction de u — v : on procede par récurrence sur v.

Soit v = (v1, v2, ..., vm). On calcule tout d'abord s1 = u — v1. s1 est une uniondIntervalles qui est incluse dans u et qui ne contient pas v1.

Pour tout i plus grand que 1 et plus petit que m, on calculera successivement $s_i = s_{i-1}$ — v_i. si est une uniondIntervalles incluse dans u et qui ne contient pas v1, v2, ..., vi. Comme v1, v2, ..., vi sont des intervalles disjoints entre eux, (v1, v2, ..., vi) est une uniondIntervalles et S_i ne contient pas cette union.

On calcule finalement $s_m = s_{m-1}$ — v_m. s_m est incluse dans u et ne contient pas (v1, v2, ..., vm). On a donc sm = u — v. On obtient donc u — v en calculant successivement s1 = u \\ v1, s2 = s1 \\ v2, ..., sm = s_{m-1} \\ vm= u — v .

Il faut noter de plus que cette opération n'est pas commutative comme le montre le deuxième exemple précédent.

L'opérateur implémentant cette opération a pour sélecteur de message :

```
\ uneUnion
"difference - calcule la difference entre l'union receptrice
du message et uneUnion. renvoie l'union resultante.
procede par recurrence sur uneUnion."
```

Problème de continuité aux bornes des intervalles fermés : On remarquera que lors des opérations de différence entre uniondIntervalles et intervalles le résultat est toujours une uniondIntervalles. Or cela ne devrait pas être toujours le cas. En effet, en otant un intervalle fermé d'une uniondIntervalles, on devrait obtenir des intervalles ouverts, car on devrait ôter

également les valeurs extrêmes de l'intervalle retranché. Mais nous avons considéré que la différence de deux intervalles fermés donnait toujours des intervalles fermés. Il y a là un problème dans notre définition de la différence, mais il n'est pas néfaste au fonctionnement des algorithmes de gestion de voisinage.

En effet, un composant ne peut pas physiquement être masqué ou visible sur une zone représentée par un intervalle à un seul point. C'est la portion de droite comprise entre les deux bornes d'un intervalle qui détermine si il y a masquage ou visibilité, et les bornes des intervalles considérés ne sont pas significatives. C'est la raison pour laquelle nous avons défini les opérateurs de différence de manière à ce qu'ils renvoient toujours des intervalles ou des unionsIntervales fermés.

III.4.4.7 Conclusion

La classe *UnionsIntervalles* implémente les objets dont nous avons besoin pour implémenter les algorithmes de gestion de voisinage. Les méthodes d'instances de cette classe qui sont : les prédicats logiques d'inclusion, les opérateurs d'union, les opérateurs d'intersection, les opérateurs de différence permettent de manipuler de façon efficace et naturelle les zones masquées et visibles que l'on construit dans les algorithmes de gestion de voisinage.

De plus, la représentation adoptée ici s'adapte naturellement à l'idée de visibilité d'un composant pour une orientation donnée. C'est pourquoi l'implémentation des algorithmes de gestion de voisinage font un usage intensif de cette classe.

La classe a été implantée très simplement dans le système Smalltalk et utilise principalement la classe *Point* pour représenter des intervalles fermés, et la classe *OrderedCollection* pour disposer d'une part d'un nombre variable d'éléments dans les unions et d'autre part pour pouvoir toujours ordonner correctement les intervalles obtenus sur la droite réelle. Une implémentation ADA nécessiterait comme travail essentiel de redéfinir le type abstrait OrderedCollection.

III.4.5 La classe ProjetComposant

Cette classe n'a été implantée que dans l'environnement de Smalltalk/V. Certains noms de classe sont donc différents de Smalltalk-80; ces différences seront précisées si cela est nécessaire.

III.4.5.1 définition et description des instances de la classe ProjetComposant

Nom de classe : ProjetComposant

Superclasse : Objet

Variables d'Instances :

nom
baseProjet
composantSelectionne
commandeSelectionnee
graphPane
flotDeRenvois
pasDeGrille
image

Variables de classe : PoliceDessin

Variables partagées : FunctionKeys

La classe *ProjetComposant* implémente la partie applicative de notre interface. C'est le "*modèle*" de l'application complète, selon le paradigme "Model-View-Controller" de Smalltalk. Les instances de cette classe gèrent donc à la fois l'interface du projet et la base du projet. La base du projet contient tous les composants qui constituent le projet à un instant donné. La superclasse de *ProjetComposant* est *Object* car il n'existe aucune classe de Smalltalk ayant des points communs avec cette classe et donc aucune classe du système n'est réutilisable à des fins de raffinement pour obtenir *ProjetComposant*. Il n'existe qu'une seule variable de classe *PoliceDessin* qui mémorise la police de caractères couramment utilisée dans cette implémentation de *ProjetComposant*. Les variables partagées (pool dictionnaries en anglais) sont limitées aux valeurs d'assignation des codes de touche du clavier et de la souris.

La signification des variables d'instances est la suivante :

Les variables d'instances relative au projet seulement :

nom : c'est le nom du projet sur lequel travaille l'utilisateur. Il sera affiché dans la zone *étiquette* de l'interface. C'est une instance de la classe *String*.

baseProjet : c'est la base de projet du projet instancié. C'est une instance de la classe *Dictionnary*.

Les variables d'instances relative à l'interface seulement :

composantSelectionne : c'est le composant couramment sélectionné par l'utilisateur et avec lequel il va converser presque directement au travers de l'interface. C'est une instance de la classe *ComposantRectangle*.

commandeSelectionnee :c'est la commande en cours d'exécution. C'est une instance de String.

graphPane : c'est la vue graphique dans laquelle est visualisée le schéma en cours de développement. C'est une instance de la classe *GraphPane* (dont l'équivalent est *FormView* en Smalltalk-80).

flotDeRenvois : c'est le *dispatcher* (appelé *controller* en Smalltalk-80) associé à la vue décrivant textuellement le composant sélectionné (la fenêtre de messages). C'est une instance de *TextEditor*.

pasDeGrille : pour disposer de points de repères de positionnement, l'interface peut visualiser une grille dans la fenêtre graphique d'édition dont l'utilisateur peut modifier le pas. C'est de ce pas dont il s'agit ici. C'est une instance de *Integer*.

image : contient le bitmap où est dessiné en permanence le schéma avant recopie sur l'écran. C'est une instance de la classe *Form*.

Nous allons maintenant décrire rapidement les méthodes implantées sur la classe *ProjetComposant*.

III.4.5.2 Méthodes de Classe et d'instance de ProjetComposant

Il y a deux méthodes de classe implantées dans la classe *ComposantRectangle*, toutes deux réservées pour la création d'une nouvelle instance de *ProjetComposant*:

- la méthode **new** : qui redéfinit le new de la classe *Object* en initialisant la classe *ComposantRectangle*, sa base de projet et le pas de grille.

- la méthode **deNom** : `uneChaine` qui appelle la précédente et assigne le nom du projet à uneChaine.

Les méthodes d'instance de base sur le projet

Ce sont les méthodes qui agissent sur le projet seulement, et qui n'envoient aucun message aux composants. Un nom de catégorie adéquat pour ces méthodes pourrait être par exemple "`projet`" :

- La méthode **baseDuProjet** renvoie le dictionnaire contenant le projet. **baseDuProjet** : `dictionnaire` assigne la base de projet au dictionnaire passé en paramètre. **do** : `unBlock` exécute le bloc Smalltalk passé en paramètre à tous les composants courants du projet.

- La méthode **nom** renvoie le nom du projet, et **nom** : `uneChaine` assigne le nom du projet à uneChaine.

- **ajoutComposant** : `unComposant` et **oterComposant** : `unComposant` ajoutent et retranchent respectivement un composant de la base du projet.

Méthodes d'interface pure

Ce sont des méthodes qui implémentent des fonctionnalités de l'interface qui n'agissent pas sur les composants. Un nom de catégorie adapté à cet ensemble de méthodes est "confortInterface".

- **choisirComposant** permet à l'utilisateur de sélectioner un composant de la base par son nom : la liste des noms de composants existant est affichée et le choix se fait en cliquant sur l'un des noms de la liste.

  coherenceBase vérifie la symétrie des liens de voisinage parmi tous les composants de la base de projet; les incohérences sont affichées en indiquant les composants dont les liaisons sont mauvaises.

  selectionComposant : unPoint permet à l'utilisateur de sélectionner un composant en "clicquant" dessus dans la fenêtre graphique d'édition.

- **grilleEcran** permet à l'utilisateur de modifier interactivement la valeur du pas de grille de la fenêtre de visualisation.

  rafraichirEcran remet à jour (redessine) l'ensemble des vues de l'interface.

Méthodes de fonctionnement de l'interface

Ce sont les méthodes nécessaires au bon fonctionnement de l'interface, pour respecter à la fois le paradigme Model-View-Controller de Smalltalk, et aussi pour assurer la mise à jour correcte de la fenêtre d'édition graphique. Un nom de catégorie adapté serait "mécaniqueInterface".

- **commandes** renvoie un tableau de chaines de caractères désignant les commandes disponibles à l'utilisateur et visualisées dans la zone de menu de l'interface.

  commande : aString est le message standard exigé par le modèle MVC pour déclencher la sélection d'une commande. La zone de menu est en fait une *ListPane* (ListView en Smalltalk-80) et "commande : *unNomDeCommande*" est le message qui est envoyé au modèle (l'instance de *ProjetComposant*) associé au *ListPane*.

- **dessin** : unFrame dessine la totalité du schéma dans la fenêtre de visualisation-édition graphique.

  dessinCreationDe : unComposant trace un nouveau composant qui vient d'être ajouté par l'utilisateur dans la fenêtre de visualisation-édition graphique.

  dessinDestructionDe : unComposant fait de même mais cette fois en effaçant un composant qui vient d'être détruit.

- **drawMove** : position **rectangle** : unRectangle **pen** : unCrayon est une méthode utilitaire utilisée lorsque l'utilisateur déplace ou déporte un composant. Elle trace unRectangle à la position unePosition dans la fenêtre graphique.

  drawRectangle : rect **pen** : crayon trace le rectangle rect dans la fenêtre d'édition graphique.

  interface est la méthode clef de l'interface de *ProjetComposant*. C'est elle qui crée la fenêtre principale et les sous-fenêtres de l'interface grâce aux messages spécifiques de Smalltalk pour la création de fenêtres (addSubPane : , model :, name :, change :, framingBlock), adressés à de nouvelles instances de *ListPane (ListView), GraphPane(FormView), TextPane (TextView)*. Ces messages engendrent automatiquement la création des *dispatchers (controllers)* associés à chacune des vues créées. Quand l'exécution de cette méthode est terminée, l'interface est prête à être utilisée.

  moveEnveloppe : unRectangle **toDirection** : uneDirection permet à l'utilisateur de déplacer l'enveloppe d'un composant suivant uneDirection. Si uneDirection a une valeur indeterminée, le composant sélectionné peut être déplacé dans toutes les directions. Cette méthode implémente également l'écho de ce déplacement dans la fenêtre graphique grâce aux techniques classiques de "rubber-banding".

- **getCoin** : unRectangle permet à l'utilisateur de saisir interactivement et graphiquement une nouvelle enveloppe pour le composant sélectionné en choisissant un nouveau coin bas-droit pour ce composant. La méthode réalise également le tracé de rectangle élastique comme écho de l'interaction.

  getEnveloppe : unComposant permet à l'utilisateur de saisir graphiquement et interactivement une nouvelle enveloppe pour le composant sélectionné. La méthode réutilise la classe *PointDispatcher* pour la saisie interactive de l'enveloppe et son écho.

- **pasDeGrille** : unEntier assigne la valeur de l'attribut pasDeGrille à unEntier.

  renvois remet à jour la vue décrivant le composant couramment sélectionné. Cette méthode est activée chaque fois que le composant sélectionné change, ou quand on redessine la totalité de l'interface.

  verifierEnveloppe : unRectangle **nom** : unNom verifie que l'enveloppe de composant choisie par l'utilisateur ne provoque pas de conflit géométrique avec d'autres composants.

Méthodes d'interface pour les composants

Ces méthodes sont celles qui sont déclenchées par l'utilisateur lorsqu'il choisit une commande. Elles relient presque directement l'utilisateur aux composants. Les contrôles qu'elles effectuent ne concernent que le nom des composants choisis ou les conflits géométriques potentiels entre composants. Ces méthodes sont directement reliées aux symboles renvoyés par la méthode "commandes" que nous avons citée plus haut. Un nom de catégorie adaptée serait "interfaceComposant".

- **creerComposant** permet à l'utilisateur de rajouter un nouveau composant au projet en donnant son nom et son enveloppe. La méthode provoque les vérifications nécessaires après la création du nouveau composant (nom unique et pas de conflit géométrique).

  detruireComposant permet à l'utilisateur de détruire un composant du projet.

- **coinComposant** permet à l'utilisateur de changer l'enveloppe d'un composant en déplacant le coin bas-droit de celui-ci. La méthode déclenche la vérification de conflits géométriques éventuels entre le composant modifié et les autres composants.

  enveloppeComposant permet à l'utilisateur de modifier interactivement l'enveloppe du composant sélectionné.

- **deplacerComposant** permet à l'utilisateur de déplacer un composant dans une direction horizontale ou verticale. C'est le composant qui est déplacé qui interdira à l'utilisateur de recouvrir ses voisins.

  deporterComposant permet à l'utilisateur de déporter le composant selectionné à l'endroit de son choix dans la fenêtre graphique. C'est la commande qui permet la déportation de composant. La méthode lance la vérification sur les conflits géométriques éventuels.

Méthodes d'activation des composants

Ces méthodes réalisent réellement la connexion entre l'interface et le composant sélectionné lors de la réalisation d'une action impliquant le composant. Elles ont la caractéristique d'être quasiment limitées à l'envoi d'un message spécialisé de la classe *ComposantRectangle*. Un nom de catégorie adapté à ces méthodes pourrait être "connexion-Composant".

- **creerComposantRectangle** : unNom **enveloppe** : unRectangle crée un nouveau composantRectangle asocié au receveur. La validité des attributs du nouveau composant (son nom et son enveloppe) est vérifiée par la classe *ComposantRectangle*.

  detruireComposantRectangle : unComposant demande à unComposant de faire le nécessaire pour sa destruction (algorithme d'apparition de voisins) et retire ce composant de la base du projet.

- **deplacerComposantRectangle** : unComposant **direction** : une Direction **sens** : unSens **distance** : uneDistance demande au composant de se déplacer de uneDistance dans la direction et le sens précisés.

  deporterComposantRectangle : unComposant **en** : unPoint demande à unComposant de s'auto-détruire et en recrée un nouveau identique au précédent placé en unPoint. le composant messagé effectuera bien sûr les algorithmes d'apparition de voisins lors de sa destruction et de disparition de voisins lors de sa recréation.

- **modifierTailleDe** : unComposant **nouveauCoin** : unPoint modifie la taille de unComposant en lui demandant de s'auto-détruire, puis en le recréant avec la position de son coin bas-droit modifiée.

  modifierTailleDe : unComposant **par** : unRectangle modifie la taille de unComposant en lui demandant de s'autodétruire et en recréant un autre composant de même nom et d'enveloppe différente.

Particularités d'implémentation de ProjetComposant

Nous avons en général respecté au mieux le paradigme MVC (Model-View-Controller) d'architecture de l'interface utilisateur. La vue principale de l'interface est divisée en trois sous-vues : la zone de menu, la zone de message, la fenêtre interactive graphique d'édition. La zone de menu est implémentée en utilisant le couple View-Controller (ListView, ListController), la zone de message utilise le couple (TextEditor, TextView). La fenêtre graphique est cependant implémentée d'une manière un peu particulière : elle utilise le couple (FormView, FormController), mais nous avons défini et implémenté le comportement interactif graphique des composants (hormis la sélection de composant déjà presque disponible dans le couple (FormView, FormController)) dans la classe *ProjetComposant* elle-même.

Pour respecter strictement le Modèle MVC il aurait fallu implémenter une sous-classe de FormView qui contienne les méthodes spécifiques d'interaction graphique (`dessin : , dessinCreationDe : , dessinDestructionDe : , drawMove : ..., drawRectangle :, moveEnveloppe :, getCoin :, getEnveloppe`). Ce développement tout à fait spécifique ne pouvait se justifier que par la cohésion de l'implémentation dans le système Smalltalk, ce qui n'est pas primordial pour un prototype. D'autre part, cela aurait alourdi significativement le portage dans un environnement dont l'architecture de l'interface est différente de celle de Smalltalk.

Les actions de l'utilisateur se déroulent toujours en deux temps (sauf pour la sélection directe de composant) : on choisit d'abord une commande dans la zone de menu, ensuite le protocole de paramétrisation de la commande a lieu (il s'agit du dialogue interface-usager pour cette commande). Ce dialogue est soit alphanumérique ou semi-graphique (nom de composant, valeur de pas de grille, affichage de la description du composant sélectionné dans la zone de message), soit complétement graphique et à manipulation directe (sélection, création, déplacement, déportation, taille) pour les actions directes sur les composants, soit un mélange des deux (cas de la création de composant : nom de composant et positionnement et forme de l'enveloppe). L'accent a en fait été surtout porté sur l'aspect "manipulation directe" de l'interface.

Il résulte de ces choix que la classe *ProjetComposant* dispose de méthodes strictement liées au modèle (`baseDuProjet, baseDuProjet : , do : , nom, nom :, ajoutComposant : , oterComposant :`) et d'autres méthodes spécialisées dans l'interface ou l'interaction avec les composants. Une implémentation puriste de *ProjetComposant* n'incluerait pas les méthodes d'interface à manipulation directe, mais ferait usage d'une sous-classe spécialisée.

Conclusion

La classe *ProjetComposant* implémente l'interface, le projet et la base de projet sur lequel on travaille. Bien qu'elle respecte le paradigme MVC de Smalltalk, les aspects de manipulation graphique directe de composants ont été implantés comme méthodes et attributs de la classe, alors qu'une implémentation plus respectueuse du modele MVC aurait nécessité la création d'une sous-classe de vue graphique spécialisée. D'où la grosse quantité de code des méthodes d'interface de cette classe.

Ceci implique qu'un portage point à point de cette classe nécessite un environnement graphique multi-fenêtre adoptant les principes du modèle MVC de Smalltalk, ou alors sa reconstruction . C'est un travail important à réaliser pour un portage en ADA. On lui préférera une réécriture de notre interface adaptée a l'environnment ADA utilisée.

III.4.6 La classe ComposantRectangle

C'est la classe la plus importante dans notre implémentation, car elle implémente les agents ainsi que leurs comportements. On y trouvera des méthodes d'instances de moindre importance comme celles de la catégorie "renvoisGéométriques", et d'autres tout à fait essentielles comme celles qui implémentent (ou contiennent) les algorithmes de gestion de voisinage. A cause des options choisies pour l'implémentation des voisinages, et pour respecter le formalisme que nous avons présenté pour la description des voisinages et des algorithmes de gestion de voisinage, il y a un nombre inhabituel de variables et de méthodes de classe.

III.4.6.1 Définition et description des instances de la classe ComposantRectangle

Nom de classe : ComposantRectangle

Superclasse : Objet

Variables d'Instances :
- nom
- enveloppe
- voisins
- projet

Variables de classe :
- Negatif
- Positif
- Horizontal
- Vertical
- Ouest
- Est
- Nord
- Sud
- SortBlocks

Variables partagées :

La classe *ComposantRectangle* implémente les composants (les *agents*) tels que nous les avons décrits dans les paragraphes III.3.3 et III.3.4.. Les composants sont des objects simples du point de vue de leur structure, mais complexes de par leur comportement. Certaines méthodes ne concernent que la géométrie du composant, d'autres implémentent des vérifications topologiques ou encore des calculs et mises à jour de voisinage après une modification. En plus de son contexte local matérialisé par la variable d'instance *voisins*, le composant doit pouvoir s'adresser parfois à l'ensemble des composants existant dans le projet, par exemple pour déterminer de nouveaux voisins; ceci est réalisé grâce à la variable d'instance *projet*, qui associe le composant au projet dont il est issu. Le composant peut alors s'adresser à la base de projet de *projet* pour interroger ou messager des composants qui ne font pas partie de ses voisins à un instant donné.

Pour respecter le formalisme que nous avons choisi, en particulier pour spécifier les orientations, la classe dispose de variables qui sont en fait des constantes initialisées lors de la première utilisation de la classe par une instance de *ProjetComposant*.

L'implémentation des voisins est un peu spécifique du modèle puisque ceux-ci doivent être mémorisés dans l'ordre des distances croissantes par rapport au composant.

Cette classe est en fait l'implémentation d'un type abstrait composant.

Description et signification des variables de classe

Les constantes déterminant les orientations sont les suivantes :

Est : est un entier qui vaut 1. C'est une instance de *Integer*.

Ouest: est un entier qui vaut 2. C'est une instance de *Integer*.

Nord : est un entier qui vaut 3. C'est une instance de *Integer*.

Sud: est un entier qui vaut 4. C'est une instance de *Integer*.

Ces quatres constantes désignent chacune une orientation possible, relativement à notre modèle.

Les variables de classe relatives au calcul d'une orientation sont les suivantes :

Positif : est un entier qui vaut 1. C'est une instance de *Integer*.

Negatif : est un entier qui vaut 2. C'est une instance de *Integer*.

Horizontal : est un entier qui vaut 0. C'est une instance de *Integer*.

Vertical : est un entier qui vaut 2. C'est une instance de *Integer*.

Ce sont les *composantes d'orientation*. En effet, elles permettent de désigner une orientation par combinaison. *Positif* et *Negatif* permettent de différencier le sens d'une orientation, c'est à dire de distinguer le Nord et le Sud, l'Ouest et l'Est. *Horizontal* et *Vertical* permettent

de différencier la direction d'une orientation, c'est-à-dire de distinguer une direction horizontale d'une direction verticale. Les valeurs de ces constantes sont telles qu'elles permettent de calculer naturellement une orientation : l'Est indiquant un sens croissant sur la direction horizontale, on obtient la valeur de Est en ajoutant *Horizontal* et *Positif*. additionner ces valeurs de constantes deux a deux nous donne des valeurs entières comprises entre 1 et 4. On a exactement :

- Est = Horizontal + Positif ,
- Ouest = Horizontal + Negatif ,
- Nord = Vertical + Positif ,
- Sud = Vertical + Negatif

ce mécanisme de calcul est intéressant lors des algorithmes de voisinage pour rendre indépendante l'implémentation des voisins de leur utilisation.

D'autre part ce mécanisme évite d'implanter des tables de correspondance entre des directions opposées ou des sens opposés. En effet, en appelant d une direction et s un sens, on a:

- antiD = Horizontal + Vertical — d
- antiS = Positif + Negatif — s

Il y a également une variable de classe relative à l'ordre des voisins dans les tables mémorisant le voisinage :

SortBlocks : c'est une table contenant quatre *blocks* de code Smalltalk, à raison de un block spécifique à chaque orientation. C'est une instance de *Array*.

Cette constante est nécessaire pour ordonner les voisins d'un composant pour chaque orientation. Chacun des quatre blocks spécifie la façon d'insérer un nouveau composant dans une table de voisins. Il s'agira toujours d'insérer un nouveau voisin dans l'ordre croissant des distances relativement au composant. En fait, chaque block de code spécifie la manière de calculer l'index (la position dans la table) d'un nouveau voisin à insérer dans une table.

Description et signification des variables d'instance

Chaque composant dispose de quatre attributs :

nom : c'est un identificateur unique désignant le composant. Il est calculé et attribué par le *projetComposant* associé au composant lors de la création du composant. Le système est conçu pour pouvoir gérer au maximum 100 composants, et les identificateurs iront de C00 à C99. C'est une instance de la classe *String*.

projet : c'est le projet associé au composant. Grâce à cet attribut, le composant peut obtenir des informations concernant les autres composants du projet, en envoyant un

message à la base du projet. Cependant le composant s'adresse toujours directement à ses voisins (ses agents pertinents). C'est une instance de *ProjetComposant.*

enveloppe : c'est le rectangle dans lequel est inscrit le composant. C'est une instance de la classe *Rectangle* disponible dans le système Smalltalk.

voisins : c'est un tableau unidimensionnel à quatre entrées, chacune d'entre elles étant associée à une orientation précise. Chacune des entrées contient une instance de *SortedCollection*, qui est une sous-classe de *OrderedCollection* (ceci est vrai aussi bien pour Smalltalk/V que Smalltalk-80). Chaque *sortedCollection* de ce tableau est ordonnée suivant la méthode indiquée à l'indice correspondant dans la variable de classe *SortBlocks.* Chaque fois qu'un composant rajoutera un nouveau voisin dans son voisinage, il le fera par un message du type :

```
(voisins at: uneOrientation)add : unComposant
```

, ou encore s'il est possible que le voisinage ait subi une modification "hors protocole" :

```
voisins
  at : uneOrientation
  put : (vDS asSortedCollection :
            (SortBlocks at: uneOrientation)).
```

Les diverses catégories des méthodes de ComposantRectangle

Pour les métodes de classe, on a les catégories suivantes :

création : méthodes qui permettent de créer un nouveau composant. Il n'y en a qu'une.

orientation : méthodes qui renvoient, calculent, ou convertissent des orientations.

composantes : méthodes qui renvoient la valeur des composantes d'orientation.

initialisation : méthodes qui initialisent la classe *ComposantRectangle.*

Pour les méthodes d'instance, les diverses catégories sont :

renvoisGéométriques : ces méthodes renvoient des informations concernant la géométrie du composant (par exemple son enveloppe, son bord suivant une direction, ...).

voisinagePrive : ce sont des méthodes privées, c'est-à-dire qui ne peuvent être déclenchées que par un composant pour lui-même ou pour un autre composant. Ces méthodes agissent directement sur le voisinage d'un composant par remplacement, destruction, ajout.

caractéristiques : ces méthodes permettent d'obtenir des informations sur le composant (par exemple son nom).

prédicatsTopologiques : ces méthodes permettent de vérifier des relations géométriques ou topologiques entre composants.

constructionVoisins : ces méthodes construisent le voisinage du composant receveur du message; elles contiennent donc la totalité ou une partie de l'algorithme de construction de voisins.

constructionVoisinsPrive : ces méthodes réalisent une partie du calcul de voisinage mais ne peuvent être déclenchées que par un composant pour lui-même.

apparitionDisparitions : ce sont les méthodes implémentant les algorithmes d'apparition ou de disparition de voisins.

actionsSurComposant : ces méthodes implémentent des actions réelles sur les composants. Il y en a trois : le placement, le déplacement, la destruction.

III.4.6.2 Méthodes de classe de ComposantRectangle

Pour la catégorie *création*, il n'y a qu'une méthode de classe :

nom : unNom **enveloppe** : unRectangle **projet** : unProjet . C'est la méthode standard de création d'une instance de *ComposantRectangle*. Le composanmt créé aura comme attributs ceux passés en paramètres et devra calculer lui-même son propre voisinage en se plaçant. La méthode effectue des vérifications de validité sur chacun des paramètres passés, de telle manière que la création d'un composant ayant des attributs incohérents est impossible.

Pour la catégorie *orientation*, les méthodes de classe implantées sont :

est : renvoie la valeur de l'orientation Est.

ouest : renvoie la valeur de l'orientation Ouest

nord : renvoie la valeur de l'orientation Nord

sud : renvoie la valeur de l'orientation Sud

chainePour : uneOrientation : renvoie le symbole de variable de classe correspondant à uneOrientation. Cette méthode est utilisée pour décrire les caractéristiques d'un composant, car on précise les voisins pour chaque orientation.

pointCardinalPour : uneOrientation permet de décomposer uneOrientation en ses deux composantes. La méthode renvoie une instance de *Point* dont les coordonnées sont respectivement la direction et le sens de l'orientation. Les méthodes d'*apparitionDisparition*, de *constructionVoisin* et d'*actionsSurComposant* s'utilisent intensivement cette méthode.

testerOrientation : dirSens est une méthode de contrôle qui vérifie que dirSens est vraiment une orientation possible.

Pour la catégorie *composantes*, les méthodes de classe implantées sont :

negatif renvoie la valeur du sens d'orientation négatif.

positif renvoie la valeur du sens d'orientation positif.

horizontal renvoie la valeur de direction horizontale.

vertical renvoie la valeur de driection verticale.

Pour la catégorie *initialisation*, les méthodes de classe implantées sont :

initialiseOrientations initialise les quatre variables de classe Est, Ouest, Nord, Sud aux valeurs définies précédemment.

initialiseComposantesOrientation initalise les quatre variables de classe Negatif, Positif, Horizonial, Vertical aux valeurs constantes définies précédemment.

initialiseBlocksDeTri initialise les quatre blocks de tri utilisés par les sortedCollection de l'attribut *voisins* des *composantRectangles* et relatifs aux quatre orientations possibles.

initialiser lance les trois méthodes de classe précédentes.

Mise à part la méthode de création de composant, les méthodes de classe de *ComposantRectangle* ne servent en fait qu'à gérer correctement l'implantation choisie des orientations.

III.4.6.3 Méthodes de la catégorie *renvoisGéométriques*

Toutes ces méthodes ne font que renvoyer des informations sur les valeurs de l'enveloppe du composant. Ces méthodes sont :

droit, **gauche**, **haut**, **bas** renvoient respectivement les bords H+, H-, V+, V- du composant récepteur du message. L'enveloppe du composant étant une instance de *Rectangle* on utilise les méthodes spécifiques de cette classe pour réaliser cette opération.

bord : uneDirection et **bordAnti** : uneDirection renvoient les bord D et $\bar{D}$ respectivement du composant receveur pour la direction uneDirection.

enveloppe renvoie le *rectangle* définissant l'enveloppe du composant.

III.4.6.4 Méthodes de la catégorie *voisinagePrive*

Ce sont des méthodes dites *privées*, c'est à dire qui ne peuvent être envoyées que par un composant pour un autre composant. Seules les instances de la classe Composant peuvent utiliser ces méthodes. Elle agissent directement sur la valeur de l'attribut *voisin* du composant receveur :

voisins et **voisins** : unVoisinage permettent respectivement d'obtenir le voisinage du composant receveur et de lui assigner directement un voisinage quelconque. Cette deuxième méthode n'est employée que lorsque l'on est sûr de la validité du voisinage affecté, par exemple après le calcul du voisinage d'un composant.

ajouterVoisin : unComposant **pourOrientation** : orientation ajoute unComposant au voisinage du composant receveur suivant l'orientation uneOrientation. Ce message ne peut être envoyé que par un composant ayant détecté le receveur comme nouveau voisin. Le paramètre *unComposant* sera donc le plus souvent l'envoyeur du message.

detruireVoisin : unComposant **pourOrientation** : uneOrientation retire du voisinage composant receveur associé à l'orientation uneOrientation le composant unComposant. Ce message ne peut être envoyé que par un composant ayant détecté la disparition du receveur comme voisin. Donc le paramètre *unComposant* a toutes chances d'être l'envoyeur du message.

remplacerVoisin : ancienEtatDeVoisin
par : nouvelEtatDeVoisin
pourOrientation : orientation
remplace l'ancien voisinage *ancienEtatDeVoisin* orienté suivant *orientation* par le nouveau voisinage orienté *nouvelEtatDeVoisin*. Ce message ne peut être envoyé que par un composant qui a été déplacé sans conflit, et dont le receveur est resté voisin.

III.4.6.5 Méthodes de la catégorie *caractéristiques*

Ces méthodes sont du genre "méthodes d'attribut", c'est à dire qu'elles encapsulent l'accession à des attributs de composant. Il s'agit essentiellement d'informer le demandeur sur des valeurs courantes d'attributs, ou de renvoyer sous un format particulier de l'information concernant une requête.

nom renvoie l'identificateur du composant

nom : unNom
enveloppe : unRectangle
projet : unProjet assigne directement les trois attributs *nom, enveloppe, projet*du composant receveur aux valeurs passées en paramètres du message. Cette méthode est essentiellement utilisée à la création d'une nouvelle instance de *ComposantRectangle*.

showCaractéristiques renvoie une description textuelle des attributs du composant receveur. Cette description est constituée de l'identificateur suivant de l'enveloppe, puis des voisins pour chacune des quatre orientations. Cette méthode étant uniquement utilisée par l'interface du projet, en particulier pour donner une description textuelle du composant dans la zone de message, elle ne précise pas le nom du projet auquel est rattaché le composant..

III.4.6.6 Méthodes de la catégorie *prédicatsTopologiques*

Ces méthodes sont des méthodes privées qui permettent de vérifier des contraintes de positionnement d'un composant par rapport à un autre. Elles ne sont utilisées que lors de la détermination d'un voisinage ou lors des algorithmes d'apparition-disparition.

estIntersecteePar : unComposant **pourDirection** teste s'il y a une intersection horizontale ou verticale entre le composant receveur et l'émetteur du message.

inferieurA : unComposant **dansOrientation** : uneOrientation teste si le receveur précède ou succède le composant émetteur pour l'orientation uneOrientation. L'ordre pris ici est celui des ordonnées croissantes pour chacune des directions horizontales ou verticales.

situeAu : uneOrientation **de** : unComposant teste si le composant receveur est placé relativement à l'orientation uneOrientation par rapport au composant unComposant. Par exemple, "comp1 situeAu : Nord de: comp2" renvoie vrai si comp1 est au-dessus de comp2.

III.4.6.7 Méthodes de la catégorie *constructionVoisins*

Ces deux méthodes calculent le voisinage du composant receveur.

determinerVoisins effectue l'algorithme de calcul de voisinage du composant receveur et assigne directement le résultat obtenu sur l'attribut *voisins* du composant receveur.

determinerVoisinsPourDirection :
uneDirection détermine et assigne le voisinage du receveur pour la direction uneDirection. C'est le même algorithme que précédemment qui est utilisé, mais il est limité à la direction uneDirection. Cette méthode est utilisée pour des raisons d'efficacité de l'interface, lors d'un déplacement de composant.

III.4.6.8 Méthodes de la catégorie *constructionVoisinsPrivé*

Ces méthodes réalisent les calculs d'intersection (détermination des $I_{DS}(C)$) préparatoires à la détermination des voisins d'un composant. Ce sont forcément des méthodes privées puisque seuls les composants peuvent déterminer leur propre voisinage.

interDir : uneDirection **par** : unComposant renvoie l'intervalle de recouvrement entre unComposant et le receveur du message suivant la direction uneDirection. Renvoie nil si le recouvrement est vide. Cette méthode est utilisée au cours de l'exécution de la méthode *intersectantsPour :* , dans le prédicat topologique *estIntersectePar*, et dans toutes les déterminations de recouvrement entre composants dans les algorithmes de gestion de voisinage.

intersectantsPour : uneDirection détermine l'ensemble des composants qui sont en intersection suivant uneDirection pour le composant receveur du message. Renvoie le résultat obtenu dans un tableau à deux entrées indexé par les sens négatif et positif. Cette méthode est messagée au cours de l'exécution de la méthode *intersectants*

intersectants détermine l'ensemble des composants qui sont en intersection avec le composant receveur, et ceci suivant toutes les orientations possibles. Renvoie le résultat obtenu dans un tableau à quatre entrées, chacune d'entre elles désignant une orientation particulière.

III.4.6.9 Méthodes de la catégorie *apparitionDisparitions*

Ces méthodes implémentent directement les algorithmes d'apparition et de disparition de voisins au cours d'une action sur un composant receveur.

```
cherchezApparitionsDuesA : unComposant
  disparition : boolDisparition
  orientation : uneOrientation
  zoneLiberee : intervalle
  voisinageAScruter : vDSComposant
```

est la traduction Smalltalk de l'algorithme d'apparition de voisins tel que nous l'avons décrit dans le paragraphe III.3.4.6 . Le receveur gére seul son propre voisinage. Cette méthode ne renvoie rien.

```
cherchezDisparitionsDuesA : unComposant
  orientation : uneOrientation
  intersection : intervalle
```

est la traduction en Smalltalk et dans notre système de l'algorithme de disparition de voisins tel que nous l'avons présenté au paragraphe III.3.4.7 . Le receveur gre seul son propre voisinage. Cette méthode ne renvoie rien.

III.4.6.10 Méthodes de la catégorie *actionsSurComposant*

Les trois méthodes citées dans cette catégorie implémentent le comportement des composants lorsqu'on exécute une action utilisateur sur eux.

placement est la traduction création (après que ses attributs *nom, enveloppe, projet* aient été initialisés), comportement que nous avons décrit dans le paragraphe ... .

destruction est la traduction Smalltalk et dans notre système du comportement du composant receveur lors de sa destruction. Nous avons décrit ce comportement au paragraphe ... .

deplacementVers : uneOrientation **de** : uneDistance est la traduction Smalltalk et dans notre système du comportement du composant receveur lors de son déplacement dans l'orientation uneOrientation sur une distance de uneDistance. Nous avons décrit ce comportement au paragraphe ... .

III.4.6.11 Conclusion

La classe *ComposantRectangle* implémente les agents tels que nous les avons définis dans notre modèle.

Au niveau de la classe, les variables de classe servent à gérer correctement les orientations telles que nous les avons définies dans notre modèle, et à pouvoir ordonner les voisinages par ordre croissant des distances par rapport au composant concerné.Au niveau des instances, les attributs restent élémentaires, les deux attributs essentiels étant l'enveloppe (*enveloppe*) et les voisinages (*voisins*).

Les méthodes de classe sont divisées en deux types : une méthode de création d'instance, et des méthodes de gestion de l'orientation et d'initialisation . Les méthodes d'instances implémentent les algorithmes de gestion de voisinage et les comportements des composants lorsqu'une action est effectuée sur eux d'une part, et les autre méthodes servent essentiellement à eclater le code en de plus petites portions et donc à simplifier le code des méthodes précédentes.

Les composants sont reliés à un projet par la méthode de création de la classe, par les méthodes d'attributs, et par les méthodes de la catégorie *actionSurComposant.*

III.4.7 Conclusion

L'implémentation de cette interface associée au système distribué MAPS a été réalisée sur micro-ordinateur compatible PC avec Smalltalk/V. Elle insiste sur l'aspect indépendant des agents (composants) et sur l'aspect manipulation directe. Toutes les modifications intervenant sur un composant sont supposées ne jamais provoquer de conflit géométrique entre composants. Le prototype est basé sur peu de classes particulières du système Smalltalk, en fait essentiellement sur la classe *SortedCollection* pour ordonner les voisins à l'intérieur de chaque voisinage. Ceci a été fait volontairement pour permette un portage facile dans un autre langage. On dépend également de la classe *UniondIntervalles* que nous avons créée et décrite précédemment.

Deux classes Smalltalk importantes très liées au modèle MAPS ont étée implémentées : la classe UniondIntervalles et la classe ComposantRectangle. La première implémente tous les mécanismes nécessaires à la manipulation géométrique de base des voisinages (pour les construire et les manipuler), et la deuxième concerne les composants du projet en tant qu'agents.

Une troisième classe un peu fourre-tout (vu l'aspect prototype du logiciel actuellement implémenté) contient à la fois la partie interface pure du système et aussi l'entité globale nécessaire qui connait tous les agents existant : la classe ProjetComposant.

La structure du logiciel et le peu de classes existant dans cette implémentation sont dûs au fait qu'un portage devait être effectué avec le langage ADA dans un environnement PC,

et il fallait donc éviter de construire des hiérarchies de classe complexes car ADA ne permet pas l'héritage (uniquement l'abstraction de type).

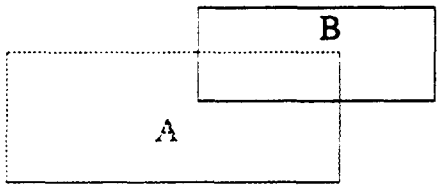
L'interface se révéle finalement assez efficace au niveau de la mise à jour des voisinages : environ trois secondes de calcul pour un contexte local de 50 voisins lors de la création, et une seconde lors de la modification par déplacement d'un composant (sur les cinquantes voisins initiaux , une trentaine d'entre eux disparaissent et un vingtaine d'autres apparaissent). Nous pensons donc que le portage en ADA ne serait pas une tâche trop difficile a réaliser, au moins du point de vue des deux classes UniondIntervalles et ComposantRectangle, et permettrait d'obtenir une efficacité suffisante pour une interface utilisateur en vrai grandeur. Par contre, la partie interface dépendra de l'environnement logiciel ADA utilisé et nécessitera de toute manière des modifications profondes, puisqu'elle est intimement liée au modèle MVC de Smalltalk.

III.5. Résolution distribuée de conflits géométriques

III.5.1 Introduction

Le modèle décrit précédemment permet la communication entre agents d'un système multi-agents, pour un problème spatial. Lors de sa création, chaque agent interroge l'instance de *projetComposant* à laquelle il est rattaché pour avoir connaissance des composants existant dans l'état actuel du projet et déterminer seul ses agents pertinents (Grâce à l'algorithme de détermination de voisins (Cf III.3.4.5). Ensuite, s'il subit une modification sans apparition de conflit, il est capable de mettre à jour seul son *voisinage*. Cette indépendance de gestion du contexte local, associée à une communication permanente avec les agents pertinents, est un élément clef de la construction et du bon fonctionnement du système.

Néanmoins, ce modèle ne fonctionne que si l'hypothèse qu'il n'existe jamais de conflit géométrique entre composants est vérifiée. Si deux composants se recouvrent, on ne sait plus déterminer les positions relatives et réciproques de deux composants. L'exemple qui suit illustre clairement le problème :



Intuitivement, on est tenté de dire que le composant B est situé au Nord de A, ou encore à l'Est de A; mais suivant notre modèle, il appartient aux quatres voisinages de A. Dès lors, la relation de voisinage que nous avons choisie est inadaptée à ce genre de situation, et le mécanisme de détermination et de gestion des agents pertinents ne fonctionne plus.

D'autre part, notre but est de proposer une aide "intelligente" à l'utilisateur de l'interface : on veut que le système résolve automatiquement les conflits géométriques créés par l'utilisateur lors de la modification ou de la construction d'un schéma.

Nous devons donc, pour proposer cette aide, construire un mécanisme de résolution de ces conflits. Le but de cette partie est de montrer que l'on peut bâtir un système de résolution de conflits géométriques entre composants, à partir du modèle de détermination et de gestion des agents pertinents que nous avons présenté précédemment.

Enfin, en réusssissant à fabriquer des algorithmes de détermination et de gestion du voisinage qui fonctionnent de manière distribuée, nous avons fait un premier pas important vers la décentralisation du système, en restant toutefois à un niveau où aucune résolution ou "intelligence" n'est nécessaire. Pour que le système puisse être à juste titre qualifié de décentralisé, il faudrait que les mécanisme de résolutions ou de planification fonctionnent également de manière distribuée. Nous aurons ainsi construit la base d'un système réellement décentralisé de planification spatiale associé à une interface utilisateur, au-dessus duquel on

pourra par la suite greffer des heuristiques ou des techniques IA indépendantes du modèle d'architecture IA choisi.

Nous proposons dans cette partie deux stratégies de résolution de conflit et de propagation de cette résolution parmi les composants rectangulaires d'une base de projet gérée telle que décrit dans les parties précédentes, ainsi que les techniques de résolution qui en découleront.

Comme dans la partie algorithmique précédente, nous essayerons systématiquement de distribuer le fonctionnement de la résolution parmi les différents composants. Ainsi chaque composant disposera des mêmes caractéristiques et du même potentiel de base de prise de décision ou de résolution proprement dite, hormis les caractéristiques fonctionnelles ou sémantiques de l'élaboration du schéma du produit (circuit électrique ou piéce mécanique). Cependant la prise de décision doit prendre en compte des aspects topologiques (et fonctionnels) du problème, suivant les caractéristiques des composants, tandis que la résolution de base relève d'un algorithme standard (que l'on pourrait appeler "moteur de résolution de conflit spatial").

Nous ne nous sommes intéressés ici qu'à l'aspect de résolution distribuée parmi les composants du système; l'aspect " prise de décision" relevant d'un caractère d'expertise important que nous n'avons pas approfondi dans cette étude.

Nous avons étudié deux stratégies générales de résolution de conflits géométriques, qui ont aboutis chacune à des moteurs de résolution très spécifiques. Nous en présentons les fondements dans le paragraphe qui suit.

III.5.2 Présentation des deux stratégies étudiées

Nous pouvons présenter le problème à résoudre de la manière suivante : après une action de l'utilisateur sur un composant C, celui-ci détecte un conflit entre lui-même et un ou plusieurs de ses agents pertinents. Il doit alors déclencher un système de résolution du (des) conflit(s). Pour ce faire, nous pensons qu'il y a essentiellement deux familles de stratégies de résolution possible : la première consiste à tenter de résoudre à tout prix et plutôt impérativement le conflit local, la seconde consiste à chercher une solution au problème local sans toutefois ignorer les contraintes de l'environnment global, ceci par propagation de la résolution et des contraintes existantes.

Dans le premier cas, nous supposons que le composant qui détecte un conflit agit en esclave. Partant d'une action de l'utilisateur, le composant modifié détecte le conflit, mais donne la priorité à l'action qui lui est demandée. Il effectue cette action, et ne teste pas la validité globale de la modification. Après avoir détecté tous les conflits issus de cette nouvelle situation, il décide des actions que les composants en conflit devront effectuer. Ceux-ci réagissent alors de la même manière que le composant précédent vis-à-vis de l'action utilisateur, et ainsi de suite jusqu'à résolution totale des conflits géométriques ou échec définitif.

Dans la deuxième stratégie, le composant ne réagit pas aveuglément à la modification demandée, mais agit plutôt en "expert" pour résoudre la situation conflictuelle. Il tentera cette

fois de résoudre le problème en prenant seul une décision dépendante de son contexte local et du conflit. Si malheureusement il ne peut envisager aucune solution personnelle au conflit, il agira en coopération avec les composants avec lesquels il est en conflit pour trouver une solution. Les composants avec lesquels il coopère travaillent de la même manière.

Les raisons du choix de ces deux stratégies sont qu'elles simulent bien le comportement naturel d'un utilisateur face à un problème de schématique, lorsqu'il doit effectuer une modification qui engendrera des conflits.

La première stratégie simule bien un comportement empirique où l'utilisateur ne voit pas très bien comment il va pouvoir résoudre le conflit, son expertise du domaine est insuffisante pour envisager immédiatement une résolution raisonnée. Il va essayer de résoudre le problème "pas à pas". Ce comportement se retrouve chez les utilisateurs novices du domaine ou quand un expert se trouve face à un problème d'un genre inconnu.

Dans le deuxième cas, on simule plutôt le comportement d'un utilisateur expert qui "reconnait" un type de situation. Il analyse cette situation et entrevoit en avance les implications d'une modification particulière. Il est capable d'utiliser sa connaissance pour arriver rapidement à une solution.

Le comportement réel d'un utilisateur fait souvent appel à ces deux types de stratégies d'une manière assez imbriquée, et nous n'avons pas étudié la coopération et l'imbrication des deux mécanisme au cours d'une même tentative de résolution. Néanmoins une première étape à franchir pour réussir à simuler le travail du concepteur consiste à être capable de simuler ces deux comportements séparément.

Pour pouvoir distribuer les mécanismes de résolution, nous proposons des structures de données dynamiques associées à chaque composant qui mémoriseent les états successifs de celui-ci au cours de la recherche, jusqu'à la détermination d'une solution. De plus ces structures permettent, si elles sont associées aux algorithmes proposés, d'installer des mécanismes de remise en cause nécessaires en cas d'échec.

Ce backtracking est également distribué entre les composants ayant participé à l'élaboration de la recherche courante. L'expertise et la représentation des connaissances est indépendante du mécanisme de résolution, l'ingénieur cogniticien est libre de choisir la représentation qu'il désire.

Nous avons également supposé que la seule action qu'un composant pouvait demander à un autre d'exécuter est un déplacement, c'est à dire un changement de position dans l'espace à l'intérieur de son contexte local, mais jamais de modification de structure.

III.5.3 Stratégie par action, puis résolution

III.5.3.1 Description

Le fonctionnement de l'algorithme de résolution basé sur cette stratégie est le suivant :

- Un composant reçoit l'ordre (par un composant "maitre" ou par l'utilisateur de l'interface) d'exécuter une action un déplacement, et par l'utilisateur de l'interface seulement une modification d'enveloppe, un changement d'orientation, ou une déportation.

- Il réalise cette action, principalement en modifiant la description de son enveloppe.

- Pour tous les composants avec lesquels il sera en conflit après cette modification, il cherchera un déplacement à effectuer par ce composant.

- chacun des composants en conflit est alors méssagé avec demande de déplacement, et il réagit à cette demande sur un même principe bien défini.

Quelques remarques s'imposent au sujet de ce mécanisme de résolution :

- Le voisinage des composants n'est utilisé que pour la communication avec des agents pertinents. En effet, la résolution donne la priorité à l'action à effectuer, mais une fois que cette action est réalisée, la détermination des conflits doit a priori se faire entre le composant modifié et ses voisins. D'autre part, un composant doit se servir de son contexte local pour prendre une décision adéquate.

- On peut recalculer les voisinages en fin de résolution de l'action initiale. Pour tous les composants modifiés de la base, on redéterminera le voisinage comme celui d'un nouveau composant créé. Pour cela, il suffira de prendre le dernier état de chaque composant de la base de projet (c'est à dire son enveloppe non modifiée ou la dernière position d'enveloppe).

- La stratégie de recherche de solution est en "profondeur d'abord". Cela est caractérisé par le fait que la décision prise par un composant à un niveau de la recherche n'est pas remise en cause par les voisins avec lesquels il se trouve en conflit . Néanmoins en cas d'échec on parcours un sous-arbre de résolution qui est frère du précédent. L'idée est de trouver une solution, mais pas à coup sur la meilleure; pour cela il faudra implémenter des heuristiques et des fonctions dévaluation de la solution adaptés [Rich 83].

Nous décrivons dans la partie suivante l'algorithme de résolution que nous avons bâti à partir de cette stratégie.

III.5.3.2 Algorithme de résolution associé

Dans la description de notre algorithme, nous utiliserons certains termes dont nous expliquons ici la signification, et qui seront mis en évidence dans le text par soulignement. En voici la liste :

1. *réussite* : il s'agit d'un mot-clef désignant la réussite de la résolution à un niveau donné de l'arbre de recherche.

2. *échec-définitif* : c'est également un mot-clef, mais qui désigne un échec de résolution à un niveau donné de l'arbre de recherche. Il est caractérisé comme définitif parce que le sous arbre de recherche à partir de ce niveau n'a pu donner aucune solution. Cette entité va provoquer une remontée (backtracking) dans la recherche de solutions.

3. *liste-des-composants-en-conflit* : ce sont tous les composants qui sont en conflit géométrique avec le composant qui exécute l'algorithme de résolution

4. *liste-des-conflits-résolus* : c'est une liste des composants qui étaient en conflit avec le composant qui exécute l'algorithme de résolution, et pour lesquels le conflit a été résolu.

5. *liste-des-composants-modifiés* : est la liste des composants qui ont subi une modification due à la résolution. Cette entité est associée avec la suivante.

6. *niveau-de-résolution* : représente la profondeur à laquelle on se trouve dans l'arbre de recherche des solutions. C'est une valeur entière.

Toutes ces données sont en fait des variables locales de l'algorithme sauf les deux dernières qui sont généralisées à l'ensemble de la résoluiton.

Description de l'algorithme en pseudo-code

A. le receveur réalise l'action demandée, mais sur un composant fictif, c'est à dire qu'il créé et mémorise une nouvelle enveloppe représentant le nouvel état hyptothétique de lui-même.
il ajoute à sa référence à la *liste-des-composants-modifiés*,
il incrémente également le *niveau-de-résolution*.

B.le *receveur* détermine et crée la *liste-des-composants-en-conflit*.

C. **si** *la liste-des-composants-en-conflit = ()*
alors on arrête la résolution et on retourne *réussite* à l'appelant.

D. **sinon**

— **tantque** il n'y a pas d' *échec-définitif* **et**
on n'a pas essayé tous les composant de liste-des-composants-en-conflit,

— choisir un composant C dans *liste-des-composants-en-conflits*.

— **tantque** *on n'a pas résolu ce conflit* **et**
on n'a pas essayé toutes les heuristiques,

— choisir une heuristique adaptée au cas de C.

— déterminer l'action que C doit réaliser.

— construire cette action et demander à C de la réaliser (C relance l'algorithme de résolution pour lui).

— **si** *C renvoie réussite*,
alors ajouter C à *liste-des-conflits-résolus*.

fin tantque

— **si** *aucune solution trouvée pour le cas de C*,

— enregistrer un *échec-définitif*.

— demander à tous les composants de *liste-des-conflits-résolus* d'annuler leur dernier état modifié.

fsi.

fin tantque.

— **E si** *on a enregistré un échec définitif*,

— le receveur annule son dernier état modifié.

— **si** *il n'y a plus d'état modifié pour le receveur*,

oter le receveur de la *liste-des-composants-modifiés*

fsi

— renvoyer *échec-définitif* à l'appelant.

sinon

— décrémenter le *niveau-de-résolution*.
si *niveau-de-résolution* vaut 0,
demander à tous les composants de *liste-des-composants-modifiés* de :

— valider leur dernier état modifié.

— recalculer leur voisinage sur le nouvel état de la base.

fsi

— renvoyer *réussite* à l'appelant.

fsi.

fsi.

Finalement, quand le receveur initial reçoit *réussite* de tous ses voisins, il leur demande de valider leur état courant, et ainsi de suite par récurrence sur tous les composants impliqués dans la résolution. Cette façon d'opérer permet de détecter des bouclages dans la résolution, c'est-à-dire le fait qu'un composant soit impliqué au moins deux fois de suite au cours d'une tentative de résolution sans qu'on ait abouti ni à un échec ni à une solution.

III.5.3.3 Structure de données et commentaires généraux sur l'algorithme

Pour pouvoir permettre ce type de résolution distribuée, chaque composant doit maintenir en cours de recherche ses états successifs. Dans notre cas, il s'agit simplement de l'enveloppe modifiée du composant. Mémoriser le dernier état modifié du composant est insuffisant, car un composant peut voir son état être modifié plusieurs fois pendant une tentative de résolution, sans que l'état courant permette une solution générale au problème. Autrement dit, pour permettre une remontée, il faut mémoriser l'ensemble des états modifiés du composant depuis le début de la résolution, ou un moyen de revenir à chacun des états au cours de la résolution.

Nous proposons donc la structure de données suivante de composant pour une adaptation à l'algorithme précédent :

1. nom du composant.

2. dernière enveloppe valide.

3. projet associé au composant.

4. voisinage du composant.

5. pile d'états temporaires du composant.

le nom du composant est supposé unique dans la base de projet, la dernière enveloppe valide est celle qui existait avant le début de la résolution, et la pile contient les états successifs du composant au cours de la résolution.

Remarques : (concernant la pile d'états temporaires) .

1. Un état de composant est caractérisé uniquement par son enveloppe dans notre description extrêmement simplifié.

2. Un composant empile son nouvel état quand il reçoit le message d'exécution d'une action.

3. un composant dépile son ancien état quand :

   a. le composant annule son état modifié de lui-même (partie E de l'algorithme).

b. le composant annule son état modifié par ordre de l'envoyeur du message d'exécution de l'action (correspond à la fin de la partie de l'algorithme : un échec définitif sur n composants annule toutes les hypothèses en cours sur les autres composants

4. un composant réinitialise sa pile (pile vide) quand une solution au problème initial a été trouvée et que le voisinage du composant a été recalculé à la fin de la résolution.

5. Le fait de ne pas dépiler lorsque le composant se déplace à un endroit libre ou à un endroit représentant une solution exprime le fait que l'hypothèse n'est pas définitve (qu'elle pourra donc être remise en cause), et permet d'effectuer le backtracking nécessaire en cas de remise en cause de cette hypothèse.

La première partie de l'algorithme (A, B) consiste à prendre en compte l'action demandée et à mettre à jour des variables locales (*liste-des-composants-en-conflits*) et des variables globales *liste-des-composants-modifiés*, *niveau-de-résolution*. On empile également un nouvel état du composant sur la pile d'états temporaires.

La partie étiquettée C sert à rendre l'algorithme plus efficace et à alléger la gestion de la liste des composants modifiés.

La partie D est la partie centrale de la résolution. elle comprend l'expertise du domaine avec le choix d'un composant en conflit, une heuristique adaptée à la situation, et la détermination de l'action que le composant en conflit doit réaliser. Elle permet également d'éliminer un sous-arbre de résolution lorsqu'un échec définitif dans ce sous-arbre est détecté. Notons que les échecs définitifs sont détectés (ou plus exactement décidés) au niveau le plus profond de la résolution grâce aux heuristiques et à l'état courant du composant détectant l'échec. La situation d'échec est alors remontée au composant "*maître*".

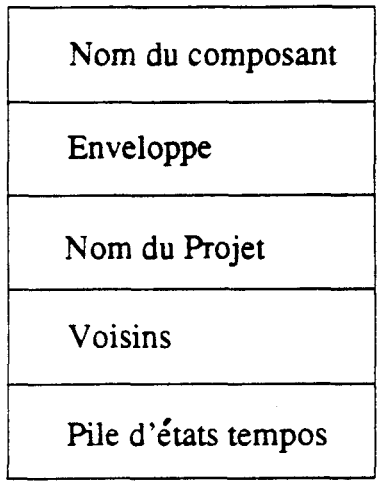
La partie E gére la fin d'une résolution par un composant. S'il y a un échec définitif, il faudra réaliser une remontée en même temps qu'on arrête la résolution courante. Sinon, il faudra pouvoir distinguer le cas où l'on se trouve à la racine de l'arbre de recherche ou dans un sous-arbre. Ceci est réalisé par le *niveau-de-résolution*. Enfin, si la résolution est terminée (une solution générale a eté trouvée), il faut réactualiser les voisinages des composants sur le nouvel état de la base.

III.5.3.4 Application à un exemple

Pour bien montrer le fonctionnement de cet algorithme de résolution, nous allons l'appliquer sur un cas concret. Nous montrerons les évolutions de la base de projet et des composants de cette base au cours de la résolution. Chaque action ou portion de l'algorithme engagée par un composant sera complétement précisée.

Représentation des composants

La représentation des composants que nous adoptons dans cette description est la suivante :

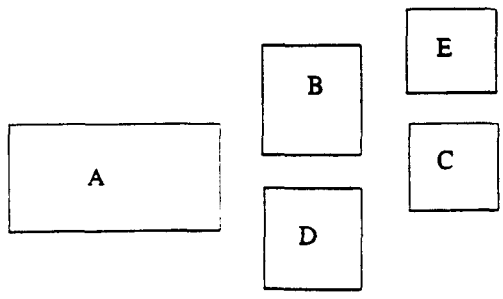


Il faut noter que la donnée la plus importante est la pile d'états temporaires. Nous préciserons toujours son contenu. L'enveloppe sera uniquement caractérisée par un nom spécifique. Les autres données ne changent pas au cours de la résolution : cela est clair pour le nom du composant et pour le nom de projet, par contre nous avons vu que les voisins ne servaient qu'à la communication avec les agents pertinents.

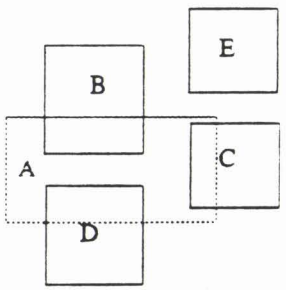
Au cours du déroulement de la résolution les numéros indiqueront la profondeur dans l'arbre de recherche, et les lettres (a, b, c, d, e) feront référence aux lettres (A, B, C, D, E) identifiant les étapes de l'algorithme (Cf III.5.3.2)

Le projet avant l'action de l'utilisateur

Nous avons pris le cas de cinq composants nommés A, B, C, D, E, disposés de la manière suivante dans l'espace :



L'utilisateur veut déporter le composant A (au sens de notre système, car l'utilisateur a le sentiment de déplacer un composant, et non de le déporter), en une nouvelle position créatrice de conflits géométriques avec B, C, D :



La représentation de la base avant la déportation de A est la suivante :

A	B	C	D	E
EA	EB	EC	ED	EE
P	P	P	P	P
((D, B, C), , ,)	((C,E), A, , D)	(, (D, B, A), E,)	(C, A, B,)	(, B, , C)
( )	( )	( )	( )	( )

Déroulement de l'algorithme

0. Après l'action de l'utilisateur, A déclenche pour lui-même l'algorithme de résolution sur la déportation, menant au nouvel état A'.

a. A crée une nouvelle enveloppe EA'.

b. A détermine et calcule les conflits engendrés par cette modification. Les composants en conflit sont (B, C, D).

d. A n'a pas reçu d'échec définitif et la liste-des-composants-en-conflit n'est pas épuisée
→ A choisit le composant B.
→ A choisit une heuristique pour le cas B.
→ A construit l'action à réaliser par B : "déplacez-vous vers le Nord de ...".
→ A demande à B de "résoudre cette action".

1. B reçoit le message de A, menant à l'état B'.

a. B calcule sa nouvelle enveloppe EB'.

b. B détermine les conflits engendrés par cette modification.
liste-des-composants-en-conflits = ().

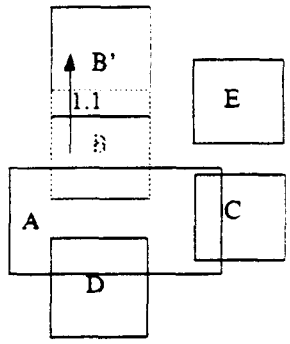
c. B renvoie *réussite à l'appelant.*

A ce moment, l'état de la base de projet est :

A	B	C	D	E
EA	EB	EC	ED	EE
P	P	P	P	P
((D, B, C), , ,)	((C,E), A, , D)	(, (D, B, A), E,)	(C, A, B,)	(, B, , C)
(EA')	(EB')	( )	( )	( )

Et l'ensemble des composants touchés par la résolution est { A,B}.

Le schéma du projet est alors le suivant : les fléches numérotées indiquent le déplacement effectué, la profondeur de résolution (le premier chiffre) et le numéro d'ordre de cette action à ce niveau de résolution (le deuxième chiffre).



0. A reçoit *réussite* de B , *liste-des-conflits-résolus* = (B).
remarque : nous avons ici le cas le plus simple de résolution, car le composant en conflit a été déporté en une place complétement libre.

d. A n'a pas reçu d'échec définitif, il reste le cas de C et D à résoudre.
— A choisit le composant C.
— A choisit une heuristique pour C.
— A construit une action à réaliser par C : " se déplacer au Nord de ...".
— A demande à C de résoudre cette action.

1. C reçoit le message de A.

a. C calcule sa nouvelle enveloppe EC'.

b. C détermine les conflits engendrés par cette modification : *liste-des-composants-en-conflit* = (E).

d. C n'a pas reçu d'échec définitif et il reste le cas de E à résoudre.
— C choisit le composant E.
— C choisit une heuristique pour E.
— C construit une action à réaliser par E : "se déplacer au Nord de ...".
— C demande à E de résoudre cette action.

2. E reçoit le message de C, et réagit de la même manière que B précédemment ... E renvoie *réussite* à C.

1. C reçoit *réussite* de E, *liste-des-conflits-résolus* = (E).
Remarque : Dans ce scénario, on a eu une résolution en chaîne : une action à réaliser sur un composant en entraine une autre sur un autre composant, et ainsi de suite jusqu'à trouver une fin à cette chaîne de résolution. Il n'y aura pas de remise en cause au cours de cette recherche.

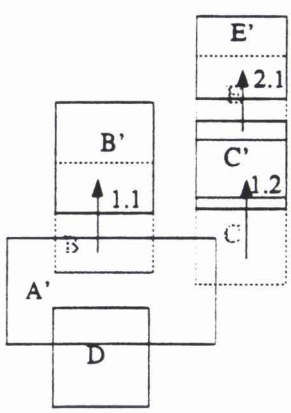
Tous les composants en conflit avec C ont été sélectionnés.

e. C n'a pas enregistré d'échec définitif.
— C renvoie *réussite* à A.

A cet instant, l'état de la base est le suivant :

A	B	C	D	E
EA	EB	EC	ED	EE
P	P	P	P	P
((D, B, C), , ,)	((C,E), A, , D)	(, (D, B, A), E,)	(C, A, B,)	(, B, , C)
(EA')	(EB')	(EC')	( )	(EE')

Le schéma du projet est alors le suivant :



Et l'ensemble des composants touchés par la résolution est { A,B, C, E}.

0. A reçoit *réussite* de C, *liste-des-conflits-résolus* = (B, C).

d. A n'a pas reçu d'échec définitif, il reste le cas de D à résoudre.
— A choisit le composant D.
— A choisit une heuristique pour le cas de D.
— A construit une action à réaliser par D : " se déporter au Nord-Est de ...".
— A demande à D de résoudre cette action.
Remarque : l'action choisie par A concerne un déplacement ou une déportation de n'importe quel type. Ici, nous avons supposé que l'heuristique choisie pas A provoquait un nouveau conflit entre D et C'.

1. D reçoit le message de A.

a. D calcule sa *nouvelle* enveloppe ED'. Cette nouvelle position de D provoque un conflit avec C'.

b. D détermine les conflits engendrés par cette action : *liste-des-composants-en-conflit* = (C').

d. D n'a pas reçu d'échec définitif et il reste le cas de C' à résoudre.
— D choisit le composant C'.
— D choisit une heuristique pour le cas de C'.
— D choist une action à réaliser par C' : "se déplacer à l'Est de ...".
— D demande à C de résoudre cette action.

2. C reçoit le message de D, calcule sa nouvelle enveloppe EC", et réagit comme B et E précédemment, c'est à dire qu'il n'y a pas de conflit, et D renvoie *réussite* à l'appelant.

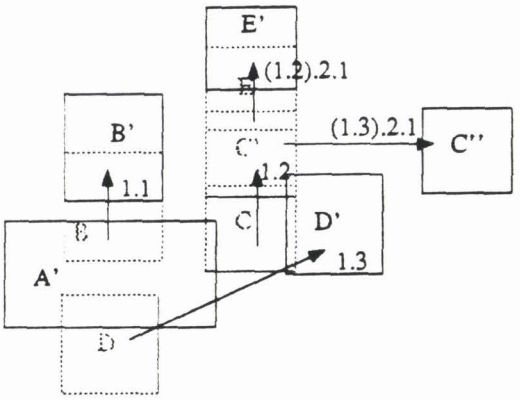
1. D reçoit *réussite* de C, *liste-des-conflits-résolus* = (C). Il n'y plus de conflit à résoudre.

e. D renvoie *réussite* à l'appelant A.

A cet instant, l'état de la base est le suivant :

A	B	C	D	E
EA	EB	EC	ED	EE
P	P	P	P	P
((D, B, C), , ,)	((C,E), A, , D)	(, (D, B, A), E,)	(C, A, B,)	(, B, , C)
(EA')	(EB')	(EC', EC'')	(ED')	(EE')

Le schéma du projet est alors le suivant :



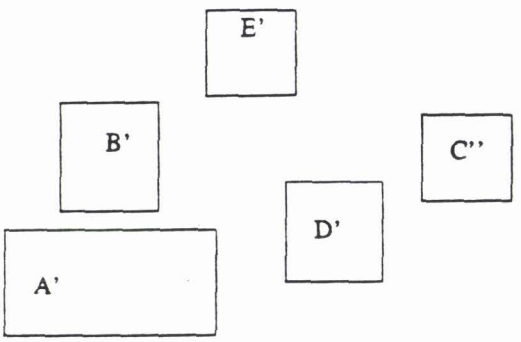
Et l'ensemble des composants touchés par la résolution est {A, B, C, D, E}.

0. A reçoit *réussite* de D, *liste-des-conflits-résolus* = (B,C,D). tous les conflits sont résolus.

e. A n'a pas reçu d'échec définitif. Le niveau de résolution est 0.
— A demande à tous les composants touchés par la résolution de valider leur dernier état (remplacer leur enveloppe par la dernière enveloppe empilée).
— A demande à tous les composants de *liste-des-composants-modifiés* de recalculer leur voisinage sur ce nouvel état de la base de projet
— A renvoie réussite à l'appelant (i.e. l'utilisateur ou l'interface).

A cet instant, l'état de la base de projet est le suivant :

A	B	C	D	E
EA'	EB'	EC''	ED'	EE'
P	P	P	P	P
((D,B), , ,)	((D,C), , , A)	(, (D, B), ,)	(C, (A, B), E,)	(, , , (D, A))
( )	( )	( )	( )	( )



Remarque : la dernière partie de la résolution, c'est-à-dire celle qui impliquait les composants D et C, montre un aspect particulier de l'algorithme : on peut remettre en cause une solution pour un conflit en tentant de résoudre un autre conflit indépendant (a priori) du premier. Nous verrons plus loin que ceci est parfois néfaste à la qualité de la solution trouvée.

III.5.3.5 Discussion sur cet exemple

1. Nous avons délibérément choisi un scénario de résolution qui ne remettait pas en cause les solutions déjà trouvées pour certains conflits. En fait, l'exemple ne montre aucun retour-arrière. Ces retours-arrière interviendront quand un composant qui cherche à résoudre un conflit ne trouve aucune solution. Dès que ce cas se produit, le composant abandonne la recherche de solution dans le sous-arbre de recherche couramment exploré, en abandonnant l'hypothèse faite sur son enveloppe (il dépile cette enveloppe de sa pile d'états temporaires), et en signalant cet échec en retour au composant qui lui avait demandé de résoudre l'action donnant son nouvel état. C'est le fait qu'on gère une pile qui permet de faire des retour-arrière à n'importe quel niveau de l'arbre de recherche.Tout ce mécanisme de remontée est géré dans la fin de la partie D et en début de la partie E de l'algorithme. Néanmoins on pourra remarquer d'une part que le fait qu'ici le déroulement de la résolution soit séquentiel permet de ne pas mémoriser dans la pile d'états temporaires le composant ayant messagé le receveur, et d'autre part que le *niveau-de-résolution* peut être géré au niveau global également à cause du fonctionnement séquentiel de la résolution. En cas de résolution fonctionnant de manière réellement distribuée, il faudra d'une part mémoriser l'envoyeur du message dans l'état temporaire et d'autre part utiliser un moyen différent du niveau de résolution pour reconnaître l'initiateur de celle-ci (Cf résolution distribuée III.5.3.8).

2. Les heuristiques et les actions construites par les composants pour résoudre un conflit ont étés prises "au hasard". Par exemple, le fait que A décide de déplacer D au Nord-Est et qu'il provoque ainsi un conflit entre D' et C' a été choisi pour illustrer un type particulier de remise en cause d'une solution. Nous verrons plus loin que cette remise en cause a conduit à un défaut de la solution finale trouvée.

3. Finalement, malgré sa simplicité (seulement cinq composants dans l'univers courant du projet), l'exemple met bien en évidence trois comportements fréquents de la recherche de solution dus à ce type d'algorithme :

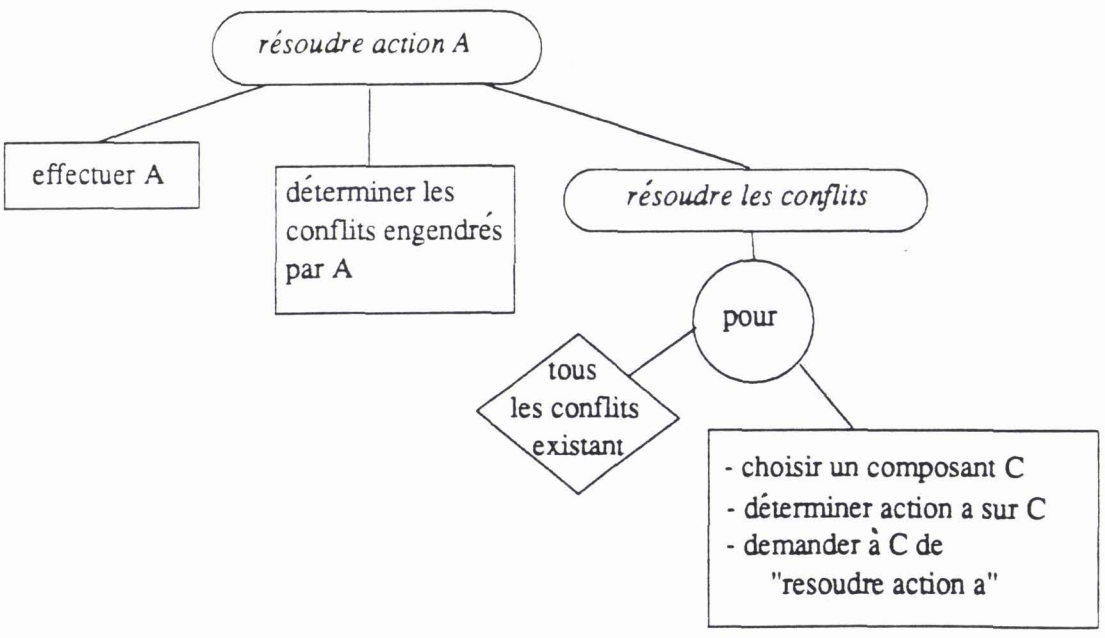
   a. on trouve une solution immédiate à un conflit : c'est le cas de la chaîne de résolution (B -> B'), ou encore (E —> E'). Nous pensons que c'est dans cette eventualité que l'algorithme proposera les meilleures solutions, d'une part parce qu'elles seront obtenues immédiatement, et d'autre part parce qu'elles ne remettent pas complétement en cause la topographie courante du projet (le passage de B à B' conserve presque complétement son voisinage).
   b. on déclenche une "solution en chaîne" cohérente pour l'ensemble du projet : c'est le cas de la chaîne de résolution ((C —> C'), (E —> E')). Cette eventualité dans l'élaboration d'une solution finale va dépendre de la partie d'expertise implantée dans le système, c'est-à-dire le choix de conflit à résoudre par le composant tentant de trouver une solution (ici nous avions choisi l'ordre d'occurence des

composants dans la *liste-des-composants-en-conflit*), dans la détermination d'une heuristique adaptée au cas de conflit rencontré, et l'action résultant de ces choix. Nous savons néanmoins que l'algorithme est capable de proposer une solution cohérente si l'expertise implantée est adaptée au problème rencontré.

c. remise en cause d'une solution indépendante déjà trouvée : c'est le cas de la chaîne de résolution ((D —> D'), (C' —> C")). Lors d'une élaboration de solution indépendante le composant C était passé de la position de C à la position de C'. Une branche parallèle et indépendante de cette première solution remet alors en cause la position de C'. Une question fondamentale est de savoir si l'on veut conserver le caractère indépendant de ces deux branches ou si on les lie dans la résolution. Nous avons choisi ici de lier les deux branches pour simplifier l'algorithme, car il faudrait sinon mémoriser comme état l'enveloppe du composant ainsi que l'initiateur de la résolution. Ici la chaîne de résolution serait alors décrite de la manière suivante : ((D —> (A, D')), (C' —> (D', C")). La gestion de l'initiateur d'une modification sur un composant alourdit l'algorithme de résolution.

III.5.3.6 Discussion sur l'algorithme

Grâce à cet algorithme de résolution, chaque composant dispose de capacités de raisonnement, uniquement limitées à l'aspect topologique du problème, et non aux problèmes plus fonctionnels des composants ou du projet final. Nous avons d'ailleurs encapsulé très localement dans l'algorithme les aspects dépendants de l'expertise du domaine. Nous pouvons résumer le fonctionnement par un arbre programmatique :



Cet arbre programmatique montre bien le caractère " en profondeur d'abord" de la résolution.

Expertise du domaine

La partie où l'expertise du domaine peut être introduite est celle où l'on choisit un composant en conflit, ainsi que l'heuristique choisie pour déterminer une action à exécuter.

- Concernant le choix d'un composant sur lequel appliquer une action, il s'agit de détecter d'avance quel est le meilleur choix de composant à manipuler.Ceci implique une reconnaissance de la situation topologique locale ainsi qu'une évaluation de la situation par rapport à des cas connus. Ces capacités qui peuvent ne faire appel qu'à des connaissances géométriques et topologiques peuvent également impliquer des aspects plus fonctionnels des composants.

- L'heuristique à choisir dépendra surtout du choix précédent de composant, mais également fortement du domaine d'application dans lequel on travaille. Dans un domaine électronique ou électrique, les aspects purement géométriques sont simples, par contre les aspects fonctionnels et les influences électriques entre composants peuvent avoir une très grande importance sur la topographie finale du projet. Par contre, dans un domaine comme la mécanique ou l'architecture, la fonction et la géométrie du composant par rapport au projet sont beaucoup plus fortement liées. Dans le premier cas, les heuristiques seront plutôt basées sur des caractères fonctionnels, et dans le deuxième sur des fonctions géométriques.

Bien que nous ayions voulu par la structure de cet algorithme séparer les qualités de résolution typiquement liées au domaine de celles liées à la géométrie, il n'est pas dit que ce choix soit réellement judicieux. Il faudrait faire une étude approfondie sur ce sujet pour infirmer ou confirmer cette hypothèse.

Remise en cause des solutions précédemment trouvées

La remise en cause d'une solution déjà établie peut parfois amener à des solutions inadaptées. Par exemple, la chaîne de résolution ((D —> D'), (C' —> C")) ne remet pas en cause (E —> E') qui devient inutile si C se place en C". Il y a alors création d'un trou inutile qui fera perdre de la place nécessaire pour disposer d'autres composants dans l'espace. Ceci montre clairement que le problème global est mal traité.

Il faudrait en fait élaborer des mécanismes de remise en cause plus sophistiqués pour pallier à ce genre d'inconvénients. On peut envisager par exemple, qu'en mémorisant le nouvel état du composant modifié, on ajoute à la pile des états temporaires la liste des composants "dépendants" du composant, c'est à dire la liste des composants en conflit suite à cette action. Dans le cas de notre exemple, cela donnerait pour la pile de C : ((EC',E), (EC",)). On peut également penser à un mécanisme inverse qui mémoriserait dans chaque état l'initiateur de la modification, cela donnerait pour C : ((A, EC'), (D, EC")). La première solution serait sans doute plus facile à gérer que la seconde car il est plus facile de remettre en cause un sous-arbre de résolution à partir d'un composant donné plutôt que de rechercher tous les composants influencés par une modification donnée.

Mais finalement, il faut constater que le fait de ne pas élaborer des mécanismes de remise en cause suffisamment poussés ménera à une dégradation topologique de la base au fur et

à mesure de la résolution (trous créés, composants éloignés, positions et alignements relatifs des composants détruits ...). Cependant des contraintes de nature fonctionnelle ou de nature non géométrique permettront d'éviter ce genre de problème. En effet, si E et C sont liés par une contrainte de proximité et d'alignement par exemple, lors de la résolution C' —> C" cette contrainte sera respectée par les heuristiques lors de la détermination de l'action à réaliser par le composant C".

Fonctionnement piloté par les données

C'est en effet un aspect intéressant du fonctionnement de cet algorithme. Chaque composant décide, seul ou avec ses agents pertinents (cela dépend de l'implémentation de l'expertise), des actions qu'il demandera aux autres de réaliser. Cette décision est prise essentielleme nt à partir du contexte local du composant. Le fonctionnement tel qu'il a été décrit est séquentiel, mais l'indépendance d'un composant vis-à-vis de celui qui l'a modifié est traduite simplement dans l'algorithme. Le mécanisme de résolution est assez facilement parallélisable si l'on fait abstraction des problèmes de synchronisation entre les envois de requête de résolution et les solutions trouvées à différents instants. A la différence des systèmes dits " à tableau noir", où chaque expert participant à l'élaboration d'une solution lit et écrit des informations sur ce tableau, chaque composant de ce système ne travaille que sur son contexte local.

Utilisation du contexte local des composants

Dans cet algorithme, nous avons pris comme hypothèse que le composant initiateur d'une modification pouvait demander des déplacements ou des déportations pour les composants avec lesquels il se trouve en conflit. Cela permet de ne pas avoir de contraintes trop fortes qui restreignent l'éventail des choix possibles.

Cependant, nous avons vu dans l'exemple que le composant A ayant pris comme décision de déporter D en D' provoquait un nouveau conflit entre D' et C'. Il est certain que l'utilisation du contexte local de D dans le choix de A aurait mené à une meilleure solution en déplaçant par exemple D vers le Sud, tout en conservant la topographie générale de la base, et sans créer de trou dans l'état final du projet. Ce principe pourrait être implanté dans le mécanisme de résolution, ce qui devrait permettre d'obtenir de meilleures solutions en général.

Malheureusement, pour pouvoir généraliser ce principe, il faut que les voisinages de composants soient remis à jour à chaque modification d'un composant. Cette opération est coûteuse et va à l'encontre de l'idée de base de cette stratégie. Il vaut donc mieux éviter de se servir du contexte local des composants pour résoudre un conflit dans cette stratégie.

Mémorisation des états de composants

Nous avons décidé dans notre algorithme de mémoriser les états successifs des composants au cours de la résolution en empilant l'enveloppe correspondant à cet état dans une pile spécialisée. On aurait pu également mémoriser simplement les actions permettant de passer d'un état à un autre, comme c'est par exemple le cas dans ARGOS-II [Farreny 81], NOAH [Sacerdoti 75], STRIPS [Nilsson 83]. L'intérêt de cette approche est de pouvoir revenir à tout instant à un état antérieur simplement et rapidement. Dans notre cas la mémorisation d'une action à exécuter est moins coûteuse en mémoire (une orientation + une distance) que celle d'une enveloppe (un rectangle), mais cependant n'est guère utile puisque l'on n'utilise pas le voisinage des composants pour trouver une action adéquate.

Nous n'avons donc pas pris cette option, mais cela peut se faire très simplement en mémorisant donc les actions permettant de transiter d'un état à l'autre, et en prévoyant un mécanisme de retour d'un état de composant au précédent lors des remontées ou de l'abandon des sous-branches de l'arbre de recherche. Cependant il faut toujours travailler sur la dernière enveloppe modifiée car on n'est jamais sûr qu'une solution intermédiaire aboutisse à une solution finale.

Expansion du schéma en cours de traitement

Il faut remarquer que cette stratégie pousse à l'expansion générale du schéma sur lequel on tente une résolution. En effet, lorsqu'un composant initiateur décide d'une action, il va sans doute déplacer ou déporter le composant qu'il modifie dans une zone proche de sa position initiale. Plus on s'enfonce profondément dans la résolution, plus on va enchaîner des actions tendant à expanser le schéma initial, d'où l'expansion génerale du schéma initial de travail.

Problèmes de cycle dans la résolution

Le fonctionnement de cet algorithme ne tient pas compte d'un phénomène génant pour la résolution : il s'agit des cycles dans la résolution. Nous désignons par le terme cycle une chaîne de résolution dans laquelle on réévalue une modification sur un composant déjà modifié précédemment dans le même sous-arbre de recherche. Un cycle pourrait être dans notre exemple : ((A —> A') —> (B —> B') —> (A' —> A") —> (B' —> B") ...). Ici A modifie B qui modifie a son tour A en remettant en cause la modification initiale sur A, A modifie ensuite B et remet en cause la modification précédente sur B ... . De tels problèmes, s'ils paraissent improbables à des niveaux bas dans la recherche, sont tout à fait envisageables si l'on descend profondément dans la résolution.

Ces cas ne sont pas difficiles à détecter, mais par contre très difficiles à traiter, car il n'est pas simple de savoir si une modification sur un composant déjà modifié facilitera l'aboutissement de la résolution ou au contraire sera néfaste à l'obtention d'une solution. Nous reparlerons à la fin de cette partie des problèmes de cycle.

Influence des heuristiques sur la résolution

Nous avons volontairement séparé dans l'algorithme la prise de décision de la résolution. Le choix d'un composant pour lequel il faut résoudre un conflit, d'une heuristique adaptée à ce cas , et la détermination d'une action à appliquer sur ce composant font partie de l'expertise vis-à-vis du domaine.

Or il n'est pas certain que l'on puisse séparer aussi radicalement l'expertise et la résolution. En effet, dans des domaines d'application comme la mécanique ou l'architecture, où les relations géométriques entre composants sont essentielles du point de vue fonctionnel du projet, le fonctionnement même de la résolution peut avoir une grande influence sur l'aboutissement à une solution. La recherche aurait par exemple parfois intérêt à être développée en largeur plutôt qu'en profondeur. Cette remarque est d'ailleurs valable pour d'autres domaines d'application où la géométrie du schéma a moins d'importance que dans ces deux domaines.

Le choix que nous avons fait est gratuit et mériterait également d'être étudié plus à fond dans un système où l'on voudrait réellement faire intervenir l'expertise du domaine. Néanmoins notre système vise essentiellement à résoudre les conflits géométriques sans être basé principalement sur une expertise du domaine d'application. C'est pourquoi dans notre étude nous avons bien séparé la partie de résolution de l'expertise du domaine.

III.5.3.7 Avantages et inconvénients de cette résolution distribuée

avantages

+ pas de prise en compte obligatoire du voisinage dans les mécanismes de base, ce qui permet une meilleure efficacité (car le contexte local n'est pas constamment remis à jour).

+ principe de fonctionnement assez proche du comportement de certains utilisateurs, permettant une élaboration incrémentale de la solution par une série de "tentative-puis-modification".

+ Si le composant initiateur d'une modification utilise des heuristiques fort évoluées qui tiennent compte de son contexte, on aura un meilleur contrôle de la résolution, en particulier on pourra garder un certaine cohérence de la base . Toujours dans le cas de notre exemple précédent, si A avait d'abord choisi de traiter le conflit avec D, puis celui avec C, il n'aurait pas rencontré la remise en cause (C' —> C") et n'aurait donc pas causé (indirectement) de trou dans la géométrie de la base.

inconvénients

— Les situations d'échec définitif peuvent être fréquentes, car le composant initiateur peut ne pas tenir compte du contexte local des composants en conflit pour décider des actions à réaliser. La stratégie décrite ici implique une multiplication des cas de conflit.

— les voisinages, ignorés dans la structure nécessaire à la gestion distribuée de la résolution, doivent être obligatoirement gérés correctement si le composant initiateur d'une modification veut tenir compte du contexte des composants en conflit pour choisir l'action à réaliser. C'est un mécanisme coûteux qui nécessitera leur réactualisation et leur mémorisation régulières dans la pile d'états temporaires (à chaque modification au moins).

— Les remontées (backtracking) seront nombreuses, surtout s'il n'y a pas de prise en compte du contexte local des composants.

III.5.3.8 Implémentation distribuée de l'algorithme

L'algorithme de résolution que nous avons décrit au paragraphe III.5.3.2. permet de répartir la résolution sur tous les composants du projet. Chaque composant exécute cet algorithme indépendamment des autres. Cependant, son fonctionnement est séquentiel puisque l'on suppose toujours recevoir une réponse immédiate d'un message envoyé à un autre composant. Ceci est contraire au principe de fonctionnement distribué du système. Si l'on peut dire que la résolution totale du problème est distribuée parmi les composants qui

exécutent chacun un algorithme de résolution avec leur propre base de fait (situation de conflit) et base de connaissance (expertise implantée et contexte du composant), le déroulement proporement dit de la résolution reste séquentiel.

Nous avons voulu ici proposer une version de cet algorithme qui puisse fonctionner réellement en milieu distribué. On supposera ici que chaque composant est associé à un processeur et à un programme exécutant l'algorithme que nous décrivons ci-après. A chaque programme en cours de déroulement est associée une file de messages reçus.

Le principe de fonctionnement du programme est de prendre systèmatiquement le message se trouvant en tête de file et de le traiter immédiatement. S'il n'existe pas de message, le programme réessaie régulièrement d'en extraire un. Un tel programme est "instancié" sur le processeur à chaque activation du composant. Ceci implique que si un composant est activé plusieurs fois au cours de la résolution, il y aura plusieurs instances de programme associées à ce composant qui se dérouleront dans le système.

On suppose également qu'il existe un mécanisme de routage des messages parmi les composants.

Le déclenchement d'une résolution se déroule en deux étapes : une partie séquentielle très simple qui initialise le système (variables globales + routage des messages), et la première activation pour un composant manipulé par l'utilisateur.

Le format des messages est le suivant :

- pour un envoyeur de message : (nom-message, paramétres, destinataire).
- pour le receveur du message : (nom-message, paramètres, envoyeur).

Un envoyeur de message n'attend pas de réponse.

Partie séquentielle

- *liste-des-composants-modifiés* <— ().
- *composant-initial* <— composant-manipulé.
- *solution-trouvée* <— faux
- initialiser le système.

Partie parallèle

niveau global

- *stopper-résolution* :
  si *solution-trouvée*
  - **pour** tous les composants de *liste-des-composants-modifiés*, leur demander de valider leur dernier état.
  - **pour** tous les composants de *liste-des-composants-modifiés*, leur demander de recalculer leur voisinage.
  - remettre à jour le schéma du projet avec les nouveaux états des composants.

sinon

a. demander à tous les composants de *liste-des-composants-modifiés* de réinitialiser leur état (revenir à l'état initial de la résolution)
b. signaler l'échec de résolution à l'utilisateur.

niveau composant

extraire un message : nom-message, paramètres, envoyeur.

- *résoudre-action* :
  - *initiateur* <— envoyeur du message.
  - calculer la nouvelle enveloppe et l'empiler sur la pile d'états temporaire.
  - ajouter *self* à la *liste-des-composants-modifiés.*
  - déterminer la *liste-des-composants-en-conflit*
  - **si** *liste-des-composants-en-conflit* = (),
    alors
    — oter*self* de la *liste-des-composants-modifiés.*
    — envoyer (*reussite*, , *initiateur*).
    sinon
    - **pour** les composants C de *liste-des-composants-en-conflit*
      — suivant expertise déterminer une action à résoudre *action.*
      — **si** aucune action possible,
      — envoyer (*échec*, , *self*).
      fsi.
      — envoyer (*résoudre-action, action, C)*

    fsi

- *réussite* :
  - rajouter *envoyeur* à la *liste-des-conflits-résolus.*
  - oter *envoyeur* de *liste-des-composants-en-conflit.*
  - **si** *liste-des-composants-en-conflit* = (),
    - **si** *self* = *composant-initial*,
      alors
      – *solution-trouvée* = vrai.
      —envoyer (*stopper-résolution*),
      sinon envoyer (*reussite*, , *initiateur*),
      fsi.

- *échec* :
  - **pour** les composants de *liste-des-conflits-resolus*
    — leur demander d'annuler leur dernier état temporaire.
  - **pour** les composants de *liste-des-composants-en-conflit*
    — leur demander de stopper leur résolution courante.
  - annuler dernier état temporaire.
  - **si** la pile d'états temporaires est vide,
    — oter *self* de *liste-des-composants-modifies*
    fsi
  - envoyer (*échec, , initiateur).*

Le principe de fonctionnement de l'algorithme est le suivant :

1. Initialiser les variables globales nécessaires à la résolution : la *liste-des-composants-modifiés*, le *composant-initial*, le booléen *solution-trouvée.*

2. lancer la résolution sur le composant manipulé par l'utilisateur, en lui envoyant le message (résoudre-action, action-utilisateur, composant-manipulé). Ce composant détermine les conflits en cours, cherche des actions à réaliser pour résoudre les problèmes. S'il n'y pas de conflit, il renvoie directement le message *réussite* au niveau global de la résolution pour terminer celle-ci proprement; et on prévient l'utilisateur de la réussite (la base et les voisinages sont remis à jour). Sinon pour chaque action à réaliser, il enverra un message *résoudre-action* au composant concerné. Si pour un composant donné il n'y a aucune action envisageable, il renvoie un échec au niveau global qui termine proprement la résolution et prévient l'utilisateur..

3. A ce moment, tous les composants qui vont être "touchés" par la résolution peuvent recevoir principalement trois messages :
   a. *résoudre-action* : Il détermine alors les conflits, initialise les variables locales à la résolution courante (*initiateur, liste-des-composants-en-conflits, liste-des-conflits-résolus),* et essaie de trouver une action pour chaque composant en conflit. Il agit en fait comme l'a fait précédemment le composant manipulé par l'utilisateur, mais à un niveau de plus en profondeur de recherche.
   b. *réusssite* : l'un des composants en conflit a réussi à résoudre son problème. on agit en conséquence.
   c. *échec* : l'un des composants en conflit a échoué dans sa résolution, il faut éliminer cette branche de résolution immédiatement et remettre correctement à jour l'état des composant sà ce niveau de recherche.

Cet algorithme ne tient pas compte des problèmes éventuels de synchronisation des messages envoyés aux composants, qui pourraient apparaître au cours de la résolution, ni de la façon dont sont envoyés ces messages. C'est en fait un algorithme très naïf qui n'a d'autre but que de donner un aperçu de ce que pourrait être une résolution parallèle dans ce contexte, ainsi que d'introduire les principaux problèmes que l'on rencontrera dans ce type de fonctionnement de la résolution. De plus, une partie des délégations demandées aux autres

composants (*valider un état, recalculer un voisinage, annuler le dernier état empilé, stopper la résolution courante)* devrait être traitée comme des messages, alors qu'ici on ne précise pas le mode d'exécution d'une telle délégation.

Cette implémentation parallèle simpliste de l'algorithme initial de résolution montre certains problèmes importants de la résolution en parallèle :

1. Si l'ordre dans lequel un composant émet et reçoit un message est important, comment garantir que la séquence de réception de messages par un composant est la même que celle de l'émission de ces messages. Si un message peut modifier l'état d'un composant et que cet état a un rôle important dans le déroulement de la résolution (ce qui est a priori le cas), alors il faudra garantir le même ordre dans la séquence d'émission et dans la séquence de réception.

2. Toute délégation doit être traitée comme un message, aussi rudimentaire soit il. Sinon il n'y aurait aucun avantage en efficacité de la résolution à attendre d'un tel mécanisme parallèle de résolution

3. Il faut être capable de gérer complétement les diverses branches de l'arbre de recherche parcourues en parallèle. Ceci implique que pour chaque instanciation d'un programme exécutant l'algorithme de résolution, il faut mémoriser l'état complet du composant pour ce programme.

4. L'intérêt essentiel d'une implémentation parallèle de l'algorithme étant d'obtenir une meilleure efficacité dans la recherche d'une solution, il faut construire des algorithmes parallèles efficaces. Celui que nous proposons ici n'a pas cette prétention.

En conclusion, nous pouvons dire que notre modèle permet de distribuer la résolution entre les composants, mais qu'une implémentation d'un algorithme de résolution fonctionnant d'une manière parallèle n'en sera pas pour autant facilitée, et qu'elle devra être étudiée très finement.

Cette stratégie qui consiste pour le composant maître à construire une action à réaliser par les composants avec lesquels il se trouve en conflit est primitive car elle fait complètement abstraction du contexte local des composants en conflit. Les concepteurs utilisent souvent cette stratégie qu'on pourrait qualifier par "*agir, puis constater et déléguer*" dans des situations où la place du composant déplacé est la plus déterminante. Cependant, dans bien des cas les concepteurs réfléchissent à "un niveau de profondeur de résolution" bien plus élevé et considèrent avant d'agir la situation (l'état) des composants environnants. La stratégie que nous présentons ci-après fait référence à ce type de raisonnement en le simulant partiellement.

III.5.4 Stratégie par tentative, puis délégation

III.5.4.1 Description

Le fonctionnement de l'algorithme de résolution basé sur cette stratégie est le suivant :

- Un composant reçoit une requête de résolution d'un autre composant, ce dernier lui donnant également des informations sur le conflit géométrique concerné. Ce conflit implique l'émetteur de la requête et le récepteur.

- Le récepteur cherche la meileure solution possible en tenant compte de son contexte local (voisinage) et des heuristiques dont il dispose.

- Si, après avoir élaboré cette solution, il reste des conflits avec d'autres composants, il leur délégue le travail de trouver une solution pour les nouveaux conflits apparus. C'est à dire qu'il leur envoie une requête de résolution comme celle reçue par lui-même précédemment.

Remarques :

- Dans cette stratégie, le voisinage des composants est très utile, car pour élaborer sa solution, le composant doit tenir compte au moins de son contexte local. Par exemple, pour pouvoir décider d'un déplacement vers une zone libre de son contexte local, il devra examiner quel est celui-ci en interrogeant ses voisins.

- Cette stratégie nécessite une remise à jour constante des voisinages des composants de la base de projet à chaque fois qu'un composant a décidé d'un déplacement ou d'une autre action (déportation). En particulier, le composant actif (celui qui décide d'une action à réaliser) doit forcer ses voisins à modifier leur contexte local.

- C'est également une stratégie de recherche "en profondeur d'abord", mais cette fois-ci en tenant du voisinage d'un composant avant de le déplacer.

III.5.4.2 Algorithme de résolution associé

Nous décrivons ici de manière tout à fait informelle un algorithme de résolution que nous avons développé à partir de cette stratégie. Comme dans la stratégie précédente, le composant exécutant est le recevuer du message de demande de résolution.

Le composant receveur reçoit une requête de résolution avec les informations suivantes :

- la nouvelle enveloppe de l'émetteur du message.

- une liste de composants avec lesquels l'émetteur est à présent en conflit.

- Le conflit qui concerne directement le receveur, C'est-à-dire l'intersection entre l'enveloppe du receveur et la nouvelle de l'émetteur.

Le receveur R de la requête répète alors les actions suivantes :

— Chercher une action (un déplacement, une déportation, ...) à réaliser pour éliminer le conflit. Il détermine cela en analysant la situation précédente ainsi que l'état courant de son voisinage. Il pourra ici utiliser des heuristiques ainsi que l'expertise du domaine pour arriver à ses fins. De toute façon, il choisira une solution qui lui paraîtra la moins mauvaise, sinon la meilleure.

— Il détermine alors sa nouvelle situation à la suite de l'action choisie. C'est-à-dire sa nouvelle enveloppe ainsi que les nouveaux conflits avec d'autres composants, engendrés par la nouvelle situation.

— tant qu'il y aura des conflits à résoudre et qu'aucun composant en conflit ne signale d'échec complet, il exécutera les actions suivantes :

- choix d'un composant en conflit avec lui.
- envoi de la requête de résolution à ce composant choisi.
- si ce dernier composant ne trouve aucune solution pour ce conflit, il faut abandonner la résolution pour cette nouvelle situation. Les composants qui ont trouvé précédemment une solution à leur conflit avec R doivent revenir à leur état précédent (de même que les voisins qui ont vu leur contexte local transformé), ainsi que leurs "descendants" dans la résolution.

  sinon mémoriser que ce composant a trouvé une solution tout en vérifiant qu'elle ne provoque pas de nouveaux conflits sur des solutions précédentes dans la résolution.

— Si tous les composants en conflit avec R ont pu résoudre leur conflit, mettre à jour le voisinage de R (dans sa nouvelle position) ainsi que l'état courant de la base de projet.

jusqu'à avoir trouvé une solution pour tous les composants en conflit ou avoir essayé toutes les heuristiques connues applicables à la situation de R.

Si tous les conflits ont pu être résolus, on renvoie "réussite" à l'émetteur, sinon "échec".

Quelques remarques préliminaires sur cet algorithme :

- La mise à jour du voisiange de R implique l'empilement du nouvel état de R avec son enveloppe modifiée ainsi que son voisinage modifié. Il faut recalculer le voisinage de R avec sa nouvelle enveloppe R' et l'enveloppe modifiée de l'envoyeur.

- De même, pour revenir au dernier état précédent, c'est-à-dire remettre en cause une hypothèse faite par un composant, il faut non seulement dépiler le dernier état du composant, mais aussi faire de même avec les composants qui étaient en conflit avec R et qui avaient trouvé une solution. La remise en cause concerne en fait une branche complète de l'abre de résolution et il faut aller jusqu'à la profondeur maximale dans l'arbre des solutions partielles trouvées pour y remettre à jour les composants

concernés par ces résolutions. Il faut donc prévoir de tels mécanismes distribués de remise en cause parmi les composants. Nous avons déjà rencontré ce problème lors de la stratégie précédente. C'est une caractéristique essentielle impliquée par cette stratégie.

III.5.4.3 Structures de données et commentaires généraux sur cet algorithme

Nous prendrons ici une structure de composant similaire à celle que nous avions choisie dans la stratégie précédente, mais avec une pile d'états temporaires plus riche en informations. Celle-ci doit contenir :

- la nouvelle enveloppe de R.
- le voisinage correspondant dans l'état actuel de la base de projet et pour cette nouvelle situation.
- les conflits observés pour cette nouvelle situation.

La nouvelle enveloppe de R, qui est le résultat du choix de R mémorise le nouvel état géométrique de R. De plus, lors des recalculs de voisinage, on se référera à cette enveloppe, puisqu'elle représente l'état du composant au moment de cette nouvelle situation.

Le voisinage correspondant à cet état du composant (hormis les conflits), permet aux composants qui doivent faire des hypothèses, un choix d'action, de connnaître le contexte local d'un composant dans l'état modifié de la base. Ainsi les heuristiques de choix d'action peuvent fonctionner correctement sur l'état hypothétique modifié du projet.

Mémoriser les conflits observés pour cette hypothèse permet, comme il a été dit plus haut, de prévoir et construire des mécanismes de remise en cause distribué pour les descendants d'un composant détectant un échec. On peut alors déléguer la remise en cause parmi ces "descendants". Le terme "descendant" d'un composant signifie ici un composant touché par la résolution après le composant concerné.

Voici un exemple d'un composant A au cours d'une résolution, touché deux fois par celle ci :

A
EA
P
(EA1, (VE1, VO1, VN1, VS1), (C11, ..., C1N)) (EA2, (VE2, VO2, VN2, VS2), (C21, ..., C2N))

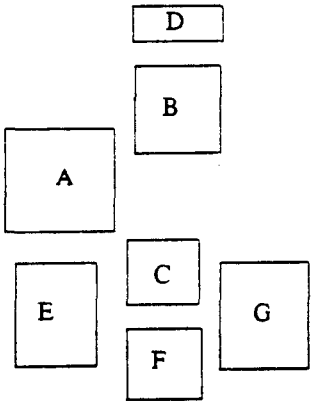
Les mises à jour de voisinage doivent se faire dès l'instant où une nouvelle situation est créée par un composant R ayant fait un choix. Il est nécessaire que les descendants de R dans la résolution puissent recalculer leurs voisins avec la nouvelle enveloppe de R. Avant donc de transmettre la situation en hypothèse à ses descendants, R doit empiler sa nouvelle enveloppe et son voisinage correspondant (pour les composant qui ne sont pas en conflit), afin que ses descendants puissent eux-même évaluer correctement le leur et déduire des actions sur un état cohérent, mais hypothétique, de la base de projet.

En fait, il est indispensable qu'au moment où un composant décide d'une action, il puisse le faire sur l'état actuel de la base, d'où l'importance de conserver constamment un état cohérent de la base, sauf pour les composants en conflit pour lesquels la notion de voisin comme nous l'avons définie n'a plus de sens.

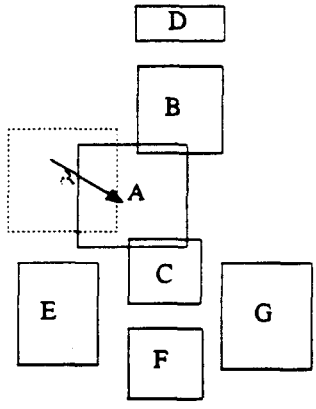
III.5.4.4 Application à un exemple

Nous décrivons ici, d'une manière aussi informelle que pour la description de l'algorithme, l'application de cette stratégie suivant cet algorithme de résolution sur un exemple simple. Le but est de donner un aperçu de la manière dont se déroulerait une résolution suivant cette stratégie.

Nous supposons que nous avons l'état initial suivant pour la base de projet :



L'utilisateur tente une déportation "Sud-Est" de A, conduisant a la situation montree sur le schéma de droite.



Après l'action de l'utilisateur sur le composant A, on a une situation conflictuelle entre A et B, ainsi qu'entre A et C. On suppose maintenant que A exécute l'algorithme décrit pour la stratégie par tentative puis délégation. Nous montrons ici le déroulement de la résolution comme nous l'avons fait lors de la précédente stratégie.

0 A reçoit l'ordre de déportation de l'utilisateur et en déduit son nouvel état A' et les deux conflits avec B et C.

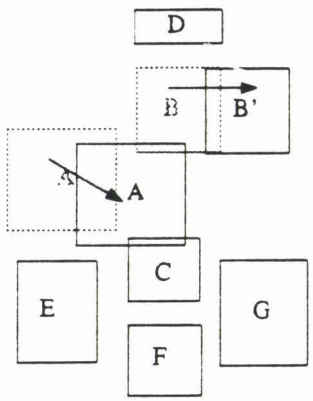
a A empile ces informations sur la structure décrite précédemment.

b A choisit, suivant les heuristiques de choix de composant et l'expertise du domaine, un premier composant parmi (B, C). Supposons que ce composant soit B.

c A envoie la requête de résolution à B, et lui délégue la résolution de ce conflit.

1 B reçoit la requête de A.

a B choisit, suivant les heuristiques et l'expertise du domaine, un déplacement Est pour éviter le conflit; ceci mène à la situation B' montrée ci dessous.



a B met à jour son voisinage par rapport à l'état courant de la base.

b B empile les informations suivantes : (EB', (, A, D, (C, G, F)), ()). C'est à dire la nouvelle enveloppe B', les voisins de B', les composants en conflits dans cette situation.

c B provoque un recalcul de voisinage à empiler pour D, C, G, F, A.

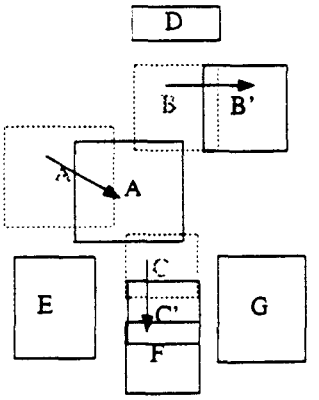
d B renvoie "réussite" à A

0 A choisit ensuite le composant C (dont le voisinage est modifié et empilé) et lui délégue également la résolution du conflit.

1 C reçoit la requête de A. Nous présentons ici deux éventualités dans la suite de la résolution.

Cas1

a C choisit un déplacement vers le "Sud" pour éviter le conflit entre A' et C. Ceci mène à la situation B' montrée ci dessous.



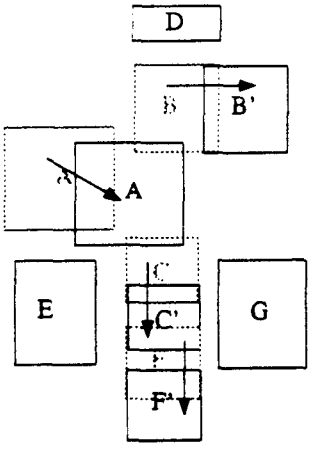
b C calcule alors son voisinage modifié et l'empile en ignorant F qui est en conflit avec C', sa nouvelle enveloppe C', et les nouveaux composants en conflit, c'est-à-dire ici F. La liste empilée est la suivante : (EC', (G, E, (A', B', D),), (F)).

c C provoque un recalcul de voisinage à modifier et à empiler pour G, E, A', B', D.

d C' est en conflit avec F. Il envoie la requête de résolution à F, et lui délégue la résolution de ce conflit.

2 F reçoit la situation donnée par C.

a F choisit un déplacement vers le Sud pour éviter le conflit. Ceci mène à la situation décrite ci-dessous.



b F met à jour son voisinage par rapport à l'état courant de la base

c F empile son nouvel état par rapport à l'état actuel de la base : (EF', (G, E, (C', B', D'),),).

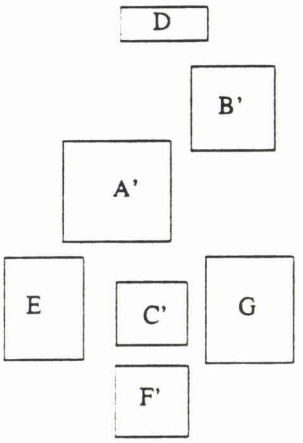
d F provoque un recalcul de voisinage à empiler et modifier pour G, E, C', B' ,D.

e F renvoie "réussite" à C.

1 Il n'y a plus de conflit pour C, C renvoie "réussite" à A.

0 Il n'y a plus de conflit pour A. Il remet à jour son voisinage avec ce nouvel état. Il provoque également une remise à jour de tous les composants de la base touchés par la résolution en demandant aux composants qui étaient en conflit avec lui (B, C) de faire ce travail. B le fera sur son état, puis C demandera à F de remettre à jour son voisinage, avant de remettre le sien à jour. La remise à jour est distribuée parmi tous les composants.

L'état final de la base est alors le suivant :



Cas2

a C décide qu'il n'y pas de solution valable et renvoie "échec" à A.

0 A reçoit ce message d'échec.

a A abandonne sa première tentative, il y a remise en cause.

b A signale à B qu'il doit annuler son état. B dépile son dernier état empilé et transmet à ses voisins ce nouveau chnagement de situation.

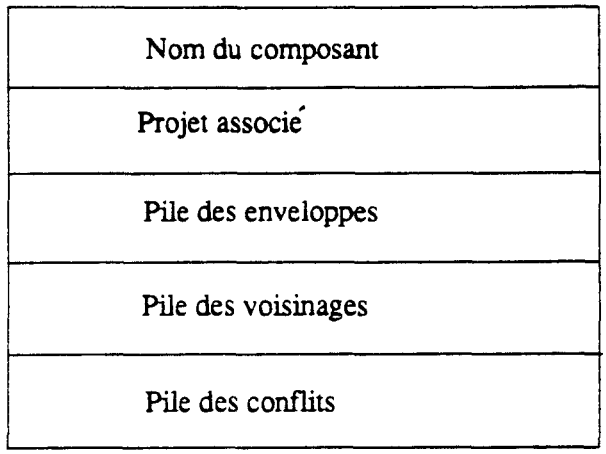
c A renvoie "échec" . L'utilisateur est prévenu que le conflit n'a pas été résolu. La base est restée dans son état initial.

Remarques :

Dès qu'un composant recalcule son voisinage suite à une hypothèse particulière, il faut, puisque ses voisins ont changé, que ceux-ci remettent à jour leur état de voisinage et l'empile. mais c'est la seule chose que ceux-ci doivent empiler puisqu'il n'y a pas eu de modification de leur enveloppe et pas de conflit non plus. Ceci entraine qu'au cours de la résolution, les piles d'état temporaires contiennent des éléments de contenu différents : un voisinage empilé ou une situation empilée suivant les cas.

La gestion des piles d'états temporaires est malaisée. Il vaut mieux tout simplement répartir les piles sur les attributs des objets composants. On aura alors, dans notre cas où seuls

l'enveloppe, les voisins, et les conflits peuvent être modifiés, une structure de composant différente :



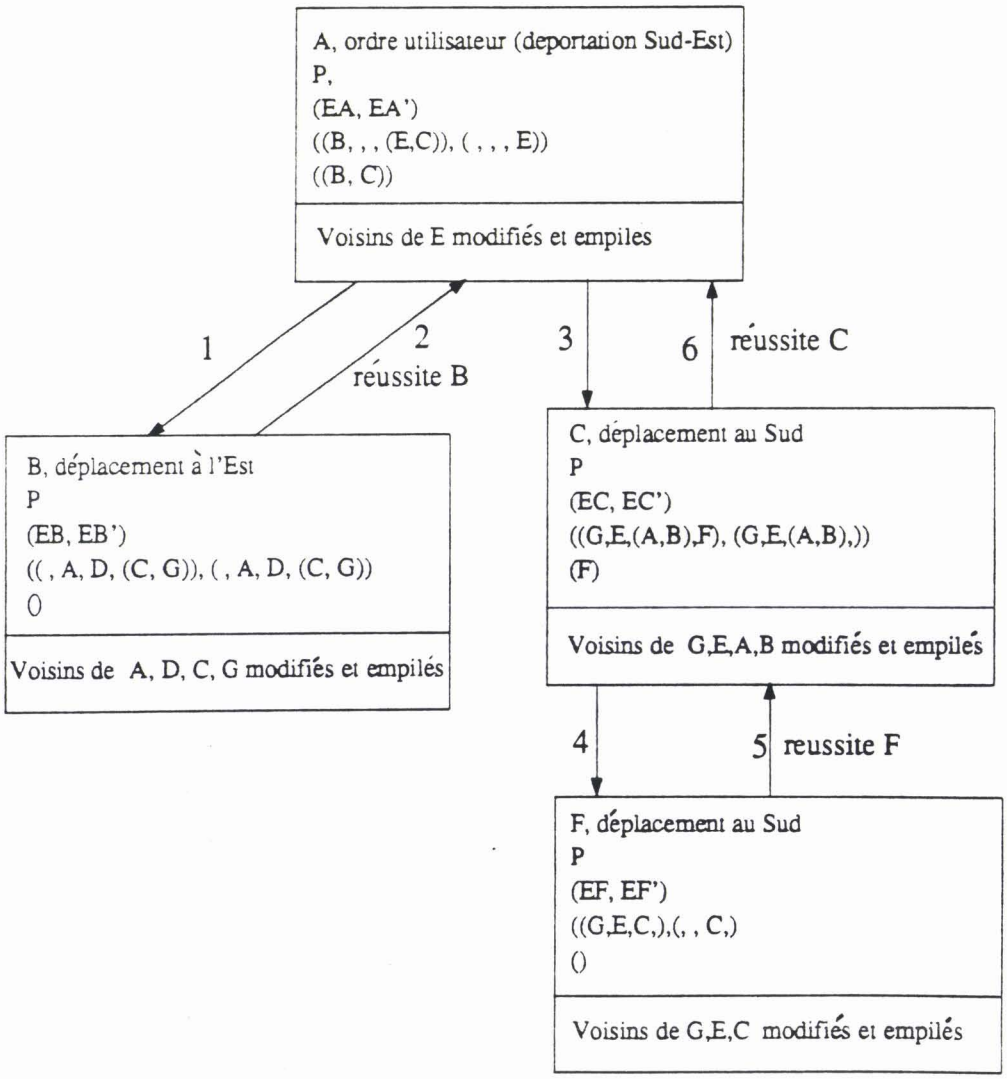
Dans tous les cas, le sommet de chacune des piles fait référence au dernier état cohérent de la base. Cette structure est plus adaptée aux diverses modifications qu'on apportera aux composants au cours de la résolution.

De la même manière que pour la remise en cause en cours de résolutiuon, il faut prévoir lors des recalculs de voisinage un mécanisme qui permet l'empilement des voisinages . Les algorithmes de calcul de voisinage vus dans la partie algorithmique modifient le voisinage d'un nouveau ou d'un ancien voisin, car c'est une relation symétrique. Il faut que le composant qui recalcule son voisinage provoque d'abord pour ses voisins un empilement de voisinage; on pourra alors utiliser normalemernt les algorithmes de mise à jour de voisinage sur ce dernier état des voisins.

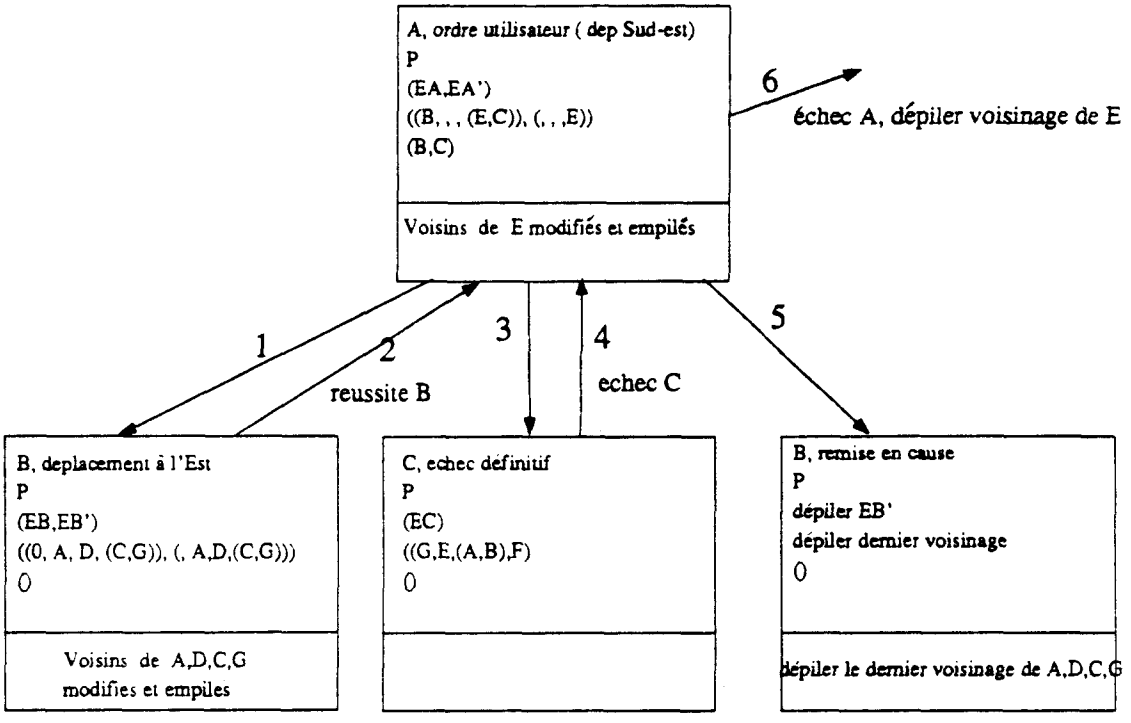
Représentation arborescente de la résolution

Nous pouvons également, pour fournir une meilleure vision de la résolution, la représenter sous forme d'arbre. Dans cet arbre nous indiquerons dans les noeuds la description de l'état des composants directement modifiés, et des indications sur ceux qui le sont indirectement. Les modifications de voisinage sont plus difficile à représenter, c'est pourquoi nous n'en donnons ici que des indications.

Dans le premier cas de résolution, nous obtenons l'arbre suivant :



Dans le deuxième cas, nous obtiendrons l'arbre suivant :



Remarque : la remise en cause doit être faite dans l'ordre inverse de la résolution pour être sûr que le voisinage dépilé sur un composant est le bon.

III.5.4.5 Remarques générales sur cet exemple

Cet exemple nous a permis de mettre en évidence la nécessité de distribuer certains mécanismes classiques des moteurs d'inférence. Il faut le faire pour :

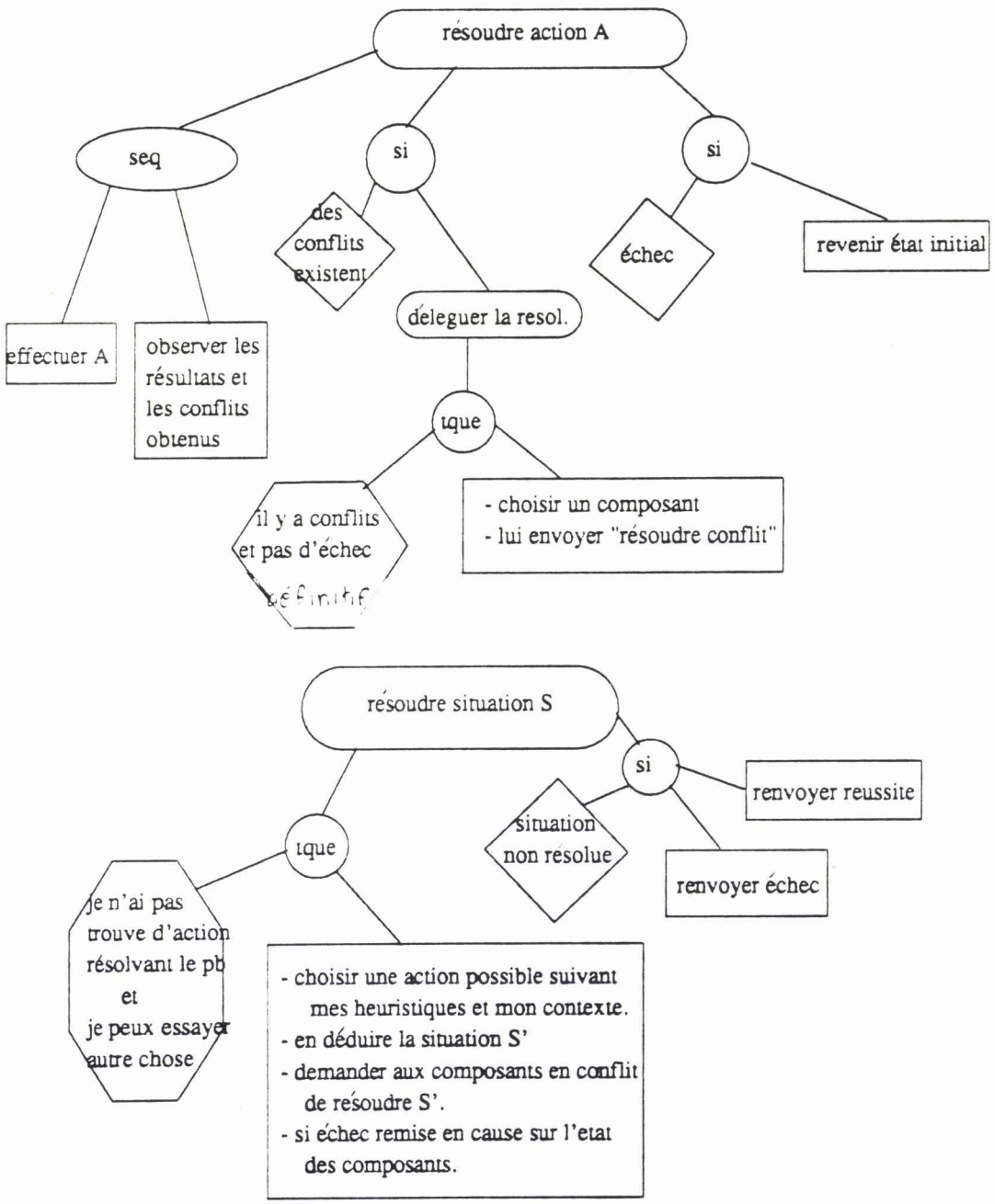
- la résolution (distribuer la prise de décision)
- la remise en cause (la déléguer aux descendants)
- la mémorisation du contexte (état de la base distribué parmi les composants).

Les heuristiques de choix de composant et d'hypothèse de modification sont dans cette stratégie des éléments cruciaux pour obtenir une bonne résolution. Du choix de composant en conflit et de la prise en compte du contexte local (ou moins local) vont dépendre la qualité de solution trouvée et l'efficacité de la résolution. Il sera sans doute nécessaire de représenter l'expertise du domaine d'une manière adaptée aux problèmes locaux.

Le mécanisme présenté ne peut fonctionner correctement que d'une manière séquentielle : en effet on distribue le contexte global sur des piles accrochées à chaque composant en supposant que les manipulations de contexte ne s'opéreront que sur le sommet des piles (supposé être le dernier état de la base). Ceci implique un fonctionnement séquentiel de la résolution, mais cependant distribuée du point de vue de la décision parmi les composants.

III.5.4.6 Discussion sur l'algorithme

Chaque composant dispose des mêmes capacités algorithmiques et des connaissances liées à leur contexte local. Nous pouvons décrire sous forme d'arbre programmatique le fonctionnement de la résolution. Les deux arbres correspondants sont :



Comme pour la stratégie précédente, l'expertise se situera dans les choix de composants qui doivent résoudre une situation de conflit, cette fois en tenant compte de leur contexte local. Les heuristiques à utiliser dans ce cas doivent être beaucoup plus riches en ce qui concerne la prise en compte du contexte local du composant pour obtenir rapidement une solution.

On remarque une indépendance importante entre la partie algorithmique et la partie "expertise". Au niveau de la décision d'actions à tenter, toutes les techniques de représentation des connaissances peuvent être envisagées a priori. Cependant celle-ci devra être distribuée parmi les composants et indépendante de la gestion des états de la base.

Avec un tel mécanisme, l'introduction de contraintes fonctionnelles doit se faire dans la prise de décision, car la résolution décrite ci-dessus suppose qu'un composant agit seul; ce qui est insuffisant car dans beaucoup de cas, la liberté d'action des composants est limitée également par les contraintes fonctionnelles dues à d'autres composants. Une solution serait peut-être de proposer aussi un voisinage fonctionnel.

Par rapport à la stratégie précédente, le fait de vérifier une hypothèse :

- complique les mécanismes de prise de décision (parce qu'il faut prendre en compte le contexte).
- diminue la fréquence des échecs définitifs (car on peut faire des hypothèse de bonne qualité).
- améliore la qualité de la solution trouvée.
- se rapproche encore plus de la manière 'naturelle' de résoudre les conflits.

III.5.5 Evaluation des deux stratégies présentées

III.5.5.1 Remarques générales

Le deux mécanismes de résolution étudiés ici nous ont permis de mettre en relief les problèmes que l'on rencontre si l'on veut dans un système de CAO interactif, introduire certaines capacités intelligentes d'aide pour l'utilisateur, dans le cas d'un système interactif de placement spatial.

Nous avons étudié particulièrement des algorithmes permettant d'envisager la résolution et la remise en cause d'une manière distribuée parmi les agents géométriques d'un système multi-agents. Bien que le déroulement de ces mécanismes reste strictement séquentiel, ils sont cependant distribués parmi les composants (graphiques ou fonctionnels) constituant le dessin et le contenu du projet de l'utilisateur. Remarquons encore une dernière fois quue nous avons pris délibérément le choix de distribuer tout le fonctionnement du système parmi les composants du projet, ceci guidant et conditionnant forcément une grande partie des choix que nous avons faits .

Les limitations volontaires que nous avons faites pour simplifier l'étude portent au niveau des composants sur les éléments suivants :

— Seul un type (plus exactement une forme) de composant est manipulé : le rectangle.

Ceci est trop limitatif si l'on veut appliquer le travail à d'autres domaines de CAO que la conception de circuits imprimés. De plus, même pour des circuits imprimés, nous n'avons pas étudié le cas des connexions entre composants ni celui des plans de masse qui sont des éléments importants de ce domaine.

— Nous ne nous sommes préoccupés que des contraintes géométriques ou topologiques, elle-mêmes limitées au cas des intersections ou recouvrements possibles entre rectangles. Les contraintes fonctionnelles ou sémantiques jouent un très grand rôle dans la conception d'un schéma, quel que soit le domaine d'application abordé.

— Lors de la résolution, nous avons considéré comme indépendants entre eux le fonctionnement de la résolution et celui de la représentation des connaissances et de l'expertise du domaine. Autrement dit, la stratégie de résolution adoptée, qui ici était en profondeur d'abord, ne peut être affectée par l'expertise que par le choix d'un composant auquel on demande de continuer la résolution. Il serait bon que l'on prévoie des reconnaissnces de situation (ou des métarégles) pour mieux orienter la stratégie de résolution dans ces cas particuliers.

Nous pouvons remarquer que les deux stratégies étudiées tendent à l'expansion de la topographie des composants sur la carte. Ceci est une conséquence directe du problème spatial. En effet, lorsqu'un composant se trouve en conflit avec un autre, deux types de façon de résoudre le problème existent :

1. On peut décider de déporter très loin le composant avec lequel on se trouve en conflit, si possible dans une zone clairsemée de l'espace, pour avoir de grandes chances d'éviter un conflit. Cette façon de faire risque de donner des effets indésirables : les composants proches qui pouvaient l'être pour des raisons sémantiques risquent de se trouver complétement déconnectés géométriquement l'un de l'autre, d'où une perte importante de la cohérence sémantique de la base. Un tel procédé de résolution peut donner des résultats de tous ordres, aussi bien en expansion qu'en contraction.

2. L'autre solution consiste à considérer que deux composants se trouvent l'un à coté de l'autre pour des raisons précises. On doit donc si possible respecter cette proximité. Dans ce cas, on déplacera ou déportera les composants en situation conflictuelle d'une très petite quantité. On provoque ainsi naturellement une "vague de déplacements" vers l'extérieur du conflit initial, menant donc à une expansion de la topographie des composants. C'est exactement de cette manière que nous procédons dans les deux stratégies étudiées, d'autant plus que les voisinages sont bâtis sur une notion géométrique de visibilité, impliquant la proximité (relativement à la zone locale) pour les premiers voisins. Comme nous faisons référence aux voisins pour déterminer une action, il est normal de constater un phénomène d'expansion.

III.5.5.2 La prise de décision

Du point de vue de la prise de décision, nous avons vu que , au cours de la résolution, si nous tenons compte du contexte des composants, il est nécessaire de mémoriser de façon distribuée les états successifs de la base. Il faut remarquer que la structure de données choisie pour le contexte d'un composant, à savoir des voisins latéraux à l'Est, l'Ouest, le Sud, le Nord est trop pauvre pour déterminer des choix efficaces pour la résolution. En effet, avec une relation de voisinage non isotropique, on ne sera pas capable de se *déplacer* en diagonale, d'où une limitation importante des choix de mouvements lors de la résolution. Les qualités incomplètes du voisinage que nous avons défini affaiblissent le potentiel de résolution des composants. Ceci était tout à fait prévisible, une vue imprécise du monde ne pouvait que diminuer les capacités de résolution d'un composant.

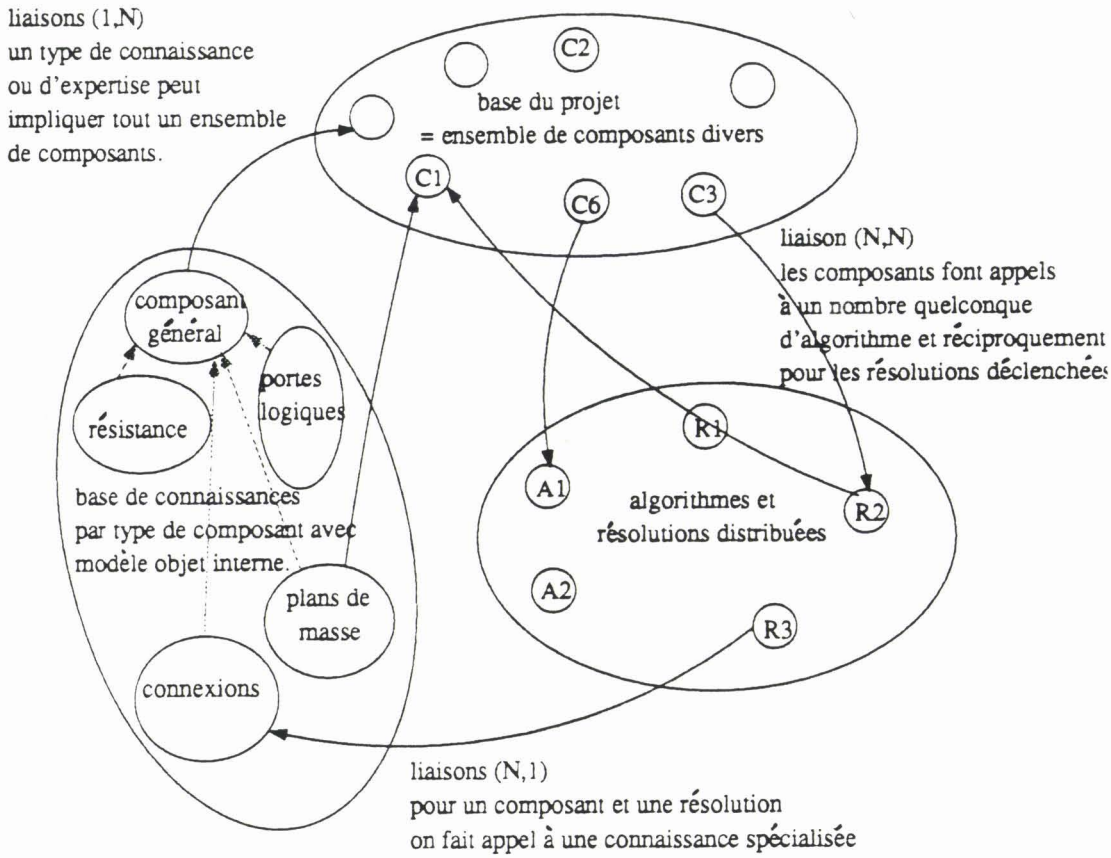
Décider de déporter un composant nécessite sa destruction puis sa recréation à un autre endroit et donc conduit à redéterminer deux fois des ensembles de voisins pour certains composants de la base. La déportation au cours d'une résolution est donc un mécanisme coûteux qui devra être évité si l'on veut obtenir des temps raisonnables pour résoudre les conflits. L'expertise devra donc donner la priorité dans la majorité des cas à un déplacement plutôt qu'à une déportation.

Cependant, le domaine d'application de notre interface, c'est-à-dire la conception de circuits imprimés, ne nécessite pas une définition géométrique très fine du voisinage. Ce ne sera pas le cas des systèmes de CAO en mécanique ou en architecture, où les liaisons géométriques entre composants ont une signification importante. Dans ce cas, la prise de décision sera assez étroitement liée non seulement au problème spatial initial, mais aussi à la façon d'agir sur chacun des composants en conflit, donc à la stratégie de résolution elle-même. On peut donc dire que dans le cas général, il faut que la prise de décision soit liée à la stratégie de résolution. Un algorithme de résolution incorporant plusieurs stratégies de résolution serait donc le bienvenu pour pouvoir tenir compte efficacement des contraintes fonctionnelles et sémantiques.

III.5.5.3 Gestion et organisation de l'expertise

Nous avons volontairement éludé dans notre étude le problème de la représentation des connaissances et l'expertise du domaine. Nous ne pouvons donc pas évaluer notre modèle de résolution sur ce critère.

Nous pouvons cependant proposer une organisation très générale des connaissances. Le modèle proposé se veut indépendant des connaissances, on pourrait donc étudier l'implantation spécifique de tout type de représentation existante. Nous proposons, toujours pour conserver une approche distribuée et un modèle objet (ou acteur), l'organisation suivante pour l'ensemble du système :



Sur ce schéma, on voit qu'un projet est constitué d'un ensemble de composants divers ayant chacun un type (résistance, condensateur, ...) et fera donc appel pour les prises de décisions comme pour son comportement à une base de connaissance associée. Une représentation des connaissances de type objet [biblio] ou frame [biblio] ou multi-représentation [biblio] serait à ce sujet la bienvenue nous semble t-il. En cours de résolution , les composants en conflit peuvent faire appel à un nombre quelconque d'algorithmes de résolution, chacun de ces algorithmes implémentant une stratégie particulière de résolution. Le choix d'une stratégie peut se faire grâce à des heuristiques où des métarègles, mais l'utilisation d'algorithmes différents au cours d'une même résolution n'a pas été traité dans ce travail et reste un problème ouvert (tout du moins en cas de fonctionnement distribué).

Un modèle objet pour les connaissances non liées à la géométrie seule serait sans doute bien adapté au modèle multi-agents que nous avons choisi. Il faut également que le modèle utilisé pour décrire les connaissances tienne compte du fait que la résolution sera distribuée. Les composants eux-même sont des acteurs qui coopèrent entre eux, et les algorithmes de résolutions répondent à des besoins spécifiques de la stratégie courante de résolution, et conservent toujours un état cohérent de la base de projet du point de vue des voisinages de composant.

III.5.5.4 Contraintes fonctionnelles et sémantiques

Notre modèle ne tient pas compte des contraintes fonctionnelles ou sémantiques. Cela est acceptable pour des cas où seule la topologie compte. Mais il ne sera cependant pas valable dans un véritable outil de schématique où le domaine est lié fortement à la géométrie. Dans des domaines tels que la mécanique et l'architecture, la fonction d'un composant est liée à sa géométrie et à ses relations géométrique avec les autres. Par exemple, le fait qu'un mur soit celui d'un pignon est une contrainte sémantique très forte qui implique un positionnement très précis par rapport à deux autres composants (les deux murs de façade).

Par contre, dans le domaine de l'électronique et plus précisément des circuits imprimés, on n'a pas vraiment de telles contraintes géométriques ayant directement rapport avec la fonction du composant. Il existe cependant des contraintes fonctionnelles fortes qui impliquent des relations géométriques entre deux ou plusieurs composants. Il vaudrait mieux, pour intégrer correctement cette interface dans un système de SCAO électronique (ou tout autre domaine dans lequel ce genre de contraintes existe), fournir des mécanismes permettant de spécifier et contrôler des contraintes fonctionnelles et sémantiques induisant des relations géométriques et topologique fortes.

III.5.5.5 Problèmes de cycle

Nous avons vu qu'il pouvait apparaître des cycles dans la résolution. C'est un problème difficile à résoudre et nous n'avons pas ici de solution générale à proposer. On peut élaborer des mécanismes assez simples d'arrêt de résolution en cas de cycle, mais cela n'est pas suffisant, car un cycle n'est pas toujours néfaste à la résolution ou à la qualité de celle-ci. Ce qui est difficile à évaluer à ce niveau est l'implication du cycle sur la résolution future. C'est tout le problème du contrôle global dans un système distribué [Buisine 89] qui est ici en jeu.

Mais redécrivons ici le problème : au cours d'une tentative de résolution, on parcourt une branche de l'arbre de recherche de solution qui nous amène à retravailler sur un composant qui a déjà été modifié auparavant, et l'action décidée concernant ce composant renforce l'hypothèse faite au cours de la première modification de position de ce composant. Deux éventualités sont alors possibles :

- au bout d'un temps fini, et vu les modifications apportées à l'état de la base, on trouve une solution : tous les composants concernés directement par la résolution (composants en conflit) ont pu être déplacés ou déportés de manière à supprimer tous les conflits. C'est le cas inespéré où la résolution se termine bien.

- la prise de décision et la résolution amènent à une boucle infinie de résolution par jeu d'équilibre des contraintes géométriques : on fera perpétuellement des déplacements qui n'apportent aucune solution au problème. Le système "boucle" indéfiniment.

Le problème à résoudre pour les cycles est donc le suivant : comment reconnaître cette situation et surtout faire la différence entre les deux cas précédents. La reconnaissance d'un renforcement d'hypothèse ou d'un "script d'actions" sur un ensemble de composants est peut-être réalisable, mais comment faire le distinguo entre les situations amenant à une solution et celles donnant un bouclage infini ? C'est là tout le problème des cycles et nous pensons qu'une étude spécifique serait nécessaire pour apporter une réponse à ce problème.

III.5.5.6 Interface homme-machine

Nous avons vu que pour qu'un système CAO ou interactif soit agréable, confortable d'emploi, il faut que certaines opérations interactives, en particulier celles ayant un caractère sémantique faible et un impact visuel fort, soient très rapides.

Hors tout porte à croire qu'un système distribué de résolution ne fera rien en ce sens. Les opérations à réaliser sont à la fois nombreuses et coûteuses en place mémoire (sauvegarde distribuée des contextes) et en temps (délégation et interrogation entre voisins et composants pour un conflit donné). Il faudra donc prévoir des limites aux mécanismes décrits ici, pour disposer d'une rapidité d'exécution acceptable dans des cas critiques. Nous pouvons proposer par un exemple un mécanisme primaire d'arrêt de la résolution à une profondeur peu élevée dans l'arbre de recherche.

D'autre part nous ne devons pas oublier que le but de cette interface est d'aider l'utilisateur dans sa conception. Or un spécialiste aura surtout besoin d'aide dans les cas complexes : beaucoup de composants au même endroit, travail de conception dans un endroit de la carte impliquant des contraintes géométriques très fortes, ...

Une solution à ce deuxième type de problème, vu la remarque précédente, est de permettre à l'utilisateur d'inhiber et d'activer ce système résolution distribuée à volonté. Ainsi, dans les cas simples, l'utilisateur peut opérer avec le système d'une manière manuelle classique, et dans les cas où il aimerait une aide, il activera le système que nous avons décrit.

Enfin, il est intéressant, à la fois pour diminuer l'ennui dû à l'attente de l'exécution de la résolution, et pour donner un aperçu de la manière de procéder choisie par le système, de montrer les états successifs de la base au cours de la résolution. A chaque nouvelle hypothèse ou action tentée par le système, on dessine la modification apportée à la base, ce qui permet une visualisation directe de la résolution. En outre, ce procédé, assorti d'une interruption possible de l'utilisateur, permet à celui-ci de reconnaître des cas de cycles, et de les stopper manuellement, ou encore de détecter que le système commence à choisir une solution inadéquate.

III.5.5.7 Situation par rapport à quelques systèmes de placement existants

Parmi les systèmes de placement existant intégrant des techniques d'intelligence artificielle, nous avons sélectionné ici quatre systèmes basés sur les techniques de satisfaction de contraintes : PIAF [Jabri & Skellern 89], ATLAS [Du Verdier 91], CADOO [Andre 86], OSC-ART [Clerc 87, Orchampt 87]. Sans revenir sur les détails de conception des ces systèmes, nous rappeler leurs caractéristiques.

CADOO est une maquette de système expert dédié à l'aménagement de compartiments propulsifs de navire. L'idée de CADOO est de partir d'une base de connaissance du problème pour générer un ensemble de contraintes, de planifier ensuite à partir de ces contraintes un ordre de placement des composants et de les placer suivant le paradigme du placement constructif, et finalement de tenter de remédier au cours du placement à un échec.

OSC-ART est dédié au placement de circuits imprimés sur une carte. C'est un système à base de régles écrit en PROLOG qui intègre un modèle de composant rectangulaire simple, associé à un système de description de contraintes géométriques utilisant le prédicat *en-regard* qui peut être interprété de différentes manières suivant les paramètres du prédicat, pour donner par exemple une contrainte comme l'adjacence ou la proximité entre composants.

PIAF concerne la conception de zone rectangulaire définissant des espaces fonctionnels pour des circuits intégrés VLSI. Il est basé sur une approche de conception descendante, intègre un système de satisfaction de contraintes et des critères d'évaluation des solutions trouvées.

ATLAS concerne le problème de l'aménagement spatial d'objets de forme géométrique quelconque modélisée par des quadtrees [Foley & Van Dam 84, Pavlidis 82]. Il intègre des contraintes de forme (recto ou verso d'un composant), d'orientation entre composants, et de positionnement (pas de recouvrement, distance minimale, maximale, alignement, adjacence, ...). L'algorithme de résolution de problème est un classsique du domaine des systèmes à satisfaction de contraintes; il est capable cependant de raffinement (précision de la qualité de solution) des solutions trouvées.

Enfin, concernant l'aspect semi-automatique et aide au concepteur pour SCAO, PEARL [Dejesus & Callan 85] fournit une aide de choix et de placement de composants pour une méthode de placement du genre constructif. Il concerne la conception d'équipement d'alimentation et est basé sur des techniques classiques d'IA (Base de connaissances décrite avec OPS5).

Nous avons détecté les différentes suivantes entre ces systèmes et MAPS :

1. Ces logiciels fonctionnent tous sur un modèle centralisé de raisonnement, adapté à la recherche d'une solution globale du problème. Au contraire MAPS travaille de manière décentralisé et fournit les bases pour résoudre des problèmes locaux et donc incomplets, caractère typique du problème général de conception.

2. Ils tentent de résoudre de le problème complet. Il faut donc que celui-ci soit complétement défini, ce qui est rarement le cas lors de prototypage ou de conception préliminaire. MAPS est dédié à la résolution de conflits locaux. L'efficacité du système en est accrue.

3. Ils simulent un placement du genre constructif qui consiste d'abord à choisir un composant à placer et ensuite à le placer. MAPS, en laissant d'une part l'utilisateur choisir le composant à placer et en s'attachant uniquement à résoudre les problèmes locaux, permet au concepteur d'utiliser les deux approches naturelles de conception : placement constructif et itératif. MAPS ne sait quand à lui réaliser que du placement itératif.

4. Ils sont principalement basés sur des techniques de satisfaction de contraintes décrites explicitement. MAPS propose à la base l'idée d'un voisinage qui socialise les agents et permet de traiter les contraintes de manière dynamique. Il propsoe un modèle où il n'y a pas de contraintes à proprement parler. Les contraintes peuvent être exprimées par un voisinage particulier, exprimant un certain type de relation à respecter entre les composants. On peut par exemple décrire un voisinage spécialisé pour l'adjacence ou des contraintes fonctionnelles. Cependant de tels travaux n'ont pas encore été réalisés et devrait l'être pour améliorer le système.

5. Ils intègrent par leurs représentation des connaissances un certain type de raisonnement. Le modèle multi-agents *microscopiques* de MAPS permet de tirer de la grande modularité locale des composants et fournit un système de conception simple [Ferber 89].

L'un des principaux défauts de MAPS tel que nous l'avons présenté est d'être incapable de fournir de manière certaine une solution satisfaisante au problème global rencontré par les concepteurs, et de poser des problèmes de détection de cycle, ou de solution globale raisonnée [Buisine 89]. Comme L.Buisine, nous pensons que ce problème d'évaluation de la solution par les agents est crucial lorsque l'on ne sait pas si le système tend naturellement à évoluer vers une situation globalement statisfaisante. Nous proposons dans le paragraphe III.6 une ébauche primitive de solution pour tenter d'évaluer les solutions fournies par un système de type MAPS.

III.5.5.8 Conclusion

Nous concluerons simplement en rappelant les résultat obtenus dans cette partie concernant la résolution.

Nous avons montré qu'un modèle distribué pouvait être implémenté pour résoudre les problèmes d'aménagement spatial. Le modèle multi-agent MAPS présenté permet de résoudre de manière totalement distribuée les conflits géométriques. Malheureusement la résolution n'a pas encore été implantée en machine.

Nous avons également montré l'importance des problèmes de cycles lors d'une résolution de conflit. Le problème sous-jacent important est l'évaluation de la solution courante au niveau

global du problème par un ensemble d'agents microscopiques indépendants focntionnant uniquement de manière locale.

Des deux stratégies présentées pour résoudre les conflits, la première est la moins fine et celle provoquant sans doute le plus de conflits par propagation, mais dans certains cas peu complexes celle qui fournira la plus rapidement une solution. La deuxième stratégie fait usage du contexte local des voisins d'un composant en conflit, pour que les agents coopérent réellement et fournissent une solution de meilleure qualité, par une évaluation seulement locale des choix.

Nous avons également étudié la parallélisation du premier algorithme de résolution, Ceci montrant qu'une implémentation réellement "acteur" du système (communication par envoi de messages asynchrones) est nécessaire pour pouvoir vraiment profiter des caractères distribués du modèle. il faudra dans ce cas implémenter des mécanismes acteurs à Smalltalk, sachant que certains travaux ont déjà été réalisés en ce domaine [Briot 89].

Nous avons également vu que les connaissances heuristiques ou les méta-connaissances pouvaient être relativement indépendante du modèle de voisinage et que les algorithmes de résolution proposés fonctionnent également d'une manière indépendante de la relation de voisinage choisie.

Enfin le modèle MAPS est adapté aux problèmes de reconception que le concepteur rencontre fréquemment de par la nature imprécise et itérative de l'activité de conception.

III.5.6 Proposition élémentaire d'architecture de SCAO intégrant MAPS

Nous avons vu qu'un défaut essentiel d'un système distribué tel que MAPS est la difficulté d'évaluation au niveau global d'un choix effectué par un composant. Un autre problème important concerne le traitement de cycles dans la résolution; comme on ne sait pas évaluer la qualité globale de la solution en court de construction, on ne sait pas discriminer les cycles à stopper (en arrêtant la résolution ou en provoquant un retour-arrière) de ceux à conserver. Nous proposons ici une solution à ce problème, qu'il faudrait cependant tester sur des cas réels.

Le modèle multi-agent de base adapté au problème de l'agencement spatial place les composants à un niveau microscopique où tous sont "frères" en quelque sorte. Aucun d'entre eux n'a la charge de collecter l'ensemble des solutions choisies par les composants et de les analyser.

L'idée que nous proposons tire parti de cette dernière remarque : on tente de décomposer en une hiérarchie les fonctions du produit et les éléments qui le composent.

Ainsi sur une carte de circuit imprimé ou sur un circuit VLSI, on peut découper le fonctionnement du produit en plusieurs sous-fonctions : alimentation, mémoire, plan de masse, ... Chaque fonction nécessite l'utilisation de certains composants très précis. Au niveau des

composants internes au sous-fonctions, un modèle du genre MAPS permet de manière satisfaisante pour la fonction en question, et de plus chaque sous-fonction peut être assimilée à un composant en tant que tel d'un autre système multi-agent. La différence est que l'agent modélisant la fonction peut effectuer des raisonnements sur les sous-agents la constituant.

En réitérant le modèle sur les fonctions plus générales ou plus spécialisées, on obtient un arbre dynamique d'agents contrôlant un sous-ensemble d'agents organisés suivant un modèle de type MAPS. Ce contrôle permet une évaluation des solutions proposées par les sous-agents, mais nécessite que les sous-agents communiquent régulièrement avec leur "père". De ce fait, on peut implanter normalement plus facilement un système d'évaluation de solution et de discrimination de certains cycles évalués comme néfastes dans la résolution. De plus, en cas d'échec de placement, on peut reporter le problème au niveau d'abstraction plus élevé de l'agent père, celui communiquant avec ses agents "frères" et essayant une autre solution à un niveau plus global.

Si ce type de modèle peut paraître parfois peu adapté au cas où les fonctions du produit n'ont pas un rapport direct avec sa géométrie, dans d'autre cas de CAO comme la mécanique ou l'architecture où la relation entre les composants à un rapport direct à la fois avec les caractères géométriques les liant, et où le rôle qu'ils jouent est directement lié à l'aspect géométrique du produit, le modèle hiérarchique multi-agent présenté s'adapte très bien au problème de l'agencement spatial. On disposera alors d'un modèle multi-agent adapté à une conception descendante du produit final, tenant compte également des aspects imprécis et incertains de la conception [Bijl 90, Moulin 90, Neveu & al 90, Trousse 89].

Nous pensons donc que le problème de l'aide à la conception dans le domaine de l'aménagement spatial peut être correctement modélisé par un système hiérarchique multi-agents dont chaquq niveau de hiérarchie suit les concepts décentralisés et géométriques de MAPS. C'est une direction de recherche future que nous avons l'intention d'explorer.

Conclusion

Les systèmes de CAO basés sur des techniques d'intelligence artificielle ont maintenant atteint un bon niveau de maturité pour spécifier et décrire les produits à concevoir. Leurs utilisateurs demandent maintenant que ces outils les aident dans leurs tâches de conception. Ceci implique l'intégration de capacités de raisonnement et de systèmes d'aide à la conception. Nous pensons cependant comme B.Trousse [Trousse 89] que l'évolution de ces systèmes ne passe pas par l'automatisation complète des tâches de conception, ni par le remplacement des concepteurs par des systèmes automatiques basés sur des techniques d'intelligence artificielle, mais plus par des systèmes semi-automatiques assistant l'expert dans certaines tâches spécifiques :

"... Les systèmes experts ne vont pas remplacer l'ingénieur ou les outils classiques de CAO, mais plutôt constituer des interfaces intelligentes qui, en manipulant des données de manière symbolique, en effectuant des raisonnements simples et donc en libérant l'ingénieur des tâches répétitives, rapprocheront l'utilisateur des programmes classiques de CAO".

Le travail décrit dans cette thèse concerne l'intégration de techniques d'IA distribuée pour réaliser un système d'aide à la conception dédié à l'aménagement spatial, ainsi que la communication bidirectionnelle entre programmes d'IA et programmes procéduraux.

Notre apport à l'intégration des techniques d'IA en CAO consiste en :

- la formalisation de modèles d'interface entre langages d'IA et langages procéduraux, ainsi qu'une étude de l'existant en matière de communication entre ces langages,
- la réalisation d'un modèle d'architecture distribuée pour l'aménagement spatial : le modèle MAPS,
- la mise au point d'un voisinage particulier adapté à la modélisation MAPS, ainsi que la mise en place d'un système de gestion distribuée de ce voisinage,
- la conception de "moteurs" de résolution de conflit géométrique entre composants.

Ce choix de modèle décentralisé permet de tirer parti de la modularité naturelle des composants dans le cas de l'agencement spatial. La décentralisation permet de construire facilement un système où chaque composant a un comportement propre suivant son état et ses caractéristiques techniques.

De plus ce modèle respecte la démarche naturelle de l'utilisateur de logiciel de schématique ayant à réaliser les tâches routinières de correction de configuration spatiale, car il travaille en raisonnant sur l'état des composants pris en tant qu'entités indépendantes, mais reliées au monde environnant par un ensemble de contraintes géométriques ou fonctionnelles.

Contrairement aux systèmes de satisfaction de contraintes centralisés, ou l'on doit fabriquer un plan d'action pour résoudre le problème de la disposition correcte des composants, c'est l'équilibre (ou la satisfaction) des contraintes portant sur le composant qui détermine naturellement l'action à réaliser, et qui contribue pour partie à la solution globale. Celle-ci est atteinte lorsque tous les composants mis en jeu dans la résolution ont satisfait leurs contraintes.

La décentralisation permet d'autre part de fabriquer un système doté d'une interface utilisateur où le concepteur est en relation directe avec les composants lors de l'interaction, qui conserve les aspects classiques d'une interface usager en CAO, et qui permet d'envisager facilement un fonctionnement semi-automatique de l'assistance apportée par le système.

Le modèle que nous avons décrit n'est pas contradictoire avec une représentation des connaissances différente portant sur des aspects plus fonctionnels des composants; il est plutôt complémentaire de représentations des connaissances à base de frames par exemple, qui supportent les notions de vues multiples d'un objet ou d'incomplétude de spécification des composants. Nous avons d'autre part vu que ce modèle est adapté à une implantation sur une architecture de machine distribuée ou parallèle, et l'on voit bien alors l'intérêt de l'attachement d'un composant à un processeur, dans le but d'améliorer notoirement les performances. Le modèle décentralisé décrit ici s'avère de plus fortement compatible avec une interface homme-machine à manipulation directe pour résoudre les problèmes d'agencement spatial.

Nous avons montré d'autre part l'importance de la socialisation des agents dans une telle architecture, socialisation qui s'effectue grâce à la notion capitale de voisinage, qui doit posséder des qualités essentielles si l'on veut assurer à la fois une bonne communication et une bonne coopération entre agents géométriques [Buisine 89, Ferber & Ghallab 88, boissier 90]. Nous avons trouvé un type de voisinage possédant les qualités minimales requises, mais nous considérons cependant qu'il n'est pas assez puissant pour modéliser certaines modifications géométriques sur les composants (comme des déplacements obliques de composant par exemple).

Les algorithmes de gestion de ce voisinage se révèlent de plus indépendants du voisinage (au prix de très légères modifications) si la propriété de symétrie est respectée. Les deux moteurs de résolution que nous avons présentés ne prennent pas en compte actuellement des heuristiques du domaine, explorent l'ensemble des solutions en profondeur d'abord, supportent le retour-arrière dans la recherche de solutions, et fonctionnent sur des communications directes entre agents. Le contrôle implanté dans ces algorithmes se révèle cependant incapable de discerner les occurences de cycle de résolution favorables à l'obtention d'une solution, des cycles néfastes à l'obtention de celle-ci. Ceci est principalement dû au caractère totalement décentralisé du contrôle :

"Un système de résolution reposant sur le modèle d'acteurs s'appuie sur un modèle de communication directe (la transmission de message avec continuation) et un modèle de contrôle parfaitement distribué. Dans un tel cadre, la dichotomie micro/macro précédemment évoquée est très affirmée de telle sorte que le conflit latent connaissances et actions locales / cohérence globale précédemment évoqué s'y pose avec une acuité toute particulière." [Buisine 89].

Si nos travaux ont mis en lumière les principes de base d'un système d'aide à l'aménagement spatial reposant sur une architecture distribuée, il est cependant nécessaire de l'améliorer sur certains points.

Il semble en premier lieu très important d'améliorer le modèle de voisinage pour que la propriété d'isotropie de la relation soit satisfaite. Ceci est indispensable pour pouvoir prendre en compte un ensemble de modifications plus riche que de simples déplacements verticaux ou horizontaux, ou pour être capable de gèrer des changements de forme géométrique des composants, ou encore pour pouvoir manipuler des formes géométriques complexes. Nous pensons que cette phase d'enrichissement du modèle de voisinage est nécessaire avant d'évaluer les qualités du modèle distribué sur un cas réel. De plus, une relation de voisinage induisant une topologie complète permettra sans doute d'élaborer des mécanismes de gestion de voisinage plus sûrs (car vérifiant des propriétés induites par la topologie) et plus efficaces (nous avons pu le remarquer sur chacun de nos quatre sous-voisinages qui induisent chacun un topologie sur un cylindre du plan grâce à la distance d_{DS}).

Notre architecture complétement décentralisée souffre d'un manque de contrôle global de la qualité de la solution. Il nous semble important de trouver une architecture adaptée à ce problème. Nous pensons que les problèmes de sélection de cycles favorables ou néfastes à l'obtention d'une solution pourraient être en partie résolus grâce à un enrichissement de notre modèle décentralisé tel que nous l'avons ébauché dans le paragraphe III.6. Nous pensons d'autre part que des comportements heuristiques lors de la résolution permettraient d'améliorer grandement les qualités de la solution obtenue, par le choix judicieux (dépendant également de contraintes fonctionnelles) d'une action à effectuer par un composant lors de la résolution.

Il nous semble également nécessaire d'envisager une intégration du modèle à d'autres environnements de CAO intégrant des capacités de raisonnement, tel que le système *ANAXAGORE* [Trousse 89]. L'évaluation du comportement du modèle en vrai grandeur sur un système de CAO complexe nous parait également important.

D'autre part, l'aspect décentralisé du modèle ne prendra toute sa dimension qu'avec une tentative d'implantation du modèle dans un contexte d'architecture de machine parallèle.

Remarquons enfin que nous avons conçu un seul type de voisinage. Celui-ci exprime des capacités de comportement géométrique socialisé, mais n'exprime cependant qu'un seul type de comportement. Ayant remarqué d'autre part que les algorithmes de gestion de voisinage étaient relativement indépendants de celui-ci, on peut penser que d'autres relations de voisinage puissent exprimer d'autres comportements sociaux, portant par exemple sur des aspects fonctionnels des composants. Ceci permettrait de modéliser plusieurs types de comportement social.

Le modèle multi-agents issu des techniques d'IA distribuée possède de nombreuses qualités pour le problème de l'aménagement spatial tel qu'il existe quand on utilise un système de CAO classique. Nous avons ici principalement édifié les premiers éléments d'un système de conception doté d'une aide à l'aménagement spatial de composants. Il reste maintenant à

l'enrichir pour le tester dans des cas de conception réels, et à voir si le paradigme décentralisé s'appuyant sur des agents élémentaires pourrait être également développé dans d'autres domaine de l'activité de conception. Il semble d'une manière général très intéressant car il s'appuie sur la modularité naturelle des composants qui constituent le produit final de la conception.

BIBLIOGRAPHIE

[Abelson & al 85] H. Abelson, G.J. Sussman,, J. Sussman.
Structure and Interpretation of Computer Progams.
The M.I.T. Electrical Engineering and Computer Science series.
Mac Graw-Hill, 1985.

[Abiteboul & Grumbach 87] S. Abiteboul, S. Grumbach.
Bases de données et objets structurés.
Technique et Sciences Informatiques, Vol 6(No 5):383–404, 87.

[Agha 86a] G. Agha.
A model of concurrent computation in distributed systems.
MIT Press, Cambridge, U.S.A., March 86.

[Agha 86b] G. Agha.
An overview of actor languages.
SIGPLAN Notices, 21(10):58–67, October 86.

[ALSYS 87] ALSYS.
ALSYS, PC-AT Ada Compiler. Application Developer's Guide, Version 3.2.
société ALSYS, France, 87.

[Andre & al 90] J.M. Andre, J.M. Chazot, Ph. Collange, D. Denier,, J.L. Ravalet.
Intégration d'un système expert dans un système de conception de plans d'armature.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 177–188. Hermes, 90.

[Andre & Chazot 89] J.M. Andre, J.M. Chazot.
FEREX : un système expert pour la conception des plans d'armatures.
TSI, 8(2):102–113, 89.

[Andre 86] J.M. Andre.
Vers un système intelligent pour l'aménagement spatial : CADOO.
In *Actes du 4ème colloque international d'IA*, pages 33–49, MArseille, 86. CIIAM.

[Arnold & Bono 88] D. B. Arnold, P.R. Bono.
CGM and CGI, Metafile and Interface Standards for Computer Graphics.
Springer Verlag, 88.

[Barbeyre & al 83] G. Barbeyre, T. Joubert,, M. Martin.
Manuel d'utilisation de PROLOG CNET, Version Pascal Multics 2.00.
Technical report, CNET, 83.

[Barnes 88] J. Barnes.
Programmer en ADA.
Inter Editions, Paris, 88.
Texte français de V. Amiot, ALSYS.

[Beaudoin Laffon & al 90] M. Beaudoin Laffon, B. Chabrier,, M. Thiellement.
Graphical layers in AVIS.
ESF internal report, 90.

[Beech & al 86] D. Beech, Members of IFIP Working Group 2.7, et al.
Concepts in User Interfaces: A Reference Model for Command and Response Languages.
Lecture Notes in Computer Science. Springer Verlag, 86.

[Begg 84] V. Begg.
Eléments d'introduction des systèmes experts en CAO.
Hermes, Paris, 84.

[Berthelot 85] A. Berthelot.
Les systèmes de CAO en Architecture dans leur environnement professionnel et informatique.
In *Etudes en CFAO: Architecture et Batiment.* Hermes, Paris, 85.
J.C. Lebahar Rédacteur.

[Bezivin 88] J. Bezivin.
Langages à objets et programmation concurrente; quelques expérimentations avec Smalltalk-80.
BIGRE + GLOBULE no 59, pages 176–187, Avril 88.

[Bijl 90] A. Bijl.
Formality in design : logic and what else ?
In *Actes de PRICAI 90*, pages 17–24. Japanese society for Artificial Intelligence, 90.

[Billet & Lhelgoum 87] A. Billet, K. Lhelgoum.
Un modèle de représentation des connaissances pour la conception assistée par ordinateur.
In *Actes de MICAD 87*, pages 433–448. Hermes, 87.

[Boissier 90] O. Boissier.
La coopération entre systèmes à base de connaissances.
Technical Report RR 96, Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle, IMAG Grenoble, Février 90.

[Borning 86] A. Borning.
Defining constraints graphically.
In *Human Factors in Computing Systems. SIGCHI'86 proceedings, Boston MA*, pages 13–17, April 86.

[Borning 87] A. Borning.
Graphical definition in THINGLAB.

HCI Journal, pages 270–295, March 87.

[Bourne & al 90] H. Bourne, A. Clement, A. Foussier, T. Bourlais,, M. Sicard.
Jade : un jeu d'outils d'aide à la décision technologique.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 148–160. Hermes, 90.

[Bourne 85] S. Bourne.
Le système UNIX.
Inter Editions, Paris, 85.
Traduit de l'américain par M. Dupuy, E.N.S.T. Paris.

[Boy 88] G.A. Boy.
Human-machine distributed intelligence : the operator assistant concept.
In *Actes des 8èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 155–168, Avignon, 88.

[Brinkley & al 87] J.F. Brinkley, B.G. Buchanan, R.B. Altman, B.S. Duncan,, C.W. Cornelius.
A heuristic refinement method for spatial constraints satisfaction problems.
Technical Report 87-05, Knowledge Systems Laboratory, Standford University, California, January 87.

[Briot 88] J.P. Briot.
From Objects to Actors : Study of a Limited Symbiosis in Smalltalk-80.
Technical Report research report LITP 88-58 RXF, Université de Paris 6, France, Septembre 88.

[Briot 89] J.P. Briot.
Actalk : a testbed for classifying and designing actor languages in the Smalltalk environment.
In *Actes de ECOOP 89*, pages 109–129. Cambridge University Press, 89.

[Buisine 89] L. Buisine.
Spécification d'une architecture multi-experts pour la simulation financière d'entreprise.
Thèse de Doctorat, Université de Lille 1, Décembre 89.

[Bureau & Bertrand 87] D. Bureau, G. Bertrand.
MAct : un outil d'intégration de différentes représentations de connaissances.
In *Actes de MICAD 87*, pages 479–489. Hermes, 87.

[CAE 4.0 88] CAE 4.0.
Metadesign CAE 4.0, Manuel de Référence.
METADESIGN SA, Lille, 88.

[Carre & Clere 88]
F. Carre, C. Clere.
Object-oriented languages and Actors: which language for a distributed approach.
In *OOPSLA 88 Conference Proceedings*, pages 73–75, 88.

[Carre & Comyn 87a]
B. Carre, G. Comyn.
Contraintes liées à une représentation par objets.
Technical Report IT 95, Laboratoire d'Informatique Fondamentale de Lille 1, Villeneuve d'Ascq, 87.

[Carre & Comyn 87b]
B. Carre, G. Comyn.
Instantiation et Représentation.
Technical Report IT 94, Laboratoire d'Informatique Fondamentale de Lille 1, Villeneuve d'Ascq, 87.

[Carre 85a]
B. Carre.
Etat de l'art en CAO de placement-routage automatique, première évaluation d'une approche système expert.
Projet d'étude EUDIL, Université de Lille 1, 85.

[Carre 85b]
B. Carre.
Système Semi-automatique de placement-routage pour circuits imprimés.
Technical report, Rapport de Stage Eudil, Université de Lille 1, Juin 85.

[Carre 89]
B. Carre.
Méthodologie orientée objet pour la représentation des connaissances. Concepts de point de vue, de représentation multiple et évolutive d'objet.
Thèse de Doctorat en Informatique, Université de Lille 1, Janvier 89.

[CGI 88]
CGI.
Computer Graphics Interface, Draft Proposal, Parts 1-8.
Technical Report DP 9636, ISO/IEC JTC1/SC24 N21, Information Processing System, Computer Graphics, 88.

[Chaillou 86]
V. Chaillou.
De la CAO à l'automatisation du processus industriel : les conséquences sur les structures système.
In *Actes de MICAD 86*. Hermes, 86.

[Chailloux 85a]
J. Chailloux.
La Machine LLM3.
Technical Report 55, INRIA Rocquencourt, 85.

[Chailloux 85b]
J. Chailloux.
LE-LISP de l'INRIA, Le Manuel de Référence, Version 15.2.
INRIA, Rocquencourt, 85.

[Cholvy & Foisseau 83a] L. Cholvy, J. Foisseau.
A CAD object-oriented and rule-based system.
In *IFIP congress*, Paris, 83. North-Holland Publishing Company.

[Cholvy & Foisseau 83b] L. Cholvy, J. Foisseau.
Etat de l'art en CAO: Problèmes des bases de données.
Le nouvel Automatisme, pages 49–54, Septembre 83.

[Cholvy & Foisseau 85] L. Cholvy, J. Foisseau.
Bases de Données CAO : modélisation d'objets complexes et des liaisons entre objets.
In Y. Gardan editor, *Etudes en CFAO Mécanique*. Hermes, Paris, 85.

[Chouraqui & Dugerdil 86] E. Chouraqui, P. Dugerdil.
Applications des langages orientés objets à la CAO en architecture.
In *Journées Langages Orientés Objets*, pages 226–232. AFCET Informatique, BIGRE + GLOBULE N 48, Janvier 86.

[Claude 86] D. Claude.
Panorama des travaux français et internationaux en matière d'échange de données CAO.
In *Actes de MICAD 86*, pages 669–680. Hermes, 86.

[Clerc & al 87] T. Clerc, M. Sadgal,, P. Orchampt.
Prototype d'un système de placement intelligent dans le plan.
In *Actes de MICAD 87*, pages 489–503. Hermes, 87.

[Colmerauer 90] A. Colmerauer.
An Introduction to PROLOG III.
Communications of the ACM, 33(7), July 90.

[Condillac 86] M. Condillac.
Prolog, Fondements et Applications.
AFCET et Dunod Informatique, Bordas, 86.

[Coutaz 87] J. Coutaz.
The Construction of User Interface and the Object Paradigm.
In *Actes de ECOOP 87*, pages 135–153, Juin 87.
BIGRE + GLOBULE N 54.

[Coutaz 89] J. Coutaz.
Architecture models for interactive software.
In *Actes de ECOOP 89*, pages 384–399. Cambridge Press, 89.

[Coutaz 90] J. Coutaz.
Interfaces Homme-Ordinateur. Conception et réalisation.
Dunod Informatique, 90.

[Coutaz 91] J. Coutaz.
Interfaces Homme-Machine : un regard critique.
TSI, 10(1):53–64, Janvier 91.

[Cox 86] B. J. Cox.
Object Oriented Programming, An Evolutionnary Approach.
Addison Wesley, 86.

[Coyne & Postmus 90] R.D. Coyne, A.G. Postmus.
Spatial application of neural networks in computer aided design.
Artificial Intelligence in engineering, 5(1):9–21, 90.

[D-PROLOG 85a] D-PROLOG.
D-PROLOG : Manuel d'utilisation du langage.
Société DELPHIA, Grenoble, 85.

[D-PROLOG 85b] D-PROLOG.
D-PROLOG, Notice de la version 3.0 sur MS-DOS.
Société DELPHIA, Grenoble, Septembre 85.

[Dalmasso 88] A. Dalmasso.
Metadesign GKS 4.0, Interface langage Microsoft C 4.0, pour MS-DOS.
Société METADESIGN, Lille, 88.

[David 81] B. David.
Méthodologie pour la construction de systèmes CAO : SIGMA CAO.
Thèse de Docteur es Sciences, INPG et UMSG, Grenoble, 81.

[Dejesus & Callan 85] E.J. Dejesus, J.P. Callan.
PEARL : a knowledge based expert assisted CAD tool.
In *The second conference on Artificial Intelligence Applications (CAIA 85)*, pages 258–263, 85.

[Delahaye 86] J.P. Delahaye.
Cours de Prolog, avec Turbo-Prolog, Eléments de base.
Editions Eyrolles, Paris, 86.

[Delobel & Adiba 84] C. Delobel, M. Adiba.
Bases de données et Systèmes relationnels.
Dunod, 84.

[Demazeau & Muller 89] Y. Demazeau, J.P. Muller, editors.
Decentralized AI, proceedings of the First European Workshop on modelling Autonomous Agents in a Multi-Agents World, Décembre 89.

[Demazeau & Muller 90] Y. Demazeau, J.P. Muller.
Decentalized A.I.
In *Decentralized A.I.*, pages 3–13, Amsterdam, July 90. North Holland, Elsevier.

[Descotte 81] Y. Descotte.
Reprépresentation et exploitation de connaissances 'expertes' en génération de plan d'actions.

Thèse de 3ème cycle, Institut National Polytechnique de Grenoble, Décembre 81.

[Devin 85] M. Devin.
Le Portage du Système LE-LISP, Mode d'emploi.
Technical Report 50, INRIA Rocquencourt, 85.

[Donz & Hurtado 84] P.H. Donz, R. Hurtado.
Le langage D-PROLOG, initiation au langage de la 5ème génération.
Editests, 84.

[Druais 87] S. Druais.
ICARE: un système d'aide à la conception d'ateliers flexibles.
In *Actes des 7èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 1559–1579, Avignon, 87.

[Du Verdier & Tsang 91] F. Du Verdier, J.P. Tsang.
Un raisonnment spatial par propagation de contraintes.
In *soumis aux 11èmes Journées Internationales sur les Systèmes experts et leurs applications*, Avignon, 91.

[Ducrot 86] A. Ducrot.
Présentation de PHIGS.
Technical Report CG97/CN21/GE2 F63, AFNOR, Décembre 86.

[Dugerdil 85] P. Dugerdil.
Une méthodologie orientée objet pour la représentation des connaissances en CAO d'architecture.
Mémoire de DEA, Université d'Aix-Marseille II. G.I.A. LUMINY - GRTC, Juin 85.

[Duncan 86] R. Duncan.
Advanced MS-DOS.
Microsoft Press, 86.

[Ege & al 87] R.K. Ege, D. Maier,, A. Borning.
The Filter Browser. Defining interface graphically.
In *ECOOP 87 proceedings*, pages 155–165. BIGRE + GLOBULE No 54, Juin 87.

[Eggen & al 90] J. Eggen, A. M. Lundteigen,, M. Mehus.
Integration of Knowledge from Different Knowledge Acquisition Tools.
In *Current Trends in Knowledge Acquisition*, Amsterdam, June 1990.

[El Dashman & Barthes 90] K. El Dashman, J.P. Barthes.
Un système intelligent de CAO.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 89–102. Hermes, 90.

[ENJEUX 84] ENJEUX.
L'assistance des normes.
ENJEUX : le nouveau manuel de la normalisation française, Décembre 84.

[Epstein & Lalonde 88] D. Epstein, W.R. Lalonde.
A smalltalk window system based on constraints.
In *OOPSLA 88 proceedings*, pages 83–94, 25-30 September 88.

[Erman & al 80] L.D. Erman, F. Hayes-roth, V.R. Lesser,, D.R. Reddy.
The Hearsay-II speech understanding system : integrating knowledge to resolve uncertainty.
Computing Surveys, 12(2), June 80.

[Erman & al 81] L.D. Erman, P.E. London,, S.F. Fickas.
The Design and an Example Use of Hearsay III.
In *IJCAI 7 proceedings*, 81.

[ESF 89] ESF.
ESF Technical Reference Guide.
Technical report, TAT, 89.

[EUROPIA90] EUROPIA.
2èmes Journées Européennes sur les applications de l'IA en Architecture, Bâtiment et Génie Civil, 90.

[Farreny 81] H. Farreny.
ARGOS-II : un système de production pour engendrer et exécuter des plans.
In *1ères journées AFCET-ADI Systèmes-experts*, Avignon, 81.

[Fauvet & Rieu 87] M.C. Fauvet, D. Rieu.
CADB: un système de gestion de base de données et de connaissances pour la CAO.
In *Actes de MICAD 87*, pages 415–431. Hermes, 87.

[Favard 89] R. Favard.
Proposition d'Insertion de Techniques d'Intelligence Artificielle dans un Système de Conception Assisté par Ordinateur.
Thèse de Doctorat Informatique, Université Claude Bernard de Lyon 1, Juillet 89.

[Ferber & Ghallab 88] J. Ferber, M. Ghallab.
Problématiques des univers multi-agents.
In *Actes des Journées Nationales d'Intelligence Artificielle*, pages 295–320, Toulouse, 14-15 Mars 88. PRC + GRECO IA, Teknea.

[Ferber 89] J. Ferber.
Objets et agents: une étude des structures de représentations et de communications en Intelligence Artificielle.

Thèse de Doctorat d'Etat, Université Pierre et Marie Curie, Paris 6, Juin 89.

[Flemming & Coyne 90] U. Flemming, R.F. Coyne.
A design system with multiple abstraction capabilities.
In *Actes des 10èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 107–121, Avignon, 90.

[Foley & Van Dam 84] J. Foley, A. Van Dam.
Fundamentals of Interactive Computer Graphics.
System Programming Series. Addison-Wesley, 84.

[Fraichard & Demazeau 90] T. Fraichard, Y. Demazeau.
Motion Planning in a multi agent world.
In Y. Demazeau, J.P. Muller, editors, *Decentralized A.I.*, pages 137–153, Amsterdam, July 90. North Holland, Elsevier.

[Gardan & Lucas 83] Y. Gardan, M. Lucas.
Techniques Graphiques Interactives en CAO.
Hermes, Paris, 83.

[Gardan 85] Y. Gardan.
Mathématiques et CAO : méthodes de base, Volume 1.
Hermes, 85.

[Gardan 86a] Y. Gardan.
Etudes en CFAO : CAO mécanique.
HermesS, Paris, 86.

[Gardan 86b] Y. Gardan.
La CFAO: Introduction, Technique et Mise en Oeuvre.
Hermes, Paris, 86.

[Gardan 91] Y. Gardan.
L'intégration dans les Systèmes CFAO : à travers le modèle ou à travers les outils.
In *Actes de la convention IA 91*, pages 455–470. MICADO, Hermes, Janvier 91.

[Gardarin 88] G. Gardarin.
Les Systèmes de Gestion de Bases de Données Orientées Objets.
In *Actes des journées AFCET sur les BD avancées : Bases de données déductives et bases de données orientées objets*, pages 59–79, Décembre 88.

[Genoud & Grabowiecki 86] P. Genoud, J.F. Grabowiecki.
Vers un logiciel graphique interactif de haut niveau: CLOVIS.
In *Actes de MICAD 86*, pages 96–110. Hermes, 86.

[Giambasi & al 83] N. Giambasi, J.C. Rault,, J.C. Sabonnadiere.
Introduction à la Conception assistée par Ordinateur.
Hermes, Paris, 83.

[Giannesini & al 85] F. Giannesini, H. Kanoui, R. Pasero,, M. Van Caneghem.
PROLOG.
Inter Editions, 85.

[GKS 85] GKS.
Graphical Kernel System, description formelle, 2ème édition.
GKS, Norme AFNOR, NF Z-73110, 85.

[GKS-Add 88] GKS-Add.
GKS Functional Description, Addendum 1.
Draft Addendum 7942/DAD1, ISO/IEC JTC1, Décembre 1988.

[GKS3.0 88] GKS3.0.
Metadesign GKS 3.0, Manuel de l'utilisateur.
METADESIGN, Lille, 88.

[GKS4.0 88] GKS4.0.
Introduction à Metadesign GKS 4.0.
METADESIGN, Lille, 88.

[Gleizes & al 91] M.P. Gleizes, P. Glize,, S. Trouilhet.
La communication entre agents hétérogènes, principes et application à SYNERGIC.
In *Actes de la Convention IA 91*, pages 344–360. A.F. MICADO, Hermes, 91.

[Gleizes 90] M.P. Gleizes.
La résolution distribuée de problèmes dans un univers multi-agents.
In *Actes de la Convention IA 90*, pages 121–135. AF MICADO, Hermes, 90.

[Goldberg & Robson 83] A. Goldberg, D. Robson.
SMALLTALK 80, the language and it's implementation.
Addison Wesley, 83.

[Guillot & al 85] J. Guillot, J.F. Rousselot,, J.C. Vignat.
SICAM : une approche modulaire de la conception de mécanismes.
In *Actes de MICAD 85.* Hermes, 85.

[Hanser 90] J.M. Hanser.
CFAO et propagation de contraintes, technologie, enjeux et étapes industrielles.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 123–146. Hermes, 90.

[Haurat & al 90] A. Haurat, F. Barbier,, P. de Vettor.
Evaluation comparative de deux langages à objets sur une application.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 14–32. Hermes, 90.

[Herrmann & Hill 89] M. Herrmann, R.D. Hill.
The structure of Tube - a tool for implementing advanced user interfaces.
In *Proceddings of the EUROGRAPHICS'89 Conference*, Berlin, 89. Springer-Verlag.

[Houbart 90] G. Houbart.
Des objets et des icones pour l'interface utilisateur du système de conception graphique COGITO.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 59–70. Hermes, 90.

[Hubert & Perdreau 90] L. Hubert, G. Perdreau.
Software Factory : Using Process Modeling for Integration Purposes.
In *Proceedings of the First International Conference on Systems Integration (ICSI 90)*, pages 14–25, Morristown, New Jersey, April 23–26 1990. IEEE Computer Society Press.

[Hubner & Markov 86] W. Hubner, Z.I. Markov.
GKS based Graphics Programming in Prolog.
Computer Graphics Forum, (5):41–50, 86.

[Jabri & Skellern 89] A. Jabri, D.J. Skellern.
PIAF: efficient IC floor planning.
IEEE EXPERT, pages 33–45, Summer 89.

[Jonckers 85] V. Jonckers.
Knowledge Based Selection and Coordination of Specialised Algorithms.
In *Actes des 5ème Journées d'Avignon sur les Systèmes Experts et leurs Applications*. ADI, AFCET informatique, 85.

[Joseph 85] R. Joseph.
An expert system approach to completing partially routed printed circuit boards.
In *22nd IEEE Design Automation Conference*, pages 523–528, 85.

[Kamada & Kawai 91] T. Kamada, S. Kawai.
A general framework for visualizing abstracts objects and relations.
ACM Transactions on Graphics, 10(1):1–40, January 91.

[Kansy & others 87] K. Kansy et al.
DIN Contributions to GKS Review.
Technical Report 72, ISO/IEC JTC1/SC24, 87.

[Keen & Seviora 87] J.S. Keen, R.E. Seviora.
Expert systems for VLSI design : a survey and perspective.
In *Actes des 7èmes journées internationales sur les systèmes experts et leurs applications*, pages 1447–1457, Avignon, 87. AFCET Informatique.

[Kernighan & Pike 86] B. Kernighan, R. Pike.
L'environnement de programmation UNIX.
Inter Editions, Paris, 86.
Traduit par E. Horlait.

[Kouka 89] E.F. Kouka.
Génération automatique de plans de masse.
TSI, 8(6):557–569, 89.

[Krasner & Pope 88] G.E. Krasner, S.T. Pope.
A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system.
Journal of Object-Oriented Programming, 1(3):26–49, 88.

[Krasner 83] G. Krasner.
SMALLTALK 80, bits of history, words of advice.
Addison Wesley, 83.

[Krzewina 86] T. Krzewina.
Une étude comparative de PHIGS et de GKS.
In *Actes de MICAD 86*, pages 111–124. Hermes, 86.

[Laasri & al 88] H. Laasri, B. Maitre,, J.P. Haton.
Organisation, Coopération et exploitation des connaissances dans les architectures de blackboard : ATOME.
In *Actes des 8èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 371–390, Avignon, 88.

[Lamoitier 78] J.P. Lamoitier.
Le langage Fortran IV.
Dunod, Collection Université, 78.

[Lander & Lesser 88] S. Lander, V.R. Lesser.
Negotiation among cooperating experts.
In *Workshop on distributed Artificial Intelligence*, Lake Arrowhead Conference Center, May, 22–25 88.

[Latombe 77] J-C. Latombe.
Une application de l'intelligence artificielle à la conception assistée par ordinateur.
Thèse de Doctorat d'Etat, Université Scientifique et Médicale de Grenoble, Novembre 77.

[Latombe 85] J.C. Latombe.
The role of artificial Intelligence in CAD/CAM analysed through several examples.
In *Actes de PROLAMAT, 6ème conférence sur les logiciels CAO/DAO pour l'industrie manufacturière*, Paris, Juin 85.

[le Verrand 82] D le Verrand.
Le langage ADA, Manuel d'évaluation.

ACT Informatique, Paris, 85.

[le Verrand 82] D le Verrand.
Le langage ADA, Manuel d'évaluation.
Dunod Informatique, 82.

[Lebahar 84] J.C. Lebahar.
La CAO en architecture : ouverture et/ou réduction du champ des possibles en rapport avec l'étude des systèmes actuels.
In *Actes de MICAD 84.* Hermes, 84.

[Lebahar 85] J.C. Lebahar.
Etudes en CFAO : Architecture et Batiment.
Hermes, 85.

[Linton & al 89] M.A. Linton, J.M. Vlissides,, P.R. Calder.
Composing User Interfaces with Interviews.
IEEE Computer, pages 8–22, February 89.

[Mac Gee & al 86] B. Mac Gee, N. Giambasi, R.L. Bath, C. Delorme,, P. Roux.
A Framework for Computer Aided Design incorporating General Hierarchical Models.
In *Actes de MICAD 86.* Hermes, 86.

[Maitre & al 89] B. Maitre, H. Laasri, T. Mondot, F. Charpillet,, J.P. Haton.
Coordination de sources de connaissances opérant dans un univers incomplet et évolutif: étude et réalisation dans ATOME.
In *Actes des 9èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 237–251, Avignon, 89.

[Makino & Ishizuka 90] T. Makino, M. Ishizuka.
A hypothetical reasoning system with constraint handling mechanism and its application to circuit block synthesis.
In *Actes de PRICAI 90.* Japanese society for artificial intelligence, 90.

[Maloney & al 89] T.H. Maloney, A. Borning,, B.N. Freeman-Benson.
Contraint technology for user interface construction in THINGLAB II.
In *OOPSLA 89 proceedings*, pages 381–388, 89.

[Marrin 84] K. Marrin.
Lisp Developpements Tools help programmers slash CAE software production cycle.
EDN, pages 129–138, Novembre 84.

[Martin & Piette 87] D. Martin, F. Piette.
Clefs pour PC AT et Compatibles.
Editions du P.S.I., 87.

[Marty & al 91] J.C. Marty, F. Ramparany, M.S. Doize,, C. Jullien.
ACKnowledge: an integrated workbench supporting knowledge

Acquisition.
In *to appear in the proceedings of the Thirteenth Annual Meeting of The Cognitive Science Society*, Chicago, August 1991.

[MASM 85] MASM.
Microsoft Macro Assembler User's Guide, for the MS-DOS Operating System.
MICROSOFT Corporation, 85.

[Masseboeuf 86] C. Masseboeuf.
MULTISYM : un environnement de simulation plus qu'un simulateur.
In *Actes de MICAD 86*, pages 255–271. Editions HERMES, 86.

[Mellish & Clocksin 84] C.S. Mellish, W.F. Clocksin.
Programming in Prolog.
Springer Verlag, Berlin, 84.

[MICAD89] MICAD.
MICAD, 8ème Conférence sur la CFAO et l'Infographie, Paris, 89.
Hermes.

[MICAD90] MICAD.
MICAD, 9ème Conférence sur la CFAO et l'Infographie, Paris, 90.
Hermes.

[Mohan & Kayshap 88] L. Mohan, R.L. Kayshap.
An object-oriented knowledge representation for spatial information.
IEEE Transactions in Software Engineering, 14(5), May 88.

[Montalban 88] M. Montalban.
La CAO du futur avec l'intelligence artificielle.
ARCHES, revue annuelle des élèves de l'Ecole Nationale des Ponts et Chaussées, Juillet 88.

[Montgomery & Durfee 90] T.A. Montgomery, E.H. Durfee.
Using MICE to study intellignet dynamic coordination.
pages 438–444, 90.

[Morvan & Lucas 76] P. Morvan, M. Lucas.
Images et ordinateurs, introduction à l'infographie interactive.
Larousse, 76.

[Morvan & Lucas 79] P. Morvan, M. Lucas.
Travaux dirigés de l'école d'été d'informatique: la réalisation de logiciels graphiques interactifs.
Eyrolles, 79.

[Moulin 90] S. Moulin.
SMAC : Un outil d'acquisition de connaissances pour des problèmes de conception.

Thèse de doctorat de l' Ecole Nationale Supérieure de l'Aéronautique et de l'espace, Septembre 90.

[MS-C 87] MS-C.
Microsoft C 5.0 Optimizing Compiler, for the MS-DOS Operating System, User's Guide and Mixed-Language Programming Guide.
MICROSOFT Corporation, 87.

[MS-DOS 84] MS-DOS.
Microsoft MS-DOS, Operating System, Programmer's Reference Manual.
MICROSOFT Corporation, 84.

[MS-FORTRAN 87] MS-FORTRAN.
Microsoft FORTRAN 4.0 Optimizing Compiler, User's Guide, for the MS-DOS Operating System.
MICROSOFT Corporation, 87.

[MS-PASCAL 85a] MS-PASCAL.
Microsoft Pascal Compiler, for the MS-DOS Operating System, User's Guide.
MICROSOFT Corporation, 85.

[MS-PASCAL 85b] MS-PASCAL.
Microsoft Pascal Reference Manual.
MICROSOFT Corporation, 85.

[MS-QuickC 87] MS-QuickC.
Microsoft QUICK C Programmer's Guide, for IBM Personnal Computers and Compatibles.
MICROSOFT Corporation, 87.

[Myers & al 90] B.A. Myers, D.A. Guise, R. B. Dannenberg, B.V. Zanden, M.S. Kosbie, E.Pervin, A. Mickish,, P. Marchal.
GARNET : comprehensive support for graphical, highly interactive user interfaces.
IEE COMPUTER, pages 71–85, January 90.

[Myers 88] B.A. Myers.
Creating User Interfaces by Demonstration.
Perspective in computing. Academic Press, London, 88.

[Neveu & al 90] B. Neveu, B.Trousse,, O. Corby.
SMECI : an expert system shell that fits engineering design.
In *Third International Symposium on Artificial Intelligence*, pages 52–58, Monterrey N.L., MEXICO, 90.

[Newell 62] A. Newell.
Some problems of basic organization in problem-solving programs.
Technical Report RM-3283-PR, RAND Corporation, December 62.

[Newman & Sproull 81] W.R. Newman, R.F. Sproull.
Principles of Interactive Computer Graphics, 2nd Edition.
Mac Graw Hill, 81.

[Nilsson 83] N.J. Nilsson.
Principles of Artificial Intelligence.
Springer-Verlag, 83.

[Noyelle 84] Y. Noyelle.
Un système Prolog écrit en Smalltalk-80.
BIGRE No 41, pages 20–29, 84.

[Odawara & al 90] G. Odawara, T. Kamuro, K. Ihjima, T.Yoshino,, Y. Dai.
A rule-based placement system for printed wiring boards.
In *24Th ACM IEEE Design Automation Conference*, pages 777–785, 90.

[Orchampt 87] P. Orchampt.
Apports des Techniques d'Intelligence Artificielle en Conception Assisté par Ordinateur : Applications à la Schématique.
Thèse de Doctorat Informatique, Université Claude Bernard de Lyon 1, Juillet 87.

[Parent & Spaccapietra 88] C. Parent, S. Spaccapietra.
Gestion d'objets complexes avec des entités complexes.
In *Journées BD avancées : Bases de données déductives et bases de données orientées objet*, pages 119–153. Journées organisées par l'AFCET, Décembre 88.

[Pavlidis 82] T. Pavlidis.
Algorithms for Graphics and Image Processing.
Springer-Verlag, 82.

[Petit 90] C. Petit.
Langage objet dédié.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 47–56. Hermes, 90.

[PHIGS 88] PHIGS.
Programmer's Hierarchical Interactive Graphics System, Part 1 : Functionnal Description.
ISO/IEC 9592-1/1988 (E), Information Processing Systems, Computer Graphics, November 88.

[Pitrat 85] E. Pitrat.
Textes, ordinateurs, et compréhension.
Eyrolles, Paris, 85.

[Preux 87] P. Preux.
Smalltalk-80 - compléments aux livres bleu et orange.

Rapport interne, Laboratoire d'Informatique Fondamentale de Lille 1, Villeneuve d'Ascq, Novembre 87.

[PROLOG/P 85] PROLOG/P.
PROLOG/P version 2.00, Manuel d'utilisation MS-DOS.
Société CRIL, Paris, 85.

[Queinnec 84a] C. Queinnec.
langage d'un autre type : LISP, 3ème édition.
Eyrolles, Collection Micro-ordinateurs, Paris, 84.

[Queinnec 84b] C. Queinnec.
LISP, mode d'emploi.
Eyrolles, Paris, 84.

[Quintrand & others 85] P. Quintrand et al.
La CAO en architecture.
Hermes, Paris, 85.

[Ratajczyk 88] M. Ratajczyk.
Translateur de fichiers DXF en Métafichiers GKS.
Rapport de stage EUDIL, Lille, 88.

[Raykan & Fox 87] C.A. Raykan, M.S. Fox.
An investigation of opportunistic constraint satisfaction in space planning.
In *proceedings of IJCAI 87*, pages 1035–1038, 87.

[Retz Schmidt 88] G. Retz Schmidt.
Various views on spatial propositions.
AI Magazine, pages 95–105, Summer 88.

[Rich 83] E. Rich.
Artificial Intelligence.
McGraw-Hill, 83.

[Rieu 86] D. Rieu.
Nature, état, dynamicité de l'objet CAO.
In *Actes des journées de bases de données avancées*, Giens, Avril 86.

[Riffiet 86] J.M. Riffiet.
La programmation sous UNIX.
Mac Graw Hill, PARIS, 86.

[Rosenschein 84] J.S. Rosenschein.
Rational interaction : Coopération among intelligent agents.
Technical Report STAN-CS-851081, Standford Research Institute, 84.

[Rueher & al 85] M. Rueher, E. Coulon, J. Azencot, P. Maillot,, E.Tosan.
Un environnement graphique pour PROLOG.
In *Actes du MICAD 85*, Paris, 85. Hermes.

[Rueher 80] M. Rueher.
Démarche d'étude pour la réalisation de systèmes de Conception Assistée par Ordinateur.
Thèse de 3ème cycle, 1980.
Université Claude Bernard, LYON 1.

[Rychener 85] M.D. Rychener.
Expert systems for engineering design.
Expert Systems, 2(1):30–44, January 85.

[Sacerdoti 75] E.D. Sacerdoti.
The non linear nature of plans.
In *IJCAI Conference proceedings*, pages 206–214, 75.

[Sacerdoti 77] E.D. Sacerdoti.
A structure for plans and behaviors.
American Elsevier, 77.

[Schramel 86] F.J. Schramel.
What can AI contribute to Computer Aided Engineering.
In *Proceedings of the 2nd conférence on computer applications in production and engineering*, pages 915–930, Copenhague, may 86.

[Shneiderman 87] B. Shneiderman.
Designing the User Interface : Strategies for effective Human-Computer Interaction.
Addison Wesley, 87.

[Slimani 86] Y. Slimani.
Structures de données et langages non procéduraux, en informatique graphique.
Thèse de Docteur-Ingénieur, 86.
Université des Sciences et Techniques de Lille, Villeneuve d'Ascq.

[Smadja 86] M.D. Smadja.
Schématique : logiciels du 3ème type.
In *Actes de MICAD 86*, pages 624–634. Editions HERMES, 86.

[SMALLTALK/V 87] SMALLTALK/V.
Tutorial and Programming Handbook.
DIGITALK inc, 87.

[Smith 80] R.G. Smith.
The contract net protocol : high level communication and control in a distributed problem solver.
IEEE Transactions on Computers, 28(12):1104–1113, 80.

[Smith 85] R.G. Smith.
Report on the 1984 Distributed Artificial Intelligence Workshop.
Artificial Intelligence Magazine, 6(3):234–243, 85.

[Smith 86] B.M. Smith.
International efforts in product data exchange.
In *Actes de MICAD 86*, pages 743–751. Hermes, 86.

[Sriram 85] D. Sriram.
A bibliography on knowledge based expert systems in Engineering.
SIGART 85, pages 32–40, 85.

[Steele Jr 84] G.L. Steele Jr.
COMMON LISP Reference Manual.
Digital Press, 84.

[Stroustrup 86] B. Stroustrup.
The C++ Programming Language.
Addison Wesley, 86.

[Szekely & Myers 88] P.A. Szekely, B.A. Myers.
A user interface toolkit based on graphical objects and constraints.
In *OOPSLA 88 proceedings*, pages 36–45, September 25-30 88.

[TB-C 87] TB-C.
Turbo C User's guide, IBM version, for PC XT, AT and true compatibles.
BORLAND International inc, 87.

[TB-PASCAL 85] TB-PASCAL.
Turbo Pascal 3.0., Manuel de Référence MS-DOS.
société FRACIEL, Tours, 85.

[TB-PROLOG 86] TB-PROLOG.
Turbo PROLOG, Guide de l'utilisateur.
BORLAND International, Scotts Valley, U.S.A., 86.

[Thomas & Lalonde 85] D.A. Thomas, W.R. Lalonde.
ACTRA : the design of an industrial fifth generation Smalltalk system.
Technical Report SCS-TR-71, School of Computer Science, Carleton University, OTTAWA, Ontario, April 85.

[Thomas & Lalonde 86] D.A. Thomas, W.R. Lalonde.
ACTORS in Smalltalk Multiprocessors : a case for limited parallelism.
Research Report SCS-TR-91, University of Ottawa, Carleton, May 86.

[Thoraval & al 90] M. Thoraval, P. Mazas,, J.P. Chatenet.
ARCHIX : un système d'aide à la conception d'automobile.

In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 163–172. Hermes, 90.

[Tinel & al 84] Y. Tinel, B. Six,, D. Verez.
Introduction aux facilités de programmation sous Multics, Cours 1 à 5.
Technical report, C.I.T.I : Centre InterUniversitaire de Traitement de l'Information, Domaine Universitaire Scientifique de Villeneuve d'Ascq, France, Novembre 84.

[Trousse 87] B. Trousse.
EXSAT : Système expert configurateur de satellites de télécommunications.
In *Actes des 7èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 335–350, Avignon, 87.

[Trousse 88] B. Trousse.
XCAD : a virtual CAD object-oriented solid modeler for an eXpert system shell.
In *3rd Internationa Conference on CAD/CAM, robotics and factories of the Future*, Southfield, Michigan USA, August 88.

[Trousse 89] B. Trousse.
Coopération entre Systèmes à base de Connaissances et Outils de CAO : l'environnement Multi-Agents ANAXAGORE.
Thèse de Doctorat Informatique, Université de Nice, Décembre 89.

[Trousse 90a] B. Trousse.
Bénéfices d'une approche orientée objet pour un environnement de CAO.
In Y. Gardan editor, *Etudes en CFAO : outils et applications de l'I.A. en CFAO*, pages 34–44. Hermes, 90.

[Trousse 90b] B. Trousse.
Cooperation between knowledge based systems and CAD tools : the ANAXAGORE environment.
In *Third International Symposium on Artificial Intelligence*, pages 83–89, Monterrey N.L., MEXICO, 90.

[Tsang & al 90] J.P. Tsang, B. Wrobel,, N. Pfeiffer.
Un raisonnment fonctionnel pour automatiser le processus de conception.
In *Actes des 10èmes Journées Internationales sur les Systèmes experts et leurs applications*, pages 151–166, Avignon, 90.

[Tsang 87] J.P. Tsang.
Planification par combinaison de plans : application à la génération de gamme d'usinage.
Thèse de doctorat, 87.
INPG, Grenoble.

[Tsuchida & al 89] M. Tsuchida, H. Yoshimura,, Y. Ohiwa.
Expert system of placement and routing for print circuit board.
In *Proceedings of the third international conference on computer applications in production and engineering (CAPE 89)*. Elsevier Publishing Company, 89.

[Valduriez 87] P. Valduriez.
Objets complexes dans les S.G.B.D. relationnels.
Technique et Sciences Informatiques, 6(5):405–417, 87.

[Valette & Foisseau 83] F.R. Valette, J. Foisseau.
Bases de données en CAO.
Hermes, Paris, 83.

[VAXLISP 84] VAXLISP.
VAX LISP 1.0, User's Guide.
DIGITAL Corporation, 84.

[VAXLISP 86] VAXLISP.
VAX LISP 1.1, Release Notes.
DIGITAL Corporation, 86.

[Verdin 88] E. Verdin.
Metadesign GKS 4.0 : Interface langage Alsys-Ada, pour MS-DOS.
Société METADESIGN, Lille, 88.

[Verroust 90] A. Verroust.
Construction d'objects géométriques définis par des contraintes.
BIGRE + GLOBULE No 67, pages 62–73, Janvier 90.

[Vogel 85] C. Vogel.
ICARE.
Technical Report H/ 26-10-85, IIRIAM GRTC-CNRS, Marseille, 85.
rapport interne.

[Voirol & Piguet 89] C. Voirol, C. Piguet.
Système expert pour la conception automatique de modules VLSI.
TSI, 8(6):509–523, 89.

[Walker & al 88] E.L. Walker, M. Herman,, T.Kanade.
A framework for representing and reasonning about three dimensional objects for vision.
AI Magazine, pages 47–58, Summer 88.

[Webster 89] B.F. Webster.
The NEXT Book.
Addison-Wesley, 89.

[Winston & Horn 84] P.H. Winston, B.K.P. Horn.
LISP, 2nd edition.
Addison Wesley, 84.

[Wirth 82] N. Wirth.
Programming in Modula-2, 2nd edition.
Springer Verlag, 82.

[Worden & others 86] R.P. Worden et al.
Co-operative expert systems.
In *ECAI Conference Proceedings*, pages 319–334, 86.

[Yoshimura & al 90] H. Yoshimura, M. Tsuchida, Y. Oshiwa, Y. Nishimura, S. Midra amd H. Uemura,, H. Ade.
knowledge based placement and routing system for printed circuit board.
In *Actes de PRICAI 90*, pages 41–48. Japanese society for artificial intelligence, 90.

Annexe II - 0
Le système graphique GKS

1. Historique et description élémentaire

A la fin des années 70, décennie qui a vu la maturité des techniques de base en infographie [Newman & Sproull 81, Foley & Van Dam 84], les concepteurs de logiciels et constructeurs de matériels infographiques ont commencé un travail de concertation en vue de l'émergence d'une norme en infographie 2D, au sein des groupes nationaux (ANSI, DIN, BSI, AFNOR) et internationaux (ISO).

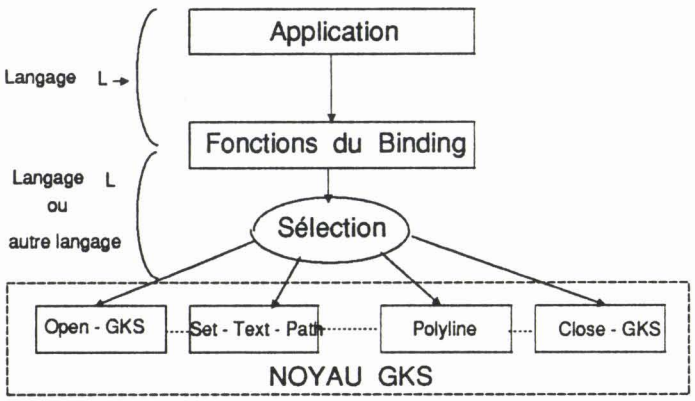
Après une bataille d'experts entre américains (groupe ANSI), partisans du "CORE system", et européens, partisans de "GKS", le standard retenu par l'ISO fut GKS, émanant initialement du groupe allemand (DIN).

GKS, abréviation de "Graphical Kernel System", est un standard destiné à l'infographie en deux dimensions. *Il regroupe un ensemble de normes, de deux catégories distinctes* :

- Le **noyau graphique** [GKS85], décrivant les fonctionnalités du standard.
- Les **interfaces langage**, encore appelées **Binding**, qui visent à formaliser et standardiser l'utilisation des implantations de GKS sur différentes machines et systèmes d'exploitation, afin d'éviter une profusion d'interfaces différentes provenant des constructeurs, qui nuiraient de fait à la portabilité des applications graphiques. Les binding ne concernent que des langages faisant déjà l'objet d'un processus de normalisation international (FORTRAN, C, PASCAL, ADA).

Schématiquement, le programmeur d'application graphique, qui utilise un noyau GKS, réalise un programme, codé en un langage quelconque que nous appellerons L, programme qui appelle les primitives décrites (syntaxiquement et sémantiquement) dans l'interface langage pour L. Ensuite, l'édition des liens sur la machine cible, que nous appellerons M, effectue les connections entre les fonctions du binding B, utilisées par son programme, et celles correspondantes (directement ou après traitement et "filtrage") dans le noyau GKS disponible sur M.

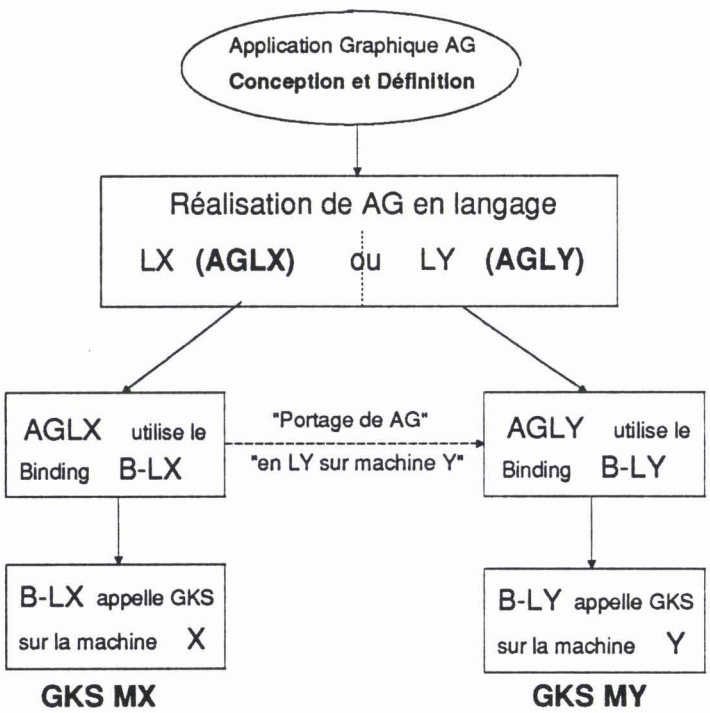
Ceci donne l'architecture logicielle suivante pour les programme d'application graphique :



Architecture logicielle d'un programme GKS

Fig 1

Le but de GKS est de rendre les applications graphiques portables sur tous types de matériels, pour un maximum de langages utilisés dans l'industrie, langages procéduraux surtout. Voici un schéma de portage d'une application graphique AG sur des machines MX et MY, portage du langage LX vers le langage LY.



Application graphique, Bindings GKS, noyaux GKS

Fig 2

Autrement dit, un programmeur qui désire porter son application, écrite en langage LX sur une machine MX, vers une machine MY en utilisant un langage LY, devra :

- réécrire son application, avec l'implantation LY existant sur MY.
- utiliser un noyau GKS disponible pour MY.

Si GKS a permis une bonne formalisation des modèles en infographie 2D (espaces (WC, NDC, DC), inputs, primitives et attributs liés...), nous devons constater que ses insuffisances (peu de primitives de base, pauvreté de celles-ci, défauts de conception pour la segmentation, définition trop liée aux machines et matériels existants), et que cette formalisation nécessaire (trois espaces, modèles des inputs graphiques, austérité de la norme), ont rebuté les programmeurs, et particulièrement nui à la diffusion et à l'utilisation de cette norme qui aujourd'hui semble définitivement réservée aux spécialistes, aux grands comptes (armée, ministères publics), et aux applications sur mini-ordinateurs. C'est pourquoi nous pensons que la norme GKS a subit un échec, au niveau de sa vulgarisation, et de son utilisation en micro-informatique, d'autres standards de fait (AUTOCAD...) devenant les plus utilisés.

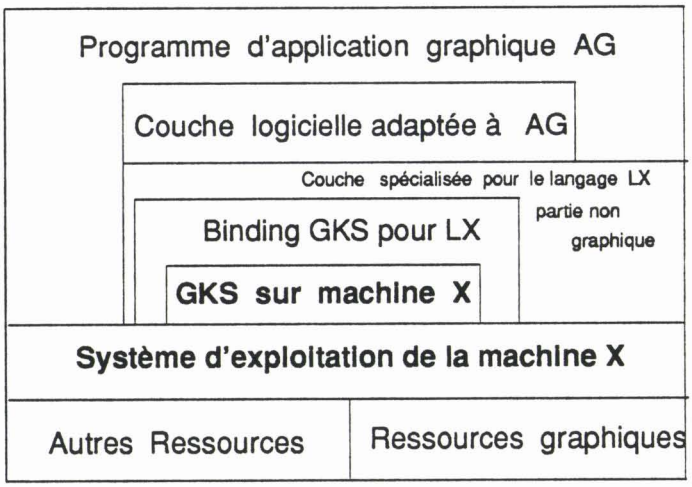
Sans décrire en détail les modèles de GKS, nous explicitons quelques notions de base permettant d'écrire des applications élémentaires. Le but est ici de rendre le lecteur plus familier avec le déroulement d'un programme de base GKS.

2. Modèle en couches de GKS

D'un point de vue logiciel de base, une application graphique AG, écrite avec GKS, est construite de la manière suivante :

- le programme d'application (écrit en langage LX) utilise trois parties spécialisées bien définies :
  1 - le système d'exploitation de la machine cible MX,
  (écriture, destruction, création de fichiers; commandes systèmes...)
  2 - l'implantation de LX sur MX,
  ces deux parties sont indépendantes et ne font pas appel au graphique.
  3 - la partie utilisant des fonctionnalités graphiques,
  (création de menus, saisies graphiques, sorties graphiques,
  structures de données CAO...)
- cette dernière partie (3) est appelée couche orientée application, et utilise :
  1 - des fonctions de l'implantation de LX sur MX
  2 - les primitives de l'interface langage normalisée LX pour GKS (GKSLX)
- l'implantation de LX sur MX utilise le système d'exploitation de MX.
- GKSLX utilise le système d'exploitation, en particulier pour faire appel aux ressources systèmes et graphiques (écrans, traceurs, appareils d'entrée).

Nous pouvons représenter tout ceci par le schéma suivant :



(deux parties adjacentes communiquent entre elles)

Modèle en couches de GKS (modifié de l'ISO 7942)

Fig 3

3. Poste de travail GKS

Le **poste de travail** GKS, ou **workstation** en anglais, est l'abstraction d'un ensemble de matériels, et de logiciels si nécessaire, permettant l'une des fonctions suivantes:

- les sorties graphiques (catégorie **OUTPUT**),
- les entrées graphiques (catégorie **INPUT**),
- les 2 précédentes (catégorie **OUTIN**),
- la manipulation d'images graphiques en mémoire (**WISS**),
- la manipulation de fichiers décrivant des images GKS, fichiers appelés *métafichiers* en lecture (**MI**), ou en écriture (**MO**).

Les postes capables de sorties physiques (OUTPUT, OUTIN), par exemple un traceur (OUTPUT), ou un écran avec une souris (OUTIN), disposent d'un **espace d'appareil** (DEVICE COORDINATES) cartésien (O, x, y), aussi appelé **espace de visualisation**, par exemple pour un écran. Les unités de ces appareils peuvent être des mètres, c'est le cas des traceurs, ou être immatérielles, comme c'est le cas d'appareils de type écran, où les unités sont exprimées en pixels élémentaires.

Les postes d'entrée (INPUT), ne disposent pas d'unité particulière, sauf pour les tablettes à digitaliser, dont les unités peuvent être des mètres ou des pouces.

Les catégories de type WISS, MI, MO, ne disposent pas d'espace de visualisation.

La manipulation des appareils d'entrée physique graphique est matérialisée par un écho, par exemple un réticule sur l'écran pour une saisie de points, ou une valeur digitale pour une saisie de valeur numérique, ou un son sonore pour une sélection parmi un menu (comme par exemple pour l'entrée de type choix (CHOICE)).

Chaque poste de travail est décrit, de façon interne à GKS, et sous forme de notice fournie au programmeur, par sa **table de description** (WORKSTATION DESCRIPTION TABLE), qui définit ses capacités et caractéristiques GKS, par exemple : la taille de l'espace DC en unités locales, le nombre d'attributs disponibles pour chaque primitive de base, etc... En cours d'exécution de programme, la **table d'état du poste** (WORKSTATION STATE LIST) décrit à tout instant les caractéristiques GKS du poste utilisé.

Pour utiliser un poste de travail, le programmeur "ouvre" celui-ci par la primitive OPEN WORKSTATION ; pour réaliser effectivement l'action d'une primitive graphique sur celui-ci, le programmeur active le poste par la primitive ACTIVATE WORKSTATION ; il peut ensuite réaliser autant d'actions graphiques qu'il le désire, jusqu'à désactiver le poste par la primitive DEACTIVATE WORKSTATION. Avant de terminer son exécution, le programme doit fermer le poste par la primitive CLOSE WORKSTATION. On peut répéter le cycle *ouverture, activation... désactivation, fermeture*, autant de fois qu'on le désire, à l'intérieur d'un programme.

Nous ne considérons ici que les stations de catégorie OUTIN, qui sont à la base de toute interaction graphique.

4. Primitives de sortie.

Pour réaliser des sorties, matérialisées par des dessins, sur le(s) poste(s) de travail actifs, on dispose de cinq primitives de base :

- tracé de **segment**, ou de **suite de segments contigüs** (POLYLINE),
- **marquage de position**(s) dans l'espace, par des symboles (POLYMARKER),
- **texte graphique** (TEXT),
- **polygone et remplissage de polygones** (FILL AREA),
- **matrice de cellules** rectangulaires (CELL ARRAY),

Les primitives les plus utilisées sont : POLYLINE, TEXT, FILL AREA.

On a aussi la possibilité d'utiliser des primitives de sortie hors-normes, appelées **primitives de tracés généralisées** (GENERALIZED DRAWING PRIMITIVE), et dont la contraction admise dans la norme est **GDP**, si l'implantation de GKS sur MX, ainsi que le poste de travail utilisé le permettent. Ce seront par exemple des tracés de cercles, arcs de cercle, ellipses, courbes de Bézier, splines... GKS fournit un protocole précis, pour chaque interface langage, destiné à l'utilisation des GDP.

A chaque primitive est associé un ensemble d'attributs de tracé, modifiables par sélection et affectation.

Pour la primitive POLYLINE, ces attributs sont :

- le type de trait (continu, tireté, mixte, pointillé...),
- le coefficient d'épaisseur de trait (un réel),
- la couleur du trait (un indice),

Pour la primitive TEXT, on a :

- la police utilisée, associée à sa qualité (un indice),
- le coefficient d'agrandissement du texte (une valeur réelle),
- l'espace inter-caractères (réel),
- le vecteur d'orientation des caractères (couple (x,y) non nul),
- le sens d'écriture, par rapport à la droite définie par le vecteur précédent (gauche, droit, bas, haut),
- l'alignement du texte par rapport à la position d'écriture du texte (horizontalement (gauche, centré, droit), verticalement (haut, crête, centré, base, bas)),
- la hauteur des caractères (réel).

Pour la primitive FILL AREA, ces attributs sont :

- le type de remplissage du polygone (contour, plein, hachuré, motif),
- un indice si l'attribut précédent est hachuré ou motif (indice),
- la couleur, pour le remplissage contour, plein, hachure (indice),
- la taille de motif (couple (x,y) non nul) pour le style motif.
- le point de référence pour les motifs (couple (x,y)).

Les attributs des GDP sont ceux de la primitive de base de même nature; par exemple ceux de POLYLINE pour les cercles, ellipses, splines.

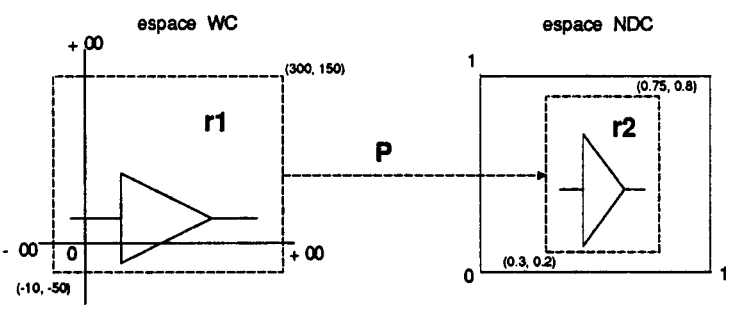
5. Espaces de GKS

Pour assurer l'indépendance des programmes vis à vis du matériel, pour que le programmeur puisse définir un modèle abstrait de son application, et pour que des images produites par GKS puissent être reprises par d'autres programmes d'application, GKS utilise trois espaces euclidiens :

- L'**espace WC** (WORLD COORDINATES) ou **espace utilisateur**, indépendant de toute implantation. c'est l'espace euclidien habituel des mathématiciens, les coordonnées sont dans l'intervalle]- oo , + oo[.
- L'**espace NDC** (NORMALIZED DEVICES COORDINATES) ou **espace de coordonnées normalisées**. C'est le sous ensemble [0, 1] x [0, 1] de R^2, espace où l'on commence à lier les valeurs abstraites de WC avec le matériel, grâce aux "WORKSTATION WINDOWS". Il permet, entre autres choses, la portabilité de la segmentation, des métafichiers, et la correspondance avec les espaces physiques en DC.
- Les **espaces DC** (DEVICE COORDINATES), ou **"espaces de visualisation"**, déjà cités précédemment; chaque poste de travail est associé à l'un d'eux. Ce sont des sous-espaces de R^2, [0, nX] x [0, nY], où nX et nY sont les valeurs maximales, en unités locales, des valeurs de coordonnées accessibles. Ces espaces sont en correspondance directe avec les dispositifs physiques de visualisation, sur lesquels seront affichés les tracés graphiques (primitives de sortie et échos de certaines entrées).

On utilise ces espaces de la manière suivante :

- on définit une ou plusieurs *projections affines Pi de WC vers NDC*, appelée(s) **transformation(s) de normalisation**, en choisissant un rectangle r1 de WC qui est projeté dans un rectangle r2 de NDC, rectangles qui sont définis librement par le programmeur, sauf r2 qui doit être inclus dans [0,1] x [0,1]. Les objets abstraits



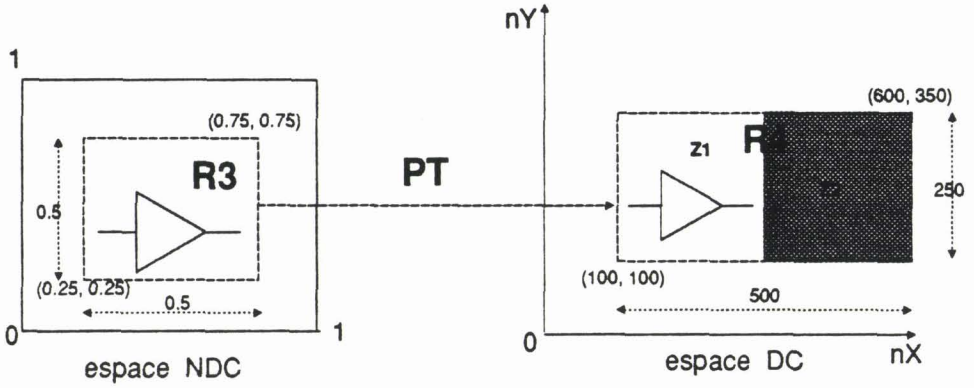
Le contenu du rectangle r1 est totalement projeté dans la zone définie par r2 , grâce à la projection P, appelée aussi transformation de normalisation; on peut définir autant de projections Pi que le permet l'implantation GKS utilisée. P ne conserve pas les formes.

Transformation de normalisation GKS

Fig 4

géométriques de WC subissent alors les transformations de formes, dues à cette projection : agrandissement, rétrécissement, anamorphose. La figure II.4 explique visuellement ce mécanisme.

- Pour chaque poste de travail, on définit sa *projection affine PTi de NDC vers DC*, appelée **"transformation de poste de travail"**, d'une façon similaire à la précédente : choix d'un rectangle R3 de NDC, projeté cette fois de façon *isotropique* dans un rectangle R4 de DC, rectangles choisis au gré de l'utilisateur. Toutefois, R3 doit être



Le contenu du rectangle R3 est projeté dans R4 par PT1.
Z1: zone gauche de R4, utilisée par, et définissant PT1.
Z2 : zone droite de R4 inutilisée par GKS, pour définir PT1.
PT est une transformation qui conserve les formes.

exemple de transformation de poste de travail GKS

Fig 5

inclus dans l'espace NDC, et R4 dans l'espace DC. GKS prend de R4 la partie la plus grande, qui doit cependant être homothétique à R3, ceci pour disposer d'une image dans NDC qui reflète l'affichage à l'écran.

La bonne conception du modèle abstrait choisi pour le programme, le choix judicieux des rectangles de WC, NDC et DC, conditionne l'efficacité, la robustesse, la maintenabilité de l'application graphique. C'est un élément essentiel de programmation GKS.

6. Découpage

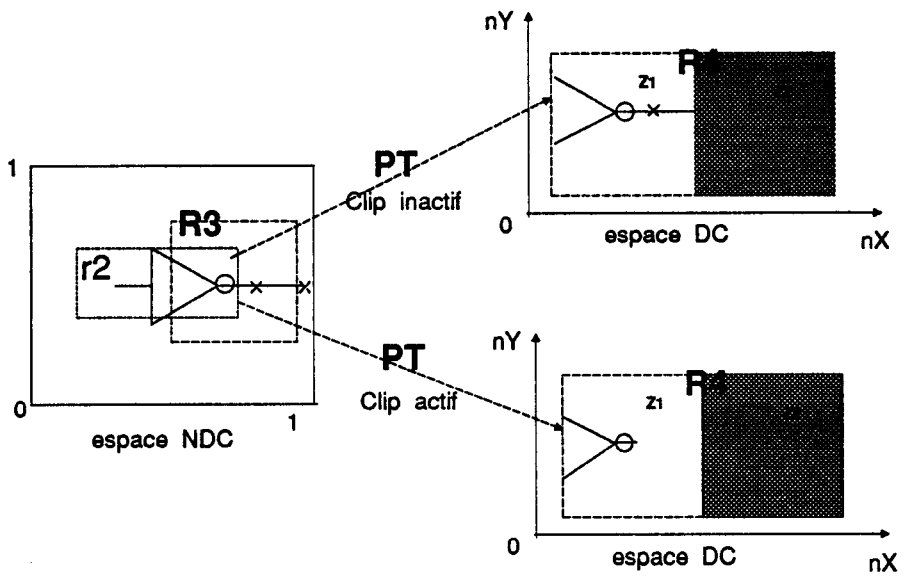
Sans entrer dans les subtilités du "clipping" (découpage) de GKS, qui conditionne la réussite technique de l'application, en particulier son efficacité, nous décrivons ici sa fonction essentielle. Nous pensons que la mauvaise utilisation de GKS, en particulier son utilisation aveugle, sans en comprendre les mécanismes profonds, est responsable pour une bonne part de son échec auprès des programmeurs ; la réalisation interne (dans GKS) du découpage et externe (dans l'application) des fonctions de découpage, doit faire l'objet d'une étude très poussée.

Découpage automatique dans DC :

- Pour chaque poste de travail, GKS découpe les sorties graphiques de manière à ce qu'aucun tracé ne sorte du rectangle R4 cité précédemment. Cette découpe fonctionne toujours.

Découpage basculable dans NDC :

- On dispose d'un indicateur de découpe que l'on peut modifier à volonté. Si celui-ci est faux, la totalité du contenu de R3, projetée en DC, sera visible dans R4, sinon la partie (r2 inter R3) de NDC, projetée en R4, sera visible.



Les effets du découpage (clipping) en GKS

Fig. 6

Le découpage, fonction essentielle de GKS, permet une sélection de l'affichage adaptée au déroulement du programme.

7. Entrées graphiques

Les interactions avec l'utilisateur, ou le milieu extérieur, se font grâce aux appareils d'entrée. Il en existe 5 classes en GKS :

- les **releveurs de coordonnées** (LOCATOR) permettent de désigner une position dans un espace à deux dimensions (donc dans WC). Ce sont les moyens d'entrée les plus importants et les plus utilisés de GKS.
- les **releveurs de suites de coordonnées** (STROKE), extensions des précédents à un groupe de positions, permettent en outre d'envisager la saisie de polygone ou la digitalisation.
- les **releveurs de valeurs numériques** et de **chaînes de caractères** (VALUATOR et STRING).
- les **releveurs de choix** (CHOICE), qui, associés à des valeurs entières, permettent des effets spéciaux (désigner des cases de menus, définir des menus à icônes...).

- les **désignateurs de segments** (PICK), qui permettent de désigner un groupe de primitives formant une image.

Le fonctionnement des entrées se déroule de la manière suivante :

- initialisation de la communication avec l'appareil physique associé (par exemple une souris) par l'intermédiaire des primitives INITIALISE-(*classe d'entrée*)...
- choix du mode d'entrée pour GKS : sur requête d'utilisateur (REQUEST), par échantillonnage (SAMPLE), par liste d'échantillons (EVENT).
- déclenchement de la fonction autant de fois qu'on le désire : REQUEST -(*classe d'entrée*), SAMPLE -(*classe d'entrée*) , GET -(*classe d'entrée*).

8. Segmentation

Un "segment" GKS mémorise, par une liste d'appel de primitives de sortie, et attributs de sortie, un résultat graphique de l'espace NDC représentant tout ou une partie d'image. Cela permet donc la manipulation d'objets graphiques. Pour des raisons essentiellement historiques (matériels graphiques existant), ce sont en fait des structures similaires aux "display-list" [Newman & Sproull 81], qui sont intégrées à GKS, pour lesquelles on ne peut pas faire de manipulation interactive, et sans signification symbolique possible. Dans les standards ISO ultérieurs, en PHIGS [PHIGS 88] le "Centralized Structure Store System", en CGI [CGI 88] les segments éditables et les figures fermées, en GKS 9X [GKS-add 87] les segments éditables, permettent en partie la manipulation et l'édition interactive d'objets graphiques. Nous ignorons volontairement les segments GKS dans notre travail, car inadaptés à la CAO. C'est le programme d'application, ou le SCAO, qui doit gérer ses propres structures éditables, avec tous les problèmes conceptuels et de portabilité que cela implique.

9. Divers

Pour faciliter le travail du programmeur et assurer une portabilité accrue des programmes d'application, GKS fournit des primitives de requête (INQUIRE). Celles-ci se divisent en trois catégories :

- les **requêtes sur les stations de travail** physiques au sens GKS. Elles permettent de connaître la table de description du poste de travail choisi, afin de moduler le programme suivant le matériel utilisé.
- les **requêtes sur l'état d'un poste de travail** en cours de déroulement d'un programme. Elles permettent de connaître les éléments de la liste d'état courante du poste de travail désigné. Ceci permet en particulier de concevoir l'application par modules indépendants les uns des autres.
- Les **requêtes sur l'état et les attributs du système** GKS utilisé, en cours de déroulement de programme.

Certaines requêtes sont très utiles, *inquire-maximum-display-surface-size* par exemple, mais beaucoup d'entre elles n'apportent que peu d'éléments intéressants au programmeur, car les tables de description et les listes d'état ne fournissent que des indices de table concer-

nant les attributs de primitives de sortie, alors que ce qui intéresse le programmeur est l'aspect visuel réel d'une sortie physique sur un poste donné, ce qui n'est pas disponible en GKS.

Les métafichiers GKSM (stations MI et MO) permettent le stockage et le transport d'images graphiques issues d'un programme P1 vers un autre programme P2, tous deux écrit en GKS. Ils ont une structure similaire aux segments, avec la possibilité d'ajouter des enregistrements utilisateurs et une entête décrivant le métafichier. Ceci ne fait pas partie du standard ISO GKS, et le comité de normalisation a conçu à cet effet une norme spécialisée appelée CGM (Computer Graphic Metafile) [Arnold & Bono 88], qui est capable de décrire toutes les images provenant des différents standards graphiques (GKS, GKS3D, PHIGS, CGI...). La simplicité de la structure des métafichiers GKSM permet la conversion d'images décrites par des standards de transfert de fichiers CAO, tels que IGES, SET, DXF [Ratajczyk 88], en images assimilables par les programmes écrits en GKS. L'inverse n'est cependant pas aussi aisé, du fait de la pauvreté des primitives de base GKS.

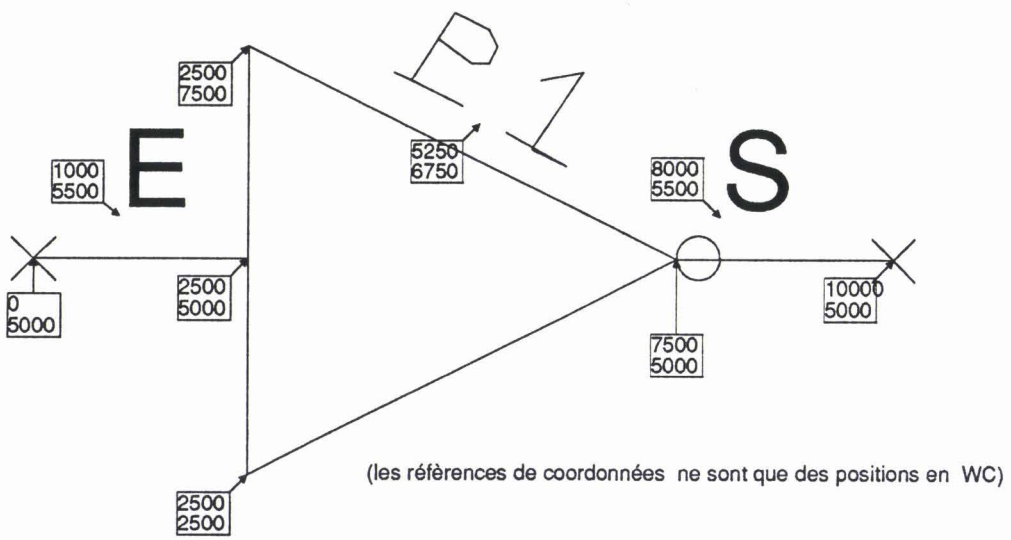
Avec ce rapide tour d'horizon du système graphique GKS, nous espérons que le lecteur aura assimilé les concepts de base qui permettent de programmer avec ce logiciel de base graphique. En guise de conclusion, nous présentons un petit programme GKS, écrit en langage informel.

10. Un exemple de programme GKS

Le programme GKS ci-après réalise les actions suivantes :

- ouverture et activation d'un poste de travail,
- paramétrage de transformation de normalisation (projection WC - DC) et de poste de travail (projection NDC - DC),
- tracé graphique d'une porte "not", avec les primitives de base (procédure afficher-porte),
- zoom avant (grossissement), et arrière (rapetissement), sur cet objet, par utilisation d'un releveur de coordonnées,
- fermeture du poste, de GKS...

Ce programme est écrit de façon informelle en "pseudo-code", et utilise directement les primitives telles qu'elles sont décrites dans la norme GKS. Voici la figure géométrique de WC que reproduira le programme, en totalité ou en partie, suivant l'échelle de zoom :



La porte "not", considérée en WC

Fig 7

Le zoom s'effectue ici par simple modification de la définition de la projection WC vers NDC; on prend réellement une portion contenue (agrandissement), ou contenant (rapetissement), le modèle en WC...

Remarque : le lecteur verra sur cet exemple (textes sources qui suivent), qu'en GKS, la notion d'objet graphique n'existe quasiment pas, et qu'on reste à un niveau très bas de programmation, de toute manière impérative.

```
programme ZOOM_SUR_PORTE_NOT
-- initialisation et déclaration des variables nécessaires
.....
debut
-- ouverture de GKS avec réservation de 10000 blocks mémoire;
-- le fichier sur lequel seront signalées les erreurs est la console.
   OPEN_GKS(STANDARD_OUTPUT,10000);
-- ouverture d'un poste de travail W1, désigné par son type
-- de connexion ("EGA"), poste de type OUTIN1 ( = catégorie OUTIN).
-- activation et effacement de la surface de visualisation du poste.
   OPEN_WORKSTATION(W1,"EGA",OUTIN1);
   ACTIVATE_WORKSTATION(W1);
   CLEAR_WORKSTATION(W1,allways);
-- paramètrage du 'Pipe-line' de transformations GKS.
   -- définition de la transformation de normalisation numéro 1
      SET_WINDOW( 1, 0.0, 10000.0, 0.0, 10000.0); -- rectangle r1 dans WC (cf. fig. 4)
      SET_VIEWPORT( 1, 0.0, 1.0, 0.0, 1.0);      -- rectangle r2 dans NDC (cf. fig. 4)
      SELECT_NORMALIZATION_TRANSFORMATION(1);
   -- définition de la transformation du poste de travail W1
      SET_WORKSTATION_WINDOW(W1,0.0,1.0,0.0,1.0);
            -- rectangle R3 dans NDC (cf. fig. 5)
            -- REMARQUE: les définitions de r2 et R3 impliquent ici
            -- que la totalité de NDC sera visible à l'écran (car r2 = R3 = NDC)
      SET_WORKSTATION_VIEWPORT(W1, 0.0, 400.0, 0.0, 400.0);-- rectangle R3 dans NDC (cf. fig. 5)
            -- carré bas gauche de 400 unités de coté

-- initialisation du releveur de coordonnées
   SET_VIEWPORT_INPUT_PRIORITY(1,0,higher);
   INITIALIZE_LOCATOR(
      W1,                -- pour le poste de travail créé
      LOCATOR1,          -- choix du releveur 1, fourni par l'implantation GKS utilisée
      5000.0,5000.0,     -- position initiale (avant saisie) en WC
      5,                 -- choix de l'écho du releveur (rectangle élastique)
      0.0, 450.0, 0.0, 450.0,  -- zone d'écho, dans laquelle l'utilisateur contrôle
                               -- complétement (écho et saisie) le releveur
      SUP_REC);          -- données supplémentaires dépendant du releveur choisi
   SET_LOCATOR_MODE( W1, LOCATOR1, request, echo-on); -- mode requête, écho actif
-- premier affichage du dessin de la porte "not"
   AFFICHER_PORTE_NOT;
```

```
-- affecte la position initiale de saisie
   POSITION_INITIALE.x := 0.0 ;
   POSITION_INITIALE.y := 0.0 ;
-- boucle de saisie
répéter
   effacer_ecran_alphanumerique;
   ecrire( STANDARD_OUTPUT, 'choisissez une zone rectangulaire de ZOOM');
--- saisie de la zone à visualiser
   REQUEST_LOCATOR(W1,LOCATOR1,
                   ETAT_EN_SORTIE,TRANSFORMATION_TROUVEE,POSITION_EN_WC);
   SAISIE_VALIDE := vrai;
--- calculs de la nouvelle fenêtre à visualiser
   si  ETAT_EN_SORTIE <> rien
      CLEAR_WORKSTATION(W1,allways);
      -- bords droits et gauche
      si  POSITION_EN_WC.x > POSITION_INITIALE.x
         DROITE := POSITION_EN_WC.x;
         GAUCHE := POSITION_INITIALE.x;
      sinsi POSITION_EN_WC.x <> POSITION_INITIALE.x
            GAUCHE := POSITION_EN_WC.x;
            DROITE := POSITION_INITIALE.x;
      sinon SAISIE_VALIDE := faux;
      fsi
--- faire la même chose avec les bords HAUT et BAS
      si  SAISIE_VALIDE
         -- modification de la transformation 1 pour visualiser le contenu
         -- de la zone saisie avec le releveur
            SET_WINDOW(1,GAUCHE,DROITE,HAUT,BAS);
         -- redessin de la porte agrandie
            CLEAR_WORKSTATION(W1,allways);
            AFFICHER_PORTE_NOT;
         -- réinitialisation du releveur au centre de la fenetre
            POSITION_INITIALE := ( (GAUCHE + DROITE)/2, (HAUT+BAS)/2 );
            INITIALISE_LOCATOR(
               W1,LOCATOR1,POSITION_INITIALE,5,0.0,450.0,0.0,450.0,SUP_REC);
      fsi
   fsi
jusque ETAT_EN_SORTIE = rien;                    fin ZOOM_SUR_PORTE_NOT
```

```
procédure AFFICHER_PORTE_NOT;
    -- dessine la porte sur l'écran graphique. Les coordonnées citées sont toutes en WC
    -- les tableaux (ou listes) sont représentés par "[.. , .. , ..]", les points par "(vx, vy)"
début
-- dessins des pattes d'entrée et sortie
    SET_POLYLINE_COLOUR_INDEX(1);
    SET_LINETYPE(1);
    SET_LINEWIDTH_SCALE_FACTOR(1);
-- patte d'entrée du not
    POLYLINE( 2, [ (0.0, 5000.0), (2500.0, 5000.0) ] );
-- patte de sortie du not
    POLYLINE( 2, [ (7500.0, 5000.0), (1000.0, 5000.0) ] );
-- le corps du not, tracé avec "Fill area"
    SET_FILL_AREA_COLOUR_INDEX(1);
    SET_FILL_AREA_INTERIOR_STYLE(contour);
    FILL_AREA( 3, [ (2500.0, 7500.0), (7500.0, 5000.0), (2500.0, 0.0) ] );
-- le petit "rond" du not et les deux croix
    SET_POLYMARKER_COLOUR_INDEX(1);
    SET_MARKER_TYPE(4);
    SET_MARKER_SIZE_SCALE_FACTOR(2);
    POLYMARKER( 1, [ (7500.0, 5000.0) ] );
    SET_MARKER_TYPE(5);
    POLYMARKER( 2, [ (0.0, 5000.0), (10000.0, 5000.0) ] );
-- textes de la porte not
    SET_TEXT_FONT_AND_PRECISION((1,haute));
    SET_CHARACTER_EXPANSION_FACTOR(2);
    SET_CHARACTER_SPACING(1);
    SET_TEXT_COLOUR_INDEX(1);
    SET_CHARACTER_HEIGHT(1000.0);
    SET_TEXT_PATH(right);
-- les caractères E et S, désignant les entrées et sorties
    SET_CHARACTER_UP_VECTOR((0.0,1.0));--caractères verticaux
    SET_TEXT_ALIGNMENT(LEFT,BOTTOM);
    TEXT((1000.0,5500.0),'E');
    TEXT((8000.0,5500.0),'S');
-- la chaîne 'P1'
    SET_CHARACTER_UP_VECTOR( (0.5, 0.866) ); -- caractères inclinés à 30 degrès
    SET_TEXT_ALIGNMENT(CENTER,BOTTOM);
    TEXT( (5250.0, 6750.0), 'P1');
fin AFFICHER_PORTE_NOTE
```

ANNEXE II-1 : textes source des ressources résidantes

Cette annexe contient les divers textes de code source pour les ressources résidantes, et un programme d'exemple en Turbo-PASCAL 87. En voici la liste :

- Le programme essai_GKS, qui est un exemple d'utilisation de la ressource résidante METADESIGN-GKS 3.0. Il permet de saisir une suite de segments à l'écran, et de la restituer par GKS. On utilise pour cela les primitives "*REQUEST-LOCATOR*" et "*POLYLINE*".
  On notera surtout l'insertion des fichiers de types à utiliser (GKSTBPA.TYP), ainsi que le fichier d'interface (GKSTBPA.INT), qui contiennent les déclarations nécessaires à l'utilisation de la ressoucre résidante GKS.
- Les fichiers de déclarations et d'interface nécessaire à l'utilisation du GKS résidant.
- La partie interface pour application, qui est à connecter (.com, ou .bin, ou .obj) avec l'application.
- La partie interface pour ressource résidante, qui est à connecter avec la ressource résidante (ici GKS). Ceci est fait par le créateur de la ressource résidante.
- Le programme principal d'installation -chargement de la ressource résidante.

Le programme d'essai de la ressource résidante GKS

```
program ESSAI_GKS(input,output);

label
       DEBUT;

type
       {$I gkstbpa.typ}

var
{----- Les type reels pour DC }
       UNITS_MAX_DX,
       UNITS_MAX_DY,
       GRID_MAX_DX,
       GRID_MAX_DY,

{----- Les types reels pour WC }
       BORD_GAUCHE,
       BORD_DROIT,
       BORD_BAS,
       BORD_HAUT           : real;

{----- Les types pour Inquire }
       ERROR_INDICATOR     : integer;
       UNITS_TYPE          : DEVICE_UNITS;

{----- Les types pour le Releveur }
       LOCATOR_STATUS              : ETAT_INPUT;
       BACK_TRANSFORMATION         : integer;
       INITIAL_LOCATOR_POINT       : POINT;

{----- Les divers tableaux utilises }
       TWO_POINTS    : array [1..2] of POINT;
       CADRE         : array [1..5] of POINT;
       POLYLINE      : POINTS_ARRAY ;

{----- Quelques types utiles }
       CAR           : char;
       I,
       NB_POINTS     : integer;

{- Insertion des procedures GKS externes }
{$I gkstbpa.int }

begin
{- Ouverture de la session GKS et activation des stations de travail }
   G_OPEN_GKS;
   G_OPEN_WORKSTATION(SCREEN,1,OUTIN);
   G_ACTIVATE_WORKSTATION(SCREEN);

{- Positionnment en mode alphanumerique pour avoir le texte }
   G_SWITCH_ECRAN(false);

{- Interrogation des coordonnees max de station }
   G_INQUIRE_MAXIMUM_DISPLAY_SURFACE_SIZE(
     SCREEN,
     ERROR_INDICATOR,
     UNITS_TYPE,
     UNITS_MAX_DX,
     UNITS_MAX_DY,
     GRID_MAX_DX,
     GRID_MAX_DY  );

{- Renvoi de ces coords a utilisateur }
   if (UNITS_TYPE = METRES)
    then writeln('Les unites machines sont des metres')
    else writeln('Ce terminal fonctionne sans unites particulieres');

{- Debut pour recommencer }
DEBUT :
   writeln('Pour les workstation_Viewport, les coords maxi sont');
   writeln('En X : ',GRID_MAX_DX:10);
   writeln('En Y : ',GRID_MAX_DY:10);


{- On efface l'espace DC physique entierement }
   G_SET_WORKSTATION_VIEWPORT(
     SCREEN,
     0,
     GRID_MAX_DX - 1,
     0,
     GRID_MAX_DY - 1  );
   G_CLEAR_WORKSTATION(SCREEN,0);


{- Questionne utilisateur pour Workstation viewports }
   writeln('affectation WORKSTATION VIEWPORT:');
   writeln('Valeur bord gauche [0.0..',GRID_MAX_DX - 1:10,']:');
   readln(BORD_GAUCHE);
   writeln('Valeur bord droit [',BORD_GAUCHE:10,'..',GRID_MAX_DX - 1: 10,']:');
   readln(BORD_DROIT);
```

```
    writeln('Valeur bord bas [0.0..',GRID_MAX_DY - 1:10,']:');
    readln(BORD_BAS);
    writeln('Valeur bord haut [',BORD_BAS: 10,'..',GRID_MAX_DY - 1 :10,']:');
    readln(BORD_HAUT);
    G_SET_WORKSTATION_VIEWPORT(
      SCREEN,
      BORD_GAUCHE,
      BORD_DROIT,
      BORD_BAS,
      BORD_HAUT  );

{- Preparation de la transformation de normalisation }
{- La transformation est d'office 1 }
{- La Window pour cette transformation est d'office 0,10,0,10 }
   G_SET_WINDOW(1, 0.0, 10.0, 0.0, 10.0);
   G_SELECT_NORMALIZATION_TRANSFORMATION(1);
   G_CLEAR_WORKSTATION(SCREEN,0);

{- BEGIN Trace Cadre }
{- Trace du cadre definissant la Window pour cette Workstation Viewport }
{-- affectation du tableau contenant les coins de Window }
    CADRE[1].X := 0.0;
    CADRE[1].Y := 0.0;

    CADRE[2].X := 10.0;
    CADRE[2].Y := 0.0;

    CADRE[3].X := 10.0;
    CADRE[3].Y := 10.0;

    CADRE[4].X := 0.0;
    CADRE[4].Y := 10.0;

    CADRE[5]   := CADRE[1];

{-- Couleur de trace du cadre = 1 }
    G_SET_POLYLINE_COLOUR_INDEX(1);

{-- Le cadre sera trace a l'epaisseur 1 }
    G_SET_LINEWIDTH_SCALE_FACTOR(2.0);

{-- Le type de ligne pour le cadre sera mixte 4 }
    G_SET_LINETYPE(4);

{-- On le trace mais l'utilisateur ne le voit pas }
{-- car fenetre alpha selectionnee }
    G_SWITCH_ECRAN(true);
    G_POLYLINE(5, CADRE);
    readln;
    G_SWITCH_ECRAN(false);

{- END Trace Cadre }


{- BEGIN Initialisation Locator }
{- Initialisation du Releveur de Coordonnees (Locator) }
{--- La station de travail est l'outin SCREEN
     Le Releveur (logique) est celui de securite (clavier) no 1
     La transformation d'initialisation est la 1
     Le point initial est le centre de la Window
     Le type d'echo est CROSSHAIR implementation no 1
     La Zone d'echo est la totalite de l'espace physique DC
     Locator Data Record ignore                                    ---}

{--- Calcul du point initial : NOTEZ que la window est [0..10]x[0..10]
     d'ou la simplicite exemplaire du calcul  ---}
     INITIAL_LOCATOR_POINT.X := 5.0;
     INITIAL_LOCATOR_POINT.Y := 5.0;

{--- Les parametres sont passes par valeur, donc pas besoin de variables
     particulieres ---}
     G_INITIALISE_LOCATOR(
       SCREEN,
       1,
       1,
       INITIAL_LOCATOR_POINT,
       1,
       0.0, GRID_MAX_DX - 1,
       0.0, GRID_MAX_DY - 1,
       1 );

{- END Initialisation Locator }

{- Messages de courtoisie a l'utilisateur }
{   writeln('Saisie de points par releveur'); }
   write('nombre de points a saisir [1..50]:');
   readln(NB_POINTS);
   write('Voulez vous une sortie clippee (o/n) ?:');
   readln(CAR);
   if (CAR = 'O') or
```

```
      (CAR = 'o')
      then G_SET_CLIPPING_INDICATOR(true)
      else G_SET_CLIPPING_INDICATOR(false);

   writeln('Saisie des',NB_POINTS : 3,'demandes, frappez <RETURN>');
   readln;
{- passage en page graphique }
   G_SWITCH_ECRAN(true);


{- BEGIN Boucle de saisie }
{- Saisie des des points definissant le polyline }
{--- Initialisations }
     TWO_POINTS[1] := INITIAL_LOCATOR_POINT;
     TWO_POINTS[2] := INITIAL_LOCATOR_POINT;

     for I := 1 to NB_POINTS
      do begin
{----- Saisie d'un point }
       G_REQUEST_LOCATOR(
         SCREEN,
         1,
         LOCATOR_STATUS,
         BACK_TRANSFORMATION,
         TWO_POINTS[2]  );
{----- trace du segment obtenu }
       if (I > 1)
        then G_POLYLINE(2,TWO_POINTS);
{----- preparation pour iteration suivante }
       TWO_POINTS[1] := TWO_POINTS[2];
{----- affectation au polyline final }
       POLYLINE[I] := TWO_POINTS[2];
      end; {DO}

{- END Boucle de Saisie }


{- retrace final du polyline }
   G_SWITCH_ECRAN(false);
   writeln('TRACAGE final du polyline saisi, tapez <RETURN>');
   readln;
   G_SWITCH_ECRAN(true);
   G_POLYLINE(NB_POINTS,POLYLINE);
   G_SWITCH_ECRAN(false);

{- Reessai de l'utilisateur }
   writeln('Voulez_vous recommencer (o/n) ? : ');
   readln(CAR);
   if (CAR = 'O') or
      (CAR = 'o')
    then goto DEBUT
    else begin
{--- Fermeture de session GKS }
     G_DEACTIVATE_WORKSTATION(SCREEN);
     G_CLOSE_WORKSTATION(SCREEN);
     G_CLOSE_GKS;
    end;

{- FIN }

end.
```

Les fichiers de déclarations à inclure dans le texte source de l'application

```
{ ------------------------------------------------------------}
{ types a utiliser dans les drivers et les programmes application }
{ ------------------------------------------------------------}

        WKS_TYPE                = (SCREEN,PLOTTER);
        WKS_CATEGORY            = (INPUT,OUTIN,OUTPUT);
        DEVICE_UNITS            = (METRES,OTHER);
        ETAT_INPUT              = (OK,NONE);
        POINT                   = record
                                    X,Y : real;
                                  end;
        POINTS_ARRAY            = array [1..50] of point;
```

```
(* ------------------------------------------------------------ *)
(* Les declarations externes de procedures residantes disponibles *)
(* a utiliser dans la partie application ecrite en TURBO-PASCAL 87 *)
(* ------------------------------------------------------------ *)

procedure G_OPEN_GKS;
        external 'GKSTBPA.BIN';
procedure G_CLOSE_GKS;
        external G_OPEN_GKS[9];
procedure G_OPEN_WORKSTATION(
                    WKID    : WKS_TYPE;
                    CONID   : integer;
                    WTYPE   : WKS_CATEGORY);
        external G_OPEN_GKS[18];
procedure G_CLOSE_WORKSTATION( WKID : WKS_TYPE);
        external G_OPEN_GKS[29];
procedure G_ACTIVATE_WORKSTATION( WKID : WKS_TYPE);
        external G_OPEN_GKS[40];
procedure G_DEACTIVATE_WORKSTATION( WKID: WKS_TYPE);
        external G_OPEN_GKS[51];
procedure G_CLEAR_WORKSTATION(
                    WKID : WKS_TYPE;
                    COFL : integer);
        external G_OPEN_GKS[62];
procedure G_SWITCH_ECRAN(B : boolean);
        external G_OPEN_GKS[73];
procedure G_INQUIRE_MAXIMUM_DISPLAY_SURFACE_SIZE(
            WKID          : WKS_TYPE;
        var ERRIND        : integer;
        var DCUNIT        : DEVICE_UNITS;
        var RX,RY,LX,LY   : real);
        external G_OPEN_GKS[84];
procedure G_SET_MOVING_STEP( FPAS: integer);
        external G_OPEN_GKS[95];
procedure G_SET_AUTOVALIDATION( MFEU: boolean);
        external G_OPEN_GKS[106];
procedure G_GET_CODEREP(var CODR: integer);
        external G_OPEN_GKS[117];
procedure G_INITIALISE_LOCATOR(
                    WKID                : WKS_TYPE;
                    LCDNR,TNR           : integer;
                var INITIAL_POINT       : POINT;
                    PET                 : integer;
                    XMIN,XMAX,YMIN,YMAX : real;
                    LDR                 : integer);
        external G_OPEN_GKS[128];
procedure G_REQUEST_LOCATOR(
                    WKID       : WKS_TYPE;
                    LCDNR      : integer;
           var  STATUS     : ETAT_INPUT;
           var  TNR        : integer;
           var  NEW_POINT  : POINT);
        external G_OPEN_GKS[139];
procedure G_SET_CLIPPING_INDICATOR( CLSW : boolean);
        external G_OPEN_GKS[150];
procedure G_SET_WINDOW( TNR:integer;
            XMIN,XMAX,YMIN,YMAX : real);
        external G_OPEN_GKS[161];
procedure G_SET_WORKSTATION_VIEWPORT( WKID : WKS_TYPE;
                        XMIN,XMAX,YMIN,YMAX : real);
        external G_OPEN_GKS[172];
procedure G_SELECT_NORMALIZATION_TRANSFORMATION( TNR: integer);
        external G_OPEN_GKS[183];
procedure G_SET_LINETYPE( LTYPE: integer);
        external G_OPEN_GKS[194];
procedure G_SET_POLYLINE_COLOUR_INDEX( COLI : integer);
        external G_OPEN_GKS[205];
procedure G_SET_LINEWIDTH_SCALE_FACTOR( LWIDTH : real);
        external G_OPEN_GKS[216];
procedure G_POLYLINE(
             N: integer;
          var TAB_POINTS     );
        external G_OPEN_GKS[227];
procedure G_POLYMARKER(
              N                 : integer;
           var LISTE_MARKER  );
        external G_OPEN_GKS[238];
procedure G_SET_FILL_AREA_COLOUR_INDEX( COFI : integer);
        external G_OPEN_GKS[249];
procedure G_FILL_AREA(
              N :integer;
           var TAB_POINTS     );
        external G_OPEN_GKS[260];
```

L'interface pour l'application

```
        NAME    Interface_programme_utilisateur

DATA    SEGMENT PUBLIC 'DATA'
INT_BR  EQU 60H
DATA    ENDS

DGROUP  GROUP   DATA
ASSUME  DS:DGROUP,SS:DGROUP

IP_TEXT SEGMENT 'CODE'
ASSUME  CS:IP_TEXT

DFP     MACRO FPName,SizePar,FPNum
PUBLIC  FPname
FPname  PROC    FAR
        MOV     BX,FPNum        ;; BX=numero de la FP BIB a appeler
        MOV     CX,SizePar      ;; CX=taille des parametres empiles en octets
        INT     INT_BR          ;; appel de la BIB residente
        RET     SizePar
FPname  ENDP
        ENDM

INCLUDE FDFP

IP_TEXT ENDS
        END
```

L'interface pour la ressource résidante

```
        NAME    interface_bibliotheque_residente

INCLUDE FEXT

_DATA   SEGMENT WORD PUBLIC 'DATA'

EXTRN   SEG_PSP:WORD    ;segment du PSP

INT_BR  EQU 60H         ; vecteur d'interruption d'appel

BIB_SP  DW ?            ; variable BIB_SP devant contenir le SP de la BR
BIB_SS  DW ?            ; variable BIB_SS devant contenir le SS de la BR
BIB_SI  DW ?            ; variable BIB_SI devant contenir le SI de la BR
BIB_DI  DW ?            ; variable BIB_DI devant contenir le DI de la BR
OLD_SP  DW ?            ; variable devant contenir SP programme appelant
OLD_SS  DW ?            ; variable devant contenir SS programme appelant

_DATA    ENDS

DGROUP  GROUP   _DATA
ASSUME  DS:DGROUP,SS:DGROUP


IB_TEXT SEGMENT PARA PUBLIC 'CODE'
ASSUME  CS:IB_TEXT

TAB_FP  DD GETPSP               ;tableau des adresses des func/proc de la BR
INCLUDE FDD                     ;fichier des noms de func/proc cree par le
                                ;prog DFPEXTDD

BIB_DS  DW ?                    ; variable BIB_DS contenue dans SEGMENT CS

PUBLIC  INSTALLATION
INSTALLATION    PROC    FAR
;proc d'installation de la partie residente de la bibliotheque; elle
;recoit le nb de para de 16 octets a garder resident par le prog principal
        PUSH    BP
        MOV     BP,SP

;Sauvegarde des registres generaux utilises par la BIB residente
        MOV     BIB_DS,DS
        MOV     BIB_SP,SP
        MOV     BIB_SS,SS
        MOV     BIB_SI,SI
        MOV     BIB_DI,DI

;Positionnement du vecteur d'interruption INT_BR sur la proc CALL_FP
        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET CALL_FP
        MOV     AL,INT_BR
        MOV     AH,25H
        INT     21H

;Retour au systeme, BIB resident
        MOV     DX,6[BP]        ; nb de para a garder en memoire
        MOV     AL,INT_BR
        MOV     AH,31H
        INT     21H

        POP     BP
        RET     2
INSTALLATION    ENDP


GETPSP  PROC    FAR
;Retourne le segment du PSP dans AX, utilise par le programme SBR
        MOV     AX,SEG_PSP
        RET
GETPSP  ENDP


CALL_FP PROC    FAR
;proc appelant les func/proc de la bibliotheque residente

;Sauvegarde de DI,SI,DS programme appelant
        PUSH    DI
        PUSH    SI
        PUSH    DS

        MOV     DS,BIB_DS       ;restitution DS bib residente

        MOV     OLD_SP,SP       ;sauvegarde de SP prog appelant
        MOV     OLD_SS,SS       ;sauvegarde de SS prog appelant

        MOV     SP,BIB_SP       ;restitution SP bib residente
        MOV     SS,BIB_SS       ;restitution SS bib residente

;Passage des parametres aux func/proc appelees
;L'operation s'effectue en dupliquant la zone de parametres situee dans
```

```
;la pile du programme appelant, dans celle de la bibliotheque residente
        CMP     CX,0            ;taille des parametres=0 ?
        JZ      ENDREP          ;si oui ne pas dupliquer le pile

        MOV     ES,OLD_SS
        MOV     SI,OLD_SP
        ADD     SI,14           ;ajout du nb d'octets empiles depuis l'int
        ADD     SI,CX           ;ajout de la taille des parametres
        SHR     CX,1            ;taille des parametres en mots

REPEAT: PUSH    ES:[SI]
        SUB     SI,2            ;duplication des parametres
        LOOP    REPEAT
ENDREP:

;Appel de la func/proc codee
        SHL     BX,1
        SHL     BX,1
        CALL    TAB_FP[BX]

;Restitution du contexte programme appelant
        MOV     SS,OLD_SS       ;restitution SS
        MOV     SP,OLD_SP       ;restitution SP

;Restitution de DS,SI,DI programme appelant
        POP     DS
        POP     SI
        POP     DI

        IRET                    ;retour d'interruption INT_BR
CALL_FP ENDP


IB_TEXT ENDS                    ; fin du code segment CS IB_TEXT
        END
```

L'installateur -chargeur de la ressource résidante

```
program BID_INST(input,output);

!- ********************************************
!-  PROGRAMME PRINCIPAL pour RESSOURCES RESIDENTES
!-
!-
!- Fonction:
!-   Ce programme "installe" la ressource residente designee a l'edition des
!-  liens de ce programme.
!- Actions:
!-   Recherche, s'il existe, le parametre de programme TAILLE_TAS_DESIRE
!-  sur la ligne de commande; cette variable donnant la taille du tas
!-  disponible pour le programme resident. Ceci est donne par l'utilisateur
!-  de la ressource residente.
!-   Calcul du nombre correspondant de paragraphes de 16 octets a garder
!-  residents pour le programme
!-   Appel de la procedure d'installation de la Ressource residente
!- ********************************************


var     [extern]
!-      premier segment memoire alloue au programme
!-      Program Segment Prefix (cf doc msdos programmers reference)
        CESXQQ   : word;

var     [public]
!-      premier segment memoire alloue au programme
!-      Program Segment Prefix (cf doc msdos programmers reference)
!-      Cette variable est identique a CESXQQ mais est destine aux
!-      parties purement METADESIGN
        SEG_PSP : word;

var
!-      pointeur sur la ligne de commandes
        ADR_LIGNE_COMMANDE      : ads of lstring(127);
!-      pointeur --> variable contenant valeur 1er seg dispo
        ADS_FIRST_FREE_SEGMENT  : ads of word;

        TAILLE_TAS_DESIRE,
        WORD_BIDON,
        TAS_MAX,
        NB_PARAGRAPHES_ALLOUES  : word;

        CHAINE_TAS_LUE  : lstring(7);

function  FREECT(SIZE: word): word; extern;
procedure INSTALLATION(NB_PAR: word); extern;


begin
!- Determiner la place libre disponible a cette instant dans le tas du
!- programme ici present (cf MS PASCAL readme.doc )
  TAS_MAX := FREECT(0)*2;

!- Affectation de la variable de Metadesign contenant le Program
!- Segment Prefix. Pour CESXQQ, voir readme.doc Ms PASCAL
  SEG_PSP := CESXQQ;

!- ********************************************
!- Lire le parametre TAILLE_TAS_DESIRE du programme...
!- D'apres la doc MS_PAscal, on trouve le debut de la ligne de commande
!- a l'offset 80h du SEG_PSP.

!- segment(ADR_LIGNE_COMMANDE) = segment contenant ligne de commande
  ADR_LIGNE_COMMANDE.s := SEG_PSP;
!- offset(ADR_LIGNE_COMMANDE)  = offset debut chaine TAILLE_TAS_DESIRE
  ADR_LIGNE_COMMANDE.r := #80;

!-  par defaut tas de 1024 octets
  TAILLE_TAS_DESIRE := 1024;

  if ((ADR_LIGNE_COMMANDE^.len) <= 7)
   then begin
    !-  lecture de la ligne de commande
    CHAINE_TAS_LUE := ADR_LIGNE_COMMANDE^;

    !-  Decodage en un word de la chaine trouvee
    !-  et affectation a TAILLE_TAS_DESIRE
    if decode(CHAINE_TAS_LUE,WORD_BIDON)
     then TAILLE_TAS_DESIRE := WORD_BIDON;
   end;

!- Le tas demande ne peut depasser le tas maximum disponoble
  if TAILLE_TAS_DESIRE > TAS_MAX
   then TAILLE_TAS_DESIRE := TAS_MAX;
!- ********************************************

!- ********************************************
!- Recherche du nombre total de paragraphes memoire pris
!- par le programme resident en tenant compte du tas desire
```

```
  !- Recherche de la valeur du 1er segment disponible apres le programme
  !- BID_INST
  ADS_FIRST_FREE_SEGMENT.s := SEG_PSP;

  !- La doc MS_DOS dit qu'a l'adresse SEG_PSP:2 on trouve l'adresse
  !- du premier segment dispo apres BID_INST
  ADS_FIRST_FREE_SEGMENT.r      :=2;
  NB_PARAGRAPHES_ALLOUES := ADS_FIRST_FREE_SEGMENT^;

  !- Chercher nombre  de para alloues initiallement au prog
  NB_PARAGRAPHES_ALLOUES := NB_PARAGRAPHES_ALLOUES - SEG_PSP;

  !- Diminution eventuelle du nombre de paragraphes a allouer a BID_INST
  NB_PARAGRAPHES_ALLOUES := NB_PARAGRAPHES_ALLOUES -
                            (TAS_MAX - TAILLE_TAS_DESIRE) div 16;

  !- Installation de la ressource residente en mev
  !- NB_PARAGRAPHES_ALLOUES paragraphes de 16 octets seront residents
  INSTALLATION(NB_PARAGRAPHES_ALLOUES);
!- ************************************

end.
```

ANNEXE II-2 : textes source de GKS4.0 résidant

Cette annexe contient divers textes de code source pour le GKS résidant. Elle est décomposée en trois parties: Les fichiers d'installation du GKS résidant, quelques sous-programmes du binding ALSYS-ADA, quelques sous-programmes du binding MS-C

Les fichiers d'installation du GKS listés ici sont :

- la partie écrite en langage assembleur, qui installe le noyau GKS et le charge. Elle contient les routines d'installation (modification MS-DOS par modification de l'int 21h, retour aprés chargement en laissant GKS résidant en mémoire), la routine qui filtre les appels de manipulation de file handle (*FILTRE*), la routine de selection de primitive GKS appelée (*DISPATCHER*), les routines de contrôle et de test de l'appel (*TEST-HANDLE, TEST-NAME*), la routine de destruction du GKS (*DEL-DEV*).
- le fichier de définitions de macros, en langage assembleur MASM, utilisées dans le fichier précédent (*DECLARE.ASM*).
- le fichier de déclarations de primitves GKS disponibles (*ROUTINES.GKS*), à insérer dans la partie assembleur.

Remarquons que le fichier chargeur-installateur de GKS, écrit en MS-PASCAL, n'a pas été inclus ici, car il est de même nature que dans le cas des ressources résidantes.

Les sous-programmes du binding ALSYS-ADA que nous faisons figurer ici sont les plus représentatifs de cette interface :

- les procédures d'ouverture et de fermeture de GKS (*OPEN-GKS, CLOSE-GKS*).
- les procédures de traitement d'erreur (*ERROR-LOGGING*), et de fermeture d'urgence (*EMERGENCY-CLOSE-GKS*).
- trois procédures liées au tracé de segment : *SET-LINETYPE, INQ-POLYLINE-FACILITIES, et POLYLINE.*
- la procédure de relevé de coordonnées : *REQUEST-LOCATOR*.

Les sous-programmes de l'interface langage MS-C que nous avons sélectionnées ici sont les même que précédemment, sauf les procdédure de traitement d'erreur GKS, qui n'y figurent pas.

L'installateur-chargeur de GKS résidant

```
page    255,80

        name    GKS_DEV

include DECLARE.ASM

DR      macro   FP_NAME
extrn   FP_NAME: far
        endm
include ROUTINES.GKS


;-------------------------------------------------------------------------------
_DATA   segment word public 'data'
extrn   SEG_PSP: word                   ;- segment du PSP

GKS_SP  dw ?
GKS_SS  dw ?

PAR_OUT db 2054 dup (?)                 ;- buffer des params de sortie du GKS

INS_MES db 13,10
        db "Metadesign GKS Version 4.00",13,10
        db "(C) Copyright Metadesign S.A. 1988",13,10
        db "is installed",13,10
LEN_MES equ this byte -INS_MES

_DATA   ends

DGROUP  group   _DATA
assume  ds:DGROUP,ss:DGROUP


;-------------------------------------------------------------------------------
GKS_DEV segment para public 'code'
assume  cs:GKS_DEV

;- variables destinees a contenir des valeurs de registres alors que le
;- segment de donnees de ce programme n'est pas positionne
GKS_DS  dw ?
OLD_SP  dw ?
OLD_SS  dw ?

;- DOS memorise l' adresse du point d'entree originel MSDOS (vecteur d'int 21h)
DOS     dd ?

;- objets utilises pour l'appel de la fonction systeme choisie
TAB_RS  dw OPEN                         ;- 3D
        dw CLOSE                        ;- 3E
        dw READ                         ;- 3F
        dw WRITE                        ;- 40
REQ_RS  dw ?                            ;- routine systeme a appeler

;- declaration et initialisation du tableau contenant les adresses des proc
;- et func appelables dans le noyau GKS
purge   DR
DR      macro   FP_NAME
        dd      FP_NAME
        endm
TAB_PG  dd DEL_DEV                      ;- case 0 initialisee par DEL_DEV
include ROUTINES.GKS


;-------------------------------------------------------------------------------
ROUTINE DEL_DEV,far
BODY    macro

BEGIN
;- restitution du vecteur d'interruption initial
   push ds
   lds  dx,DOS
   mov  al,21h
   mov  ah,25h
   int  21h
   pop  ds

;- suppression du code residant
   mov  ax,SEG_PSP
   mov  es,ax
   mov  bx,es:2ch
   mov  ah,49h
   int  21h                             ;- suppression bloc programme
   mov  es,bx
   mov  ah,49h
   int  21h                             ;- suppression bloc environnement
endm
ENDR    DEL_DEV


;-------------------------------------------------------------------------------
```

```
ROUTINE DISPATCHER
BODY    macro
local   P_IN,VAL,ADS,P_OUT,CALL_PG

BEGIN
;- Sauvegarde des registres du programme appelant
   push ax
   push bx
   push cx
   push dx
   push di
   push si
   push ds
   push es
   mov  OLD_SP,sp
   mov  OLD_SS,ss

;- es_si := ds_dx
   mov  si,ds
   mov  es,si
   mov  si,dx

;- restitution des registres du device residant
   mov  ds,GKS_DS
   mov  sp,GKS_SP
   mov  ss,GKS_SS

;- empilage des parametres
   mov  bx,es:[si]                      ;- bx := NO_FONCTION demandee

   cmp  cx,4
   jb   P_OUT
   P_IN:                                ;- parametres d'entree
    ja   ADS
    VAL:                                ;- params in passes par valeur
     push es:2[si]
     jmp  short P_OUT
    ADS:                                ;- params in passes par adresse
     push es
     inc  si
     inc  si
     push si

   P_OUT:                               ;- parametres de sortie
    cmp  bx,0
    jg   CALL_PG
    mov  ax,offset DGROUP:PAR_OUT
    push ax
    neg  bx

;- Appel de la procedure designee
   CALL_PG:
    shl  bx,1
    shl  bx,1
    call TAB_PG[bx]

;- restitution du contexte programme appelant
   mov  ss,OLD_SS
   mov  sp,OLD_SP
   pop  es
   pop  ds
   pop  si
   pop  di
   pop  dx
   pop  cx
   pop  bx
   pop  ax

;- nombre d'octets traites
   mov  ax,cx
endm
ENDR    DISPATCHER


;-------------------------------------------------------------------------
ROUTINE OUT_PARAMETERS
BODY    macro

BEGIN
;- Sauvegarde des registres du programme appelant
   push ax
   push cx
   push di
   push si
   push ds
   push es

;- duplication du bloc de parametres de sortie dans le buffer d'adresse ds_dx
;- le bloc de parametres de sortie est celui genere par la derniere fonction
;- GKS appelee
```

```
    mov  ax,ds
    mov  es,ax
    mov  di,dx
    mov  ds,GKS_DS
    mov  si,offset DGROUP:PAR_OUT
    cld
    rep  movsb

 ;- restitution du contexte programme appelant
    pop  es
    pop  ds
    pop  si
    pop  di
    pop  cx
    pop  ax

 ;- nombre d'octets traites
    mov  ax,cx
 endm
 ENDR    OUT_PARAMETERS


 ;-------------------------------------------------------------------------------
 ROUTINE TEST_NAME
 BODY    macro
 local   IF_0,THEN_0,ENDIF_0

 BEGIN
 ;- le nom de device a l'adresse ds:dx est t-il le nom du device GKS
    xchg si,dx
    IF_0:
     cmp  word ptr   [si],'EM'
     jne  THEN_0
     cmp  word ptr  2[si],'AT'
     jne  THEN_0
     cmp  word ptr  4[si],'KG'
     jne  THEN_0
     cmp  word ptr  6[si],'4S'
     jne  THEN_0
     cmp  word ptr  8[si],'D.'
     jne  THEN_0
     cmp  word ptr 10[si],'VE'
     je   ENDIF_0
    THEN_0:
     xchg dx,si
     inc  sp
     inc  sp
     jmp  short JMP_DOS
    ENDIF_0:
    xchg dx,si
 endm
 ENDR    TEST_NAME


 ;-------------------------------------------------------------------------------
 ROUTINE TEST_HANDLE
 BODY    macro
 local   IF_0,ENDIF_0

 BEGIN
 ;- le numero de handle correspond t-il a un device residant
    IF_0:
     cmp  bx,255
     jae  ENDIF_0
    THEN
     inc  sp
     inc  sp
     jmp  short JMP_DOS
    ENDIF_0:
 endm
 ENDR    TEST_HANDLE


 ;-------------------------------------------------------------------------------
 ;- l'appel ne nous concerne pas, donc appel DOS
    JMP_DOS:
     jmp  DOS


 ;-------------------------------------------------------------------------------
 FILTRE  proc
 ;- point d'entree systeme
    cmp  ah,3Dh
    jb   JMP_DOS
    cmp  ah,40h
    ja   JMP_DOS

 ;- decodage fonction et appel
    push bx
    mov  bl,ah
```

```
    xor  bh,bh
    sub  bl,30h
    shl  bx,1
    mov  bx,TAB_RS[bx]
    mov  REQ_RS,bx
    pop  bx
    jmp  REQ_RS

    OPEN:
     call TEST_NAME
     mov  ax,255
     jmp  short RETURN

    CLOSE:
     call TEST_HANDLE
     jmp  short RETURN

    READ:
     call TEST_HANDLE
     call OUT_PARAMETERS
     jmp  short RETURN

    WRITE:
     call TEST_HANDLE
     call DISPATCHER

    RETURN:
     push bp
     mov  bp,sp
     and  word ptr 6[bp],0FFFEh
     pop  bp
     iret

FILTRE  endp

;-------------------------------------------------------------------------
ROUTINE INSTALLATION,far
BODY    macro
local   NB_PARAGRAPHES

        T_WORD  PARAGRAPHS_TO_KEEP
BEGIN
;- Sauvegarde des registres generaux utilises par la GKS_DEV residante
   mov  GKS_DS,ds
   mov  GKS_SP,sp
   mov  GKS_SS,ss

;- recuperation du vecteur d'interruption 21h et sauvegarde dans DOS
   mov  al,21h
   mov  ah,35h
   int  21h
   mov  word ptr DOS,bx
   mov  bx,es
   mov  word ptr DOS+2,bx

;- message copyright
   mov  ah,40h
   mov  bx,2
   mov  cx,LEN_MES
   mov  dx,offset DGROUP:INS_MES
   int  21h

;- positionnement du vecteur d'interruption 21h sur la procedure FILTRE
   mov  dx,cs
   mov  ds,dx
   mov  dx,offset FILTRE
   mov  al,21h
   mov  ah,25h
   int  21h

;- retour au systeme avec PARAGRAPHS_TO_KEEP residants
   mov  dx,PARAGRAPHS_TO_KEEP
   xor  al,al
   mov  ah,31h
   int  21h
endm
ENDR    INSTALLATION


GKS_DEV ends
        end
```

Les macros en assembleur, pour le fichier précédent

```
ROUTINE macro   R_NAME,DISTANCE
        TAILLE_PARAMETRES = 0
        TAILLE_VARIABLES  = 0
R_NAME  proc    DISTANCE
ifb     <DISTANCE>
        BASE_PARAMETRES   = 4
else
        BASE_PARAMETRES   = 6
public  R_NAME
endif
        TYPE_DECLARATION  = 0
        endm


VAR     macro
        TYPE_DECLARATION  = 1
        endm


DEFTYPE macro   VARIABLE,TAILLE,PTR_TYPE
local   OFFSET_PARAMETRE

if      TYPE_DECLARATION eq 0
        OFFSET_PARAMETRE  = BASE_PARAMETRES+TAILLE_PARAMETRES
        TAILLE_PARAMETRES = TAILLE_PARAMETRES+TAILLE
else
        TAILLE_VARIABLES  = TAILLE_VARIABLES+TAILLE
        OFFSET_PARAMETRE  = -TAILLE_VARIABLES
endif
ifb     <PTR_TYPE>
        VARIABLE equ [bp+OFFSET_PARAMETRE]
else
        VARIABLE equ &PTR_TYPE ptr [bp+OFFSET_PARAMETRE]
endif
        endm


BOOLEAN macro   VARIABLE
        DEFTYPE VARIABLE,2,byte
        endm

T_BYTE  macro   VARIABLE
        DEFTYPE VARIABLE,2,byte
        endm

CHAR    macro   VARIABLE
        DEFTYPE VARIABLE,2,byte
        endm

INTEGER macro   VARIABLE
        DEFTYPE VARIABLE,2,word
        endm

T_WORD  macro   VARIABLE
        DEFTYPE VARIABLE,2,word
        endm

LONGINT macro   VARIABLE
        DEFTYPE VARIABLE,4,dword
        endm

REAL4   macro   VARIABLE
        DEFTYPE VARIABLE,4,dword
        endm

REAL8   macro   VARIABLE
        DEFTYPE VARIABLE,8,qword
        endm

VARS    macro   VARIABLE
        DEFTYPE VARIABLE,4
        endm

VARS_SA macro   VARIABLE
        DEFTYPE VARIABLE,6
        endm

ADSMEM  macro   VARIABLE
        DEFTYPE VARIABLE,4
        endm

ADDRESS macro   VARIABLE
        DEFTYPE VARIABLE,4
        endm


BEGIN   macro
if      (TAILLE_PARAMETRES+TAILLE_VARIABLES) ne 0
  push bp
```

```
  mov  bp,sp
endif
if      TAILLE_VARIABLES ne 0
  sub  sp,TAILLE_VARIABLES
endif
        endm

ENDR    macro   R_NAME
  BODY
if      TAILLE_VARIABLES ne 0
  mov  sp,bp
endif
if      (TAILLE_PARAMETRES+TAILLE_VARIABLES) ne 0
  pop  bp
endif
  ret  TAILLE_PARAMETRES
R_NAME  endp
        endm

DO      macro
        endm
THEN    macro
        endm
```

Les déclarations assembleur des routines GKS disponibles

```
DR      OPEN_GKS                                =   1;
DR      CLOSE_GKS                               =   2;
DR      OPEN_WS                                 =   3;
DR      CLOSE_WS                                =   4;
DR      ACTIVATE_WS                             =   5;
DR      DEACTIVATE_WS                           =   6;
DR      CLEAR_WS                                =   7;
DR      UPDATE_WS                               =   8;
DR      POLYLINE                                =   9;
DR      POLYMARKER                              =  10;
DR      TEXTE                                   =  11;
DR      FILL_AREA                               =  12;
DR      CELL_ARRAY                              =  13;
DR      SET_POLYLINE_INDEX                      =  14;
DR      SET_LINETYPE                            =  15;
DR      SET_LINEWIDTH_SCALE_FACTOR              =  16;
DR      SET_POLYLINE_COLOUR_INDEX               =  17;
DR      SET_POLYMARKER_INDEX                    =  18;
DR      SET_MARKER_TYPE                         =  19;
DR      SET_MARKER_SIZE_SCALE_FACTOR            =  20;
DR      SET_POLYMARKER_COLOUR_INDEX             =  21;
DR      SET_TEXT_INDEX                          =  22;
DR      SET_TEXT_FONT_AND_PRECISION             =  23;
DR      SET_CHAR_EXPANSION_FACTOR               =  24;
DR      SET_CHAR_SPACING                        =  25;
DR      SET_TEXT_COLOUR_INDEX                   =  26;
DR      SET_CHAR_HEIGHT                         =  27;
DR      SET_CHAR_UP_VECTOR                      =  28;
DR      SET_TEXT_PATH                           =  29;
DR      SET_TEXT_ALIGNMENT                      =  30;
DR      SET_FILL_AREA_INDEX                     =  31;
DR      SET_FILL_AREA_INTERIOR_STYLE            =  32;
DR      SET_FILL_AREA_STYLE_INDEX               =  33;
DR      SET_FILL_AREA_COLOUR_INDEX              =  34;
DR      SET_PATTERN_SIZE                        =  35;
DR      SET_PATTERN_REFERENCE_POINT             =  36;
DR      SET_ASF                                 =  37;
DR      SET_COLOUR_REP                          =  38;
DR      SET_WINDOW                              =  39;
DR      SET_VIEWPORT                            =  40;
DR      SET_VIEWPORT_INPUT_PRIORITY             =  41;
DR      SELECT_NORMALIZATION_TRANSF             =  42;
DR      SET_CLIPPING_INDICATOR                  =  43;
DR      SET_WS_WINDOW                           =  44;
DR      SET_WS_VIEWPORT                         =  45;
DR      INITIALISE_LOCATOR                      =  46;
DR      INITIALISE_STROKE                       =  47;
DR      INITIALISE_VALUATOR                     =  48;
DR      INITIALISE_CHOICE                       =  49;
DR      INITIALISE_STRING                       =  50;
DR      SET_LOCATOR_MODE                        =  51;
DR      SET_STROKE_MODE                         =  52;
DR      SET_VALUATOR_MODE                       =  53;
DR      SET_CHOICE_MODE                         =  54;
DR      SET_STRING_MODE                         =  55;
DR      REQUEST_LOCATOR                         =  56;
DR      REQUEST_STROKE                          =  57;
DR      REQUEST_VALUATOR                        =  58;
DR      REQUEST_CHOICE                          =  59;
DR      REQUEST_STRING                          =  60;
DR      WRITE_ITEM_TO_GKSM                      =  61;
DR      GET_ITEM_TYPE_FROM_GKSM                 =  62;
DR      READ_ITEM_FROM_GKSM                     =  63;
DR      INTERPRET_ITEM                          =  64;
DR      INQ_OPERATING_STATE_VALUE               =  65;
DR      INQ_LEVEL_OF_GKS                        =  66;
DR      INQ_LIST_OF_AVAILABLE_WS_TYPES          =  67;
DR      INQ_MAX_NORMALIZATION_TRANSF_NB         =  68;
DR      INQ_SET_OF_OPEN_WS                      =  69;
DR      INQ_CURRENT_PRIMITIVE_ATR_VALUES        =  70;
DR      INQ_CURRENT_INDIVIDUAL_ATR_VALUES       =  71;
DR      INQ_CURRENT_NORMALIZATION_TRANSF_NB     =  72;
DR      INQ_LIST_OF_NORMALIZATION_TRANSF_NB     =  73;
DR      INQ_NORMALIZATION_TRANSF                =  74;
DR      INQ_CLIPPING                            =  75;
DR      INQ_WS_CONNECTION_AND_TYPE              =  76;
DR      INQ_WS_STATE                            =  77;
DR      INQ_WS_DEFERRAL_AND_UPDATE_STATES       =  78;
DR      INQ_TEXT_EXTENT                         =  79;
DR      INQ_LIST_OF_COLOUR_INDICES              =  80;
DR      INQ_COLOUR_REP                          =  81;
DR      INQ_WS_TRANSF                           =  82;
DR      INQ_LOCATOR_DEVICE_STATE                =  83;
DR      INQ_STROKE_DEVICE_STATE                 =  84;
DR      INQ_VALUATOR_DEVICE_STATE               =  85;
DR      INQ_CHOICE_DEVICE_STATE                 =  86;
DR      INQ_STRING          STATE               =  87;
DR      INQ_WS_
                                                =  88;
        INQ_WS_CLASSIFICATION                   =  89;
DR      INQ_MAX_DISPLAY_SURFACE_SIZE            =  90;
```

```
DR      INQ_POLYLINE_FACILITIES                          = 91;
DR      INQ_PREDEFINED_POLYLINE_REP                      = 92;
DR      INQ_POLYMARKER_FACILITIES                        = 93;
DR      INQ_PREDEFINED_POLYMARKER_REP                    = 94;
DR      INQ_TEXT_FACILITIES                              = 95;
DR      INQ_PREDEFINED_TEXT_REP                          = 96;
DR      INQ_FILL_AREA_FACILITIES                         = 97;
DR      INQ_PREDEFINED_FILL_AREA_REP                     = 98;
DR      INQ_PATTERN_FACILITIES                           = 99;
DR      INQ_PREDEFINED_PATTERN_REP                       = 100;
DR      INQ_COLOUR_FACILITIES                            = 101;
DR      INQ_PREDEFINED_COLOUR_REP                        = 102;
DR      INQ_LIST_OF_AVAILABLE_GDP                        = 103;
DR      INQ_GDP                                          = 104;
DR      INQ_MAX_LENGTH_OF_WS_STATE_TABLES                = 105;
DR      INQ_NB_OF_AVAILABLE_LOGICAL_INPUT_DEVICES        = 106;
DR      INQ_DEFAULT_LOCATOR_DEVICE_DATA                  = 107;
DR      INQ_DEFAULT_STROKE_DEVICE_DATA                   = 108;
DR      INQ_DEFAULT_VALUATOR_DEVICE_DATA                 = 109;
DR      INQ_DEFAULT_CHOICE_DEVICE_DATA                   = 110;
DR      INQ_DEFAULT_STRING_DEVICE_DATA                   = 111;
DR      INQ_PIXEL_ARRAY_DIMENSIONS                       = 112;
DR      INQ_PIXEL_ARRAY                                  = 113;
DR      INQ_PIXEL                                        = 114;
DR      EMERGENCY_CLOSE_GKS                              = 115;
DR      CIRCLE                                           = 116;
DR      CIRCULAR_ARC                                     = 117;
DR      ELLIPSE                                          = 118;
DR      ELLIPTICAL_ARC                                   = 119;
DR      RECTANGLE                                        = 120;
DR      SET_WS_CONFIGURATION                             = 121;
DR      INQ_WS_CONFIGURATION                             = 122;
DR      INQ_DEFERRAL_ABILITY                             = 123;
DR      INQ_WS_NAME_AND_DISPLAY_TYPE                     = 124;
DR      SET_CHAR_BASE_VECTOR                             = 125;
DR      SET_WS_VIEWPORT_ADJUSTMENT                       = 126;
DR      SET_GRAPHIC_DISPLAY_MODE                         = 127;
DR      SET_XOR_DISPLAY_MODE                             = 128;
DR      SET_GRAPHIC_OUTPUT_SPEED_MODE                    = 129;
DR      INQ_DISPLAY_MODES                                = 130;
DR      SET_PEN_CAPACITIES                               = 131;
DR      SET_PEN_WIDTH                                    = 132;
DR      SET_ASPECT_SCALING                               = 133;
DR      INQ_PEN_CAPACITIES                               = 134;
DR      INQ_PEN_WIDTH                                    = 135;
DR      INQ_ASPECT_SCALING                               = 136;
DR      SET_LOCATOR_DEFINITION                           = 137;
DR      INQ_VALIDATION_CODE                              = 138;
DR      SET_STROKE_DEFINITION                            = 139;
```

Ouverture et fermeture GKS en ADA

```
with FILE,GKS_TYPES,GKS_NAMES,GKS_RUN,ERROR_HANDLING;
use  FILE,GKS_TYPES,GKS_NAMES,GKS_RUN;

package body GKS is

-------------------------------------------------------------------------------
-- fonctions OPEN et CLOSE GKS

procedure OPEN_GKS(
        ERROR_FILE                : in string  := DEFAULT_ERROR_FILE;
        AMOUNT_OF_MEMORY          : in natural := DEFAULT_MEMORY_UNITS) is

type        T_P_I is
        record
         FONCTION                 : integer;
         AMOUNT_OF_MEMORY         : integer;
         TEST_ERROR_MODE          : boolean;
        end record;

        P_I                         : T_P_I;
        LEN                         : integer;
begin
-- ouverture du device GKS
   HANDLE_GKS := F_OPEN ("METAGKS4.DEV");
   if F_GKS_NOT_INSTALLED then
    return;
   end if;

-- sauvegarde du nom du fichier d'erreur
   LEN := ERROR_FILE'length;
   if LEN > 64 then LEN := 64; end if;
   ERR_FILE (1..LEN) := ERROR_FILE (1..LEN);

-- faut t'il tester les erreurs ?
   VR_TEST_ERROR := (ERROR_FILE (1..3) /= "NUL");

   P_I.AMOUNT_OF_MEMORY := AMOUNT_OF_MEMORY;
   P_I.TEST_ERROR_MODE  := VR_TEST_ERROR;

   P_I.FONCTION := - C_OPEN_GKS;
   P_WRITE_BUFFER (HANDLE_GKS, P_I'size /8, P_I'address);

-- ouverture du fichier "Noms des fonctions Gks"
   HANDLE_NAMES := F_OPEN ("\GKS\DIVERS\GKS.NOM");

-- creation du fichier d'erreurs si necessaire
   if not VR_TEST_ERROR then return; end if;

      if (ERR_FILE (1..3) = "CON") then HANDLE_ERROR := 1;
   elsif (ERR_FILE (1..3) = "AUX") then HANDLE_ERROR := 3;
   elsif (ERR_FILE (1..3) = "PRN") then HANDLE_ERROR := 4;
   else
    HANDLE_ERROR := F_CREATE (ERR_FILE);
    if HANDLE_ERROR = -1 then
     HANDLE_ERROR := 2;
     P_PROCESS_ERROR (200, C_OPEN_GKS);
    end if;
   end if;

   P_TEST_ERROR (C_OPEN_GKS);
end OPEN_GKS;


procedure CLOSE_GKS is

        FONCTION: integer;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   FONCTION := - C_CLOSE_GKS;
   P_WRITE_BUFFER (HANDLE_GKS, 2, FONCTION'address);

   P_TEST_ERROR (C_CLOSE_GKS);

   if HANDLE_ERROR > 4 then P_CLOSE (HANDLE_ERROR); end if;
   if HANDLE_NAMES > 4 then P_CLOSE (HANDLE_NAMES); end if;
   P_CLOSE (HANDLE_GKS);

   HANDLE_ERROR :=  2;
   HANDLE_NAMES := -1;
   HANDLE_GKS := -1;
end CLOSE_GKS;
```

le traitement d'erreurs GKS en ADA

```
separate (GKS)
procedure EMERGENCY_CLOSE_GKS is

        FONCTION        : integer;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   FONCTION := C_EMERGENCY_CLOSE_GKS;
   P_WRITE_BUFFER (HANDLE_GKS, 2, FONCTION'address);
end EMERGENCY_CLOSE_GKS;

--------------------------------------------------------------------------------
separate (GKS)
procedure ERROR_LOGGING(
        ERROR_INDICATOR         : in ERROR_NUMBER;
        NAME                    : in string;
        ERROR_FILE              : in string := DEFAULT_ERROR_FILE) is

        M_ERROR_NUMBER          : constant string := "Error number ";
        M_IN                    : constant string := " in ";
begin
   P_WRITE_STRING (HANDLE_ERROR, M_ERROR_NUMBER);
   P_WRITE_STRING (HANDLE_ERROR, ERROR_NUMBER'image (ERROR_INDICATOR) );
   P_WRITE_STRING (HANDLE_ERROR, M_IN);
   P_WRITE_STRING (HANDLE_ERROR, NAME);
   P_WRITE_CRLF (HANDLE_ERROR);
end ERROR_LOGGING;
```

procédures pour le tracé de segment en ADA

```
separate (GKS)
procedure SET_LINETYPE(
        TYPE_OF_LINE        : in LINETYPE) is

type        T_P_I is
        record
         FONCTION        : integer;
         LT              : integer;
        end record;

        P_I               : T_P_I;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   P_I.LT := integer (TYPE_OF_LINE);

   P_I.FONCTION := - C_SET_LINETYPE;
   P_WRITE_BUFFER (HANDLE_GKS, P_I'size /8,P_I'address);

   P_TEST_ERROR (C_SET_LINETYPE);
end SET_LINETYPE;
```

```
separate (GKS)
procedure INQ_POLYLINE_FACILITIES(
        TYPE_OF_WS              : in WS_TYPE;

        ERROR_INDICATOR             : out ERROR_NUMBER;
        LIST_OF_TYPES               : out LINETYPES.LIST_OF;
        NUMBER_OF_WIDTHS        : out natural;
        NOMINAL_WIDTH               : out DC.MAGNITUDE;
        RANGE_OF_WIDTHS             : out DC.RANGE_OF_MAGNITUDES;
        NUMBER_OF_INDICES       : out natural) is

type        T_P_I is
        record
         FONCTION               : integer;
         WS_TYPE                : integer;
        end record;
type        T_P_O is
        record
         ERREUR                     : integer;
         NB_EPAISSEURS              : integer;
         EPAISSEUR_NOMINALE,
         EPAISSEUR_MIN,
         EPAISSEUR_MAX              : T_REAL;
         NB_INDEX_PREDEFINIS        : integer;
         LINE_TYPES_N               : integer;
         LINE_TYPES_L               : T_INTEGER_LIST (1..32);
        end record;

        P_I                     : T_P_I;
        P_O                     : T_P_O;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   P_I.WS_TYPE := integer (TYPE_OF_WS) -1;

   P_I.FONCTION := - C_INQ_POLYLINE_FACILITIES;
   P_WRITE_BUFFER (HANDLE_GKS, P_I'size /8, P_I'address);
   P_READ_BUFFER (HANDLE_GKS, P_O'size /8, P_O'address);

   ERROR_INDICATOR := ERROR_NUMBER'val (P_O.ERREUR);
   if P_O.ERREUR /= 0 then return; end if;

   NUMBER_OF_WIDTHS        := natural (P_O.NB_EPAISSEURS);
   NOMINAL_WIDTH       := DC.MAGNITUDE (P_O.EPAISSEUR_NOMINALE);
   RANGE_OF_WIDTHS.MIN         := DC.MAGNITUDE (P_O.EPAISSEUR_MIN);
   RANGE_OF_WIDTHS.MAX         := DC.MAGNITUDE (P_O.EPAISSEUR_MAX);
   NUMBER_OF_INDICES       := natural (P_O.NB_INDEX_PREDEFINIS);
   declare
        LINE_TYPES_L        : LINETYPES.LIST_VALUES (1..P_O.LINE_TYPES_N);
   begin
    for I in LINE_TYPES_L'range
    loop
     LINE_TYPES_L (I) := LINETYPE (P_O.LINE_TYPES_L(I));
    end loop;
    LIST_OF_TYPES := LINETYPES.LIST (LINE_TYPES_L);
   end;
end INQ_POLYLINE_FACILITIES;
```

```
separate (GKS)
procedure POLYLINE(
        POINTS              : in WC.POINT_ARRAY) is

type        T_P_I is
        record
         FONCTION        : integer;
         POINTS_N        : integer;
         POINTS_L        : WC.POINT_ARRAY (1..256);
        end record;

        P_I               : T_P_I;
        POINTS_N        : integer;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   POINTS_N := POINTS'length;
   if POINTS_N > 256 then
    POINTS_N := 256;
   end if;
   P_I.POINTS_N := POINTS_N;
   if POINTS_N > 0 then
    P_I.POINTS_L (1..P_I.POINTS_N) := POINTS (1..P_I.POINTS_N);
   end if;

   P_I.FONCTION := - C_POLYLINE;
   P_WRITE_BUFFER (HANDLE_GKS, 2*2 + POINTS_N*8, P_I'address);

   P_TEST_ERROR (C_POLYLINE);
end POLYLINE;
```

la procédure de relevé de coordonnées

```
separate (GKS)
procedure REQUEST_LOCATOR(
        WS                  : in WS_ID;
        DEVICE                  : in DEVICE_NUMBER;

        STATUS                  : out INPUT_STATUS;
        TRANSFORMATION          : out TRANSFORMATION_NUMBER;
        POSITION        : out WC.POINT) is

type        T_P_I is
        record
         FONCTION         : integer;
         WS                 : integer;
         DEVICE                 : integer;
        end record;
type        T_P_O is
        record
         ERREUR                 : integer;
         STATUS                 : integer;
         TRANSFORMATION         : integer;
         POSITION         : WC.POINT;
        end record;

        P_I                 : T_P_I;
        P_O                 : T_P_O;
begin
   if F_GKS_NOT_INSTALLED then return; end if;

   P_I.WS := integer (WS);
   P_I.DEVICE := integer (DEVICE);

   P_I.FONCTION := - C_REQUEST_LOCATOR;
   P_WRITE_BUFFER (HANDLE_GKS, P_I'size /8, P_I'address);
   P_READ_BUFFER (HANDLE_GKS, P_O'size /8, P_O'address);

   if P_O.ERREUR /= 0 then
    P_PROCESS_ERROR (P_O.ERREUR, C_REQUEST_LOCATOR);
    return;
   end if;

   STATUS := INPUT_STATUS'val (P_O.STATUS);
   TRANSFORMATION := TRANSFORMATION_NUMBER (P_O.TRANSFORMATION);
   POSITION := P_O.POSITION;

end REQUEST_LOCATOR;
```

Ouverture et fermeture GKS en C

```
/* inclusion des .h */
#include <stdio.h>
#include <io.h>
#include <gks.h>
#include <utigks.h>
#include <nomgks.h>

/* Declaration des fonctions externes */
extern        int        Bc_F_Gks_Installe(void);        /* bcudev.c */
extern        int     Bc_F_Verifier_Erreur (Gint);        /* bcertr.c */

/* Declaration des variables externes */
extern        int        Bc_V_Dev_Gks_Id;        /* numero de handle */

/* ------------------------------------------------------------------- */
/* Open GKS */
Gint        gopengks (errfile, memory)
FILE        *errfile;        /* error file pointer */
Glong        memory;                /* bytes of memory available for buffer space */

{ /* begin gopengks */

 /* Declaration de variables de passage de parametre */
 struct Type_Param_In {
        Gint                No_Fonction;
        Gint                Nb_Buffer;
        unsigned        Test_Error_Mode;
 } Param_In;

 /* debut de la fonction */
 /* printf(" proc gopengks \n ");
  */

 /* ouverture du handle */
 if (Bc_F_Open_Device_Gks() == 0) return(-1);          /* GKS n'est pas installe */

 /* Gks est installe */
  /* On passer au module de gestion d'erreur le pointeur sur le fichier */
  Bc_PW_Ptr_File_Error (errfile);

 /* On remplit la structure */
 Param_In.No_Fonction = -GKS_C_OPEN_GKS;
 Param_In.Nb_Buffer   = (Gint) memory;        /* conversion long --> int */
 if (errfile == NULL)
    Param_In.Test_Error_Mode = 0;
 else
    Param_In.Test_Error_Mode = 1;

 /* On ecrit sur le fichier GKS resident */
 write ( Bc_V_Dev_Gks_Id, &Param_In, 5);

 /* test des erreurs */
 return ( Bc_F_Verifier_Erreur (GKS_C_OPEN_GKS));

} /* end gopengks */

/* ********************************************************************** */

/* Close Gks */
Gint        gclosegks()
       /* Cette procedure ferme GKS */

{ /* begin gclosegks */

 /* Declaration de variables de passage de parametre */
 struct Type_Param_In {
        Gint        No_Fonction;
 } Param_In;

 /* debut de la fonction */
 /* printf("proc gclosegks \n ");
  */

 /* gks est il la ? */
 if (Bc_F_Gks_Installe() == 0) return(-1);          /* GKS n'est pas installe */

 /* Gks est installe */
 /* On met les infos dans la structure */
 Param_In.No_Fonction = -GKS_C_CLOSE_GKS;

 /* On ecrit sur le fichier GKS resident */
 write ( Bc_V_Dev_Gks_Id, &Param_In, 2 );

 /* test des erreurs */
 return ( Bc_F_Verifier_Erreur (GKS_C_CLOSE_GKS));

} /* end gclosegks */
```

procédure POLYLINE de tracé de segment

```
/* ------------------------------------------------------------ */
/* Polyline */
Gint        gpolyline (npoints, points)
Gint        npoints;        /* number of points */
Gpoint       *points;        /* points array */

{ /* begin gpolyline */

 /* Declaration de variables de passage de parametre */
 struct Type_Param_In {
        Gint         No_Fonction;
        Gint        Nb_Points;
        Gpoint        Tab_Points[NB_MAX_POINTS_DANS_LISTE];
 } Param_In;

 /* Declaration des variables de travail */
 Gint        I;

 /* ..................................... */
 /* debut de la fonction */
  /*  printf("proc gpolyline \n ");
   */

 /* gks est il la ? */
 if (Bc_F_Gks_Installe() == 0) return(-1);          /* GKS n'est pas installe */

 /* Gks est installe */
 /* On remplit la structure */
 Param_In.No_Fonction        = -GKS_C_POLYLINE;
 if (npoints > NB_MAX_POINTS_DANS_LISTE) npoints = NB_MAX_POINTS_DANS_LISTE;
 Param_In.Nb_Points = npoints;
 for (I=0 ; I < npoints ; I++)
 {
    Param_In.Tab_Points[I] = points[I];
 }

 /* On ecrit sur le fichier GKS resident */
 write ( Bc_V_Dev_Gks_Id, &Param_In, 4+(npoints*8) );

 /* test des erreurs */
 return ( Bc_F_Verifier_Erreur (GKS_C_POLYLINE));

} /* end gpolyline */

/* ********************************************************************** */
```

procédure de saisie de coordonées en C

```
/* Request Locator */
Gint        greqloc (ws, dev, reponse)
Gint        ws;                 /* workstation identifier */
Gint        dev;                 /* locator device number */
Gqloc        *reponse;          /* OUT locator reponse */

       /* Cette procedure recoit un point
        */

{ /* begin greqloc */

 /* Declaration de variables de passage de parametre */
 struct Type_Param_In {
        Gint          No_Fonction;
       Gint         Ws_Id;
       Gint         Lid;
 } Param_In;

 struct Type_Param_Out {
        Gint         No_Error;
        Gint         Status;
       Gint         Norm_Transf_Nb;
       Gpoint         Loc_Position;
 } Param_Out;

 unsigned int count = 14;

 /* debut de la fonction */
 /* printf("proc greqloc \n");
  */

 /* gks est il la ? */
 if (Bc_F_Gks_Installe() == 0) return(-1);          /* GKS n'est pas installe */

 /* Gks est installe */
 /* On met les infos dans la structure */
 Param_In.No_Fonction = -GKS_C_REQUEST_LOCATOR;
 Param_In.Ws_Id = ws;
 Param_In.Lid = dev;

 /* On ecrit sur le fichier GKS resident */
 write ( Bc_V_Dev_Gks_Id, &Param_In, 6 );

 /* On lit les infos */
 read ( Bc_V_Dev_Gks_Id, &Param_Out, count);

 if (Param_Out.No_Error == 0)
 {
   /* On affect les infos */
   reponse->status = (Gistat) Param_Out.Status;
   (reponse->loc).transform = Param_Out.Norm_Transf_Nb;
   (reponse->loc).position  = Param_Out.Loc_Position;
 }
 else Bc_P_Ecrire_Erreur (Param_Out.No_Error, GKS_C_REQUEST_LOCATOR);

 /* la fonction retourne l'erreur */
 return (Param_Out.No_Error);

} /* end greqloc */
```

ANNEXE II-3 : textes source de l'interface D-PROLOG - METADESIGN-GKS3.0

Cette annexe contient les textes de code source principaux pour l'interface entre D-PROLOG et METADESIGN-GKS3.0.

Les fichiers listés ici sont, dans l'ordre :

- La "banque" GKS, déclarant l'ensemble des primitives GKS externes utilisables sous forme de prédicats étrangers.
- Un petit programme d'exemple d'utilisation , qui trace un suite de segments contigus à l'écran.
- la gestion et le contrôle des données transmises à GKS, et inversement.
- le module contenant les définitions des prédicats étrangers. Ce fichier est à modifier et recompiler, quand on veut insérer un prédicat étranger.

GKS.LOG, **la "banque GKS" à déclarer en D-PROLOG.**

```
gks :
  alien(open-gks,4) &
  alien(close-gks,5) &
  alien(open-wks,6) &
  alien(close-wks,7) &
  alien(activ-wks,8) &
  alien(deactiv-wks,9) &
  alien(clear-wks,10) &
  alien(set-clipping,11) &
  alien(set-window,12) &
{  alien(switch-ecran,13) &   /*  inutile */ }
  alien(set-wks-viewport,14) &
  alien(select-norm,15) &
  alien(set-line-width,16) &
  alien(set-line-type,17) &
  alien(set-line-color,18) &
  alien(set-marker-type,19) &
  alien(polyline,20) &
  alien(polymarker,21) &
  alien(fill-area,22) &
  alien(pas,23) &
  alien(init-locator,24) &
  alien(request-locator,25) &
  alien(saisie-points,26) &
  alien(set-marker-color,27) &
  alien(set-marker-size,28) &
  alien(set-char-expansion,29) &
  alien(set-text-color,30) &
  alien(set-char-spacing,31) &
  alien(set-char-height,32) &
  alien(set-fill-area-color,33) &
  alien(set-text-path,34) &
  alien(set-char-up-vector,35) &
  alien(set-text-font-precis,36) &
  alien(set-text-align,37) &
  alien(text,38) &
  alien(box,39) &
  alien(arc,40) &
  alien(inquire-max-surface,41).

?save(bank,gks).
```

POLYLINE.LOG, un petit programme d'exemple PROLOG.

```
go :
  load(bank,gks) & gks &
  open-gks &
  set-window(1,100.0,300.0,100.0,200.0) &
  open-wks(screen,outin) &
  activ-wks(screen) &
  set-wks-viewport(screen,100.0,300.0,100.0,200.0) &
  tracer-cadre &
{  init-locator(screen,1,100.0,100.0,0.0,319.0,0.0,239.0) &
  pas(3.0) &
  essai-polyline &}
  sortie .

essai-polyline :
  set-line-width(2.0) &
  set-line-type(1) &
  set-line-color(1) &
  read-line(_ch) &
  make(integer,_ch,_nbpoint) &
  saisie-points(120,100,140,170,160,190,250,138) &
  polyline .

sortie :
  clear-wks &
  deactiv-wks &
  close-wks &
  close-gks.

tracer-cadre :
  set-line-width(1.0) &
  set-line-type(1) &
  set-line-color(2) &
  saisie-points(100,300,100,200) &
  polyline .

saisir-points(_n) :
  switch-ecran(false) &
  set-marker-type(1) &
  request-locator(screen,1,_n) .


oui(\o).
oui(\O).

?save(polyline).
```

PL40ALI.INC, la gestion et le contrôle des données transmises.

```
(******   * ***    ******    **     ******    ******
**    **  *** **  **    **   **    **    **  **
**    **  **      **    **   **    **    **  **  ****
*******   **      **    **   **    **    **  **    **
**        **       ******    ****   ******    ******
** (C) DELPHIA sarl )

(* (C) Sources D-Prolog : Version 3.0 du 01/sep/85 *)

          (*****************
           * INTERFACE ali *
           *****************)

(* MODULE ALI : Interface les predicats externes *)
(* ----------   declares par la primitive alien. *)

(* ===========  A NE PAS MODIFIER  ============= *)


(* CONSTANTES ET TYPES UTILES *)
(* ========================== *)

CONST
   clgid = 31;        (* taille max. identificateur *)
   clgch = 127;       (* taille max. chaine de caracteres *)
   cextsup = 255;     (* numero max. predicat externe *)

TYPE
   tindex = integer;              (* index *)
   tident = LSTRING (clgid);      (* identificateur *)
   tchcars = LSTRING (clgch);     (* chaine de caracteres *)
   tnoext = 1 .. cextsup;         (* predicat externe *)


(* FONCTIONS D'ACCES AUX PARAMETRES DU PREDICAT EXTERNE *)
(* =================================================== *)

FUNCTION nbpar                    (* donne le nombre de parametres *)
      : integer                   (* nombre de parametres *)
      ; EXTERNAL;

PROCEDURE getpar (                (* donne le nieme parametre *)
      n: integer;                 (* numero du parametre *)
  VAR t,                          (* type et *)
      v: tindex                   (* valeur du parametre *)
      ); EXTERNAL;


(* PARAMETRES EN ENTREE DU PREDICAT EXTERNE *)
(* ======================================== *)

(* FONCTIONS CONTROLANT LE TYPE D'UN PARAMETRE EN ENTREE *)
(* ---------------------------------------------------- *)

FUNCTION isvar (                  (* controle le type variable *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION isident (                (* controle le type identificateur *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION ischar (                 (* controle le type caractere *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION isint (                  (* controle le type entier court *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION islong (                 (* controle le type entier long *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION isreal (                 (* controle le type reel *)
      t: tindex                   (* type du parametre *)
      ): boolean                  (* resultat du controle *)
      ; EXTERNAL;

FUNCTION isnil (                  (* controle le type nil *)
      t,                          (* type et *)
      v: tindex                   (* valeur du parametre *)
      : boolean                   (* resultat du controle *)
      ; EXTERNAL;
```

```
(* FONCTIONS DONNANT LA VALEUR D'UN PARAMETRE EN ENTREE *)
(* ---------------------------------------------------- *)

FUNCTION getvar (              (* donne le numero de substitution *)
      v: tindex                (* valeur du parametre *)
      ): tindex                (* numero de substitution de la variable *)
      ; EXTERNAL;

FUNCTION getident (            (* donne l'identificateur *)
      v: tindex;               (* valeur du parametre *)
  VAR id: tident;              (* identificateur *)
  VAR lg: integer              (* et sa longueur *)
      ): tindex                (* et son index dans le dictionnaire *)
      ; EXTERNAL;

FUNCTION getcar (             (* donne le caractere *)
      v: tindex                (* valeur du parametre *)
      ): char                  (* caractere *)
      ; EXTERNAL;

FUNCTION getint (              (* donne l'entier court *)
      v: tindex                (* valeur du parametre *)
      ): integer               (* entier court *)
      ; EXTERNAL;

FUNCTION getlong (             (* donne l'entier long *)
      v: tindex                (* valeur du parametre *)
      ): integer4              (* entier long *)
      ; EXTERNAL;

FUNCTION getreal (             (* donne le nombre reel *)
      v: tindex                (* valeur du parametre *)
      ): real                  (* nombre reel *)
      ; EXTERNAL;

FUNCTION getstring (           (* teste et donne la chaine de caracteres *)
      t,                       (* type et *)
      v: tindex;               (* valeur du parametre *)
  VAR st: tchcars;             (* chaine de caracteres *)
  VAR lg: integer              (* et sa longueur *)
      ): boolean               (* resultat du controle *)
      ; EXTERNAL;


(* PARAMETRES EN SORTIE DU PREDICAT EXTERNE *)
(* ======================================== *)

(* FONCTIONS DE SUBSTITUTION D'UNE VARIABLE LIBRE *)
(* ---------------------------------------------- *)

FUNCTION setvar (              (* lie deux variables libres entre elles *)
      s1,                      (* numero de substitution de la variable 1 *)
      s2: tindex               (* numero de substitution de la variable 2 *)
      ): boolean               (* controle de la liaison *)
      ; EXTERNAL;

FUNCTION setident (            (* substitue la variable par l'identificateur *)
      s: tindex;               (* numero de substitution *)
      id: tident;              (* identificateur *)
      lg: integer              (* et sa longueur *)
      ): boolean               (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setchar (             (* substitue la variable par le caractere *)
      s: tindex;               (* numero de substitution *)
      c: char                  (* caractere *)
      ): boolean               (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setint (              (* substitue la variable par l'entier court *)
      s: tindex;               (* numero de substitution *)
      i: integer               (* entier court *)
      ): boolean               (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setlong (             (* substitue la variable par l'entier long *)
      s: tindex;               (* numero de substitution *)
      l: integer4              (* entier long *)
      ): boolean               (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setreal (             (* substitue la variable par le nombre reel *)
      s: tindex;               (* numero de substitution *)
      r: real                  (* nombre reel *)
      ): boolean               (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setnil (              (* substitue la variable par nil *)
      s: tindex                (* numero de la substitution *)
```

```
      ): boolean              (* controle de la substitution *)
      ; EXTERNAL;

FUNCTION setstring (          (* substitue la variable par la chaine *)
      s: tindex;              (* numero de la substitution *)
      st: tchcars;            (* chaine de caracteres *)
      lg: integer             (* et sa longueur *)
      ): boolean              (* controle de la substitution *)
      ; EXTERNAL;


(* PROCEDURE AFFECTANT LE CODE RETOUR DU PREDICAT EXTERNE *)
(* ====================================================== *)

PROCEDURE fail (
  var rc : integer
      );external;

 PROCEDURE argerr (
   var rc : integer
      );external;
                              (* En entree du predicat externe,  *)
                              (* le code retour est initialise   *)
                              (* a la valeur -1 (predicat vrai). *)

PROCEDURE ret (               (* modifie le code retour *)
      cr: integer             (* valeur a retourner *)
                              (* = 0  si predicat faux *)
                              (* = 2  si nombre d'arguments errone *)
                              (* = 3  si type d'argument errone *)
                              (* = 41 si predicat externe invalide *)
      ); EXTERNAL;
```

PL30PEX.PAS , le texte source, en MS-PASCAL, des prédicats étrangers ajoutés à D-PROLOG, dans le cas de GKS.

```
{******   * ***    ******    **      ******    ******
**    **  *** **  **    **   **    **    **  **
**    **  **      **    **   **    **    **  **  ****
*******   **      **    **   **    **    **  **    **
**        **       ******    ****   ******    ******
** (C) DELPHIA sarl )

(* (C) Sources D-Prolog : Version 3.0 du 01/sep/85 *)

MODULE PL30PEX [PUBLIC];


{DIRECTIVES DE COMPILATION}
{$DEBUG-,$LIST-,$SYMTAB-}

{$INCLUDE:'types'}

VAR [extern] PAS : integer;

VAR tablepoint : points_array(50);


{INTERFACE PREDICATS EXTERNES}
{$INCLUDE:'PL30ALI.INC'}


{INTERFACE UTILISATEUR}
FUNCTION fttylir  (* lit un caractere sans attente ni echo a l'ecran *)
    : char        (* caractere lu *)
    ; EXTERNAL;
FUNCTION fdefile
    : boolean
    ; EXTERNAL ;

{$INCLUDE:'PRIMIT.GKS'}
PROCEDURE SWITCH_ECRAN(B:boolean);extern;

           {**********************
           * PREDICATS EXTERNES *
           **********************}

PROCEDURE callext(     (* evalue un predicat externe *)
      prn: tnoext;     (* numero du predicat externe *)
  VAR rc: integer);    (* code retour renvoye *)

  (*** LIT UN CARACTERE SUR L'ECRAN ***)
  PROCEDURE xttyin;
  VAR t, v: tindex;
  BEGIN
    IF nbpar = 1 THEN
    BEGIN
      getpar(1,t,v);
      IF isvar(t) THEN
        IF setchar(getvar(v),fttylir) THEN (* ok *)
        ELSE fail(rc)                      (* echec *)
      ELSE rc := 3;                        (* type d'argument errone *)
    END ELSE rc := 2;                      (* nombre d'arguments errone *)
  END;

  (*** ECRIT DES CARACTERES SUR L'ECRAN ***)
  PROCEDURE xttyout;
  VAR t, v: tindex; st: tchcars; lg: integer;
  BEGIN
    IF nbpar = 1 THEN
    BEGIN
      getpar(1,t,v);
      IF getstring(t,v,st,lg) THEN
      BEGIN
        {MS() st[0] := chr(lg); {MS)}
        write(output,st:lg) (* ok *)
      END ELSE rc := 3;      (* type d'argument errone *)
    END ELSE rc := 2;        (* nombre d'arguments errone *)
  END;

  (*** CONTROLE DU DEFILEMENT D'UN TEXTE  ***)
PROCEDURE xcontrol;
BEGIN
  IF nbpar = 0 THEN
   IF fdefile THEN          (* ok *)
      ELSE fail(rc)         (* echec *)
  ELSE rc := 2;
END;

  (*** PROCEDURES GRAPHIQUES GKS ***)

(*** FONCTIONS UTILES POUR GKS ***)
FUNCTION pair(nb : integer):boolean;
BEGIN
```

```
  pair := true;
  IF (nb mod 2) = 1 THEN pair := false;
END;

FUNCTION isboolean(v : tindex; var lbool : boolean) : boolean;
var
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isboolean := true;
  IF (lid='TRUE') OR (lid='true') THEN
    lbool := true
  ELSE
    IF (lid='FALSE') OR (lid='false') THEN
      lbool := false
    ELSE isboolean := false;
END;


FUNCTION iswkstype (v : tindex; var wkstype : wks_type):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  iswkstype := true;
  IF (lid='SCREEN') OR (lid='screen') THEN
     wkstype := SCREEN ELSE
     IF (lid='PLOTTER') OR (lid='plotter') THEN
         wkstype := PLOTTER ELSE
         iswkstype := false;
END;

FUNCTION isdirection (v : tindex; var ldirect : direction):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isdirection := true;
  IF (lid='RIGHT') OR (lid='right') THEN
     ldirect := RIGHT ELSE
     IF (lid='LEFT') OR (lid='left') THEN
         ldirect := LEFT ELSE
         IF (lid='UP') OR (lid='up') THEN
            ldirect := UP ELSE
            IF (lid='DOWN') OR (lid='down') THEN
               ldirect := DOWN ELSE
               isdirection := false;
END;

FUNCTION isdevunit (v : tindex; var ldevunit : device_units):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isdevunit := true;
  IF (lid='METRES') OR (lid='metres') THEN
     ldevunit := METRES ELSE
     IF (lid='OTHER') OR (lid='other') THEN
         ldevunit := OTHER ELSE
         isdevunit := false;
END;

FUNCTION isprecision (v : tindex; var lprecis : precision):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isprecision := true;
  IF (lid='CHAIN') OR (lid='chain') THEN
     lprecis := CHAIN ELSE
     IF (lid='CHARACTER') OR (lid='character') THEN
         lprecis := CHARACTER ELSE
         IF (lid='STROKE') OR (lid='stroke') THEN
            lprecis := STROKE ELSE
            isprecision := false;
END;

FUNCTION ishorizontal (v : tindex; var lhoriz : horizontal_align):boolean;
VAR
  lt : tindex;
```

```
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  ishorizontal := true;
  IF (lid='HNORMAL') OR (lid='hnormal') THEN
     lhoriz := HNORMAL ELSE
     IF (lid='HLEFT') OR (lid='hleft') THEN
         lhoriz := HLEFT ELSE
         IF (lid='HCENTER') OR (lid='hcenter') THEN
            lhoriz := HCENTER ELSE
            IF (lid='HRIGHT') OR (lid='hright') THEN
               lhoriz := HRIGHT ELSE
               ishorizontal := false;
END;

FUNCTION isvertical (v : tindex; var lvertic : vertical_align):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isvertical := true;
  IF (lid='VNORMAL') OR (lid='vnormal') THEN
     lvertic := VNORMAL ELSE
     IF (lid='VTOP') OR (lid='vtop') THEN
         lvertic := VTOP ELSE
         IF (lid='VCAP') OR (lid='vcap') THEN
            lvertic := VCAP ELSE
            IF (lid='VHALF') OR (lid='vhalf') THEN
               lvertic := VHALF ELSE
               IF (lid='VBASE') OR (lid='vbase') THEN
                  lvertic := VBASE ELSE
                  IF (lid='VBOTTOM') OR (lid='vbottom') THEN
                     lvertic := VBOTTOM ELSE
                     isvertical := false;
END;

FUNCTION isetatinput (v : tindex; var letainput : etat_input):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  isetatinput := true;
  IF (lid='OK') OR (lid='ok') THEN
     letainput := OK ELSE
     IF (lid='NONE') OR (lid='none') THEN
         letainput := NONE ELSE
         isetatinput := false;
END;

FUNCTION iswkscateg (v : tindex; var lwkscat : wks_category):boolean;
VAR
  lt : tindex;
  lid : tident;
  lgid : integer;
BEGIN
  lt := getident(v,lid,lgid);
  iswkscateg := true;
  IF (lid='INPUT') OR (lid='input') THEN
     lwkscat := INPUT ELSE
     IF (lid='OUTIN') OR (lid='outin') THEN
         lwkscat := OUTIN ELSE
         IF (lid='OUTPUT') OR (lid='output') THEN
            lwkscat := OUTPUT ELSE
            iswkscateg := false;
END;


PROCEDURE xopengks;    (*** OPEN_GKS ***)
BEGIN
  IF nbpar=0 THEN
    OPEN_GKS
  ELSE rc := 2;
END;

PROCEDURE xclosegks;    (*** CLOSE_GKS ***)
BEGIN
  IF nbpar=0 THEN
    CLOSE_GKS
  ELSE rc := 2;
END;

PROCEDURE xopenwks;    (*** OPEN_WORKSTATION ***)
VAR
   t1,v1,t2,v2 : tindex;
   lwkstyp : wks_type;
```

```
   lwkscat : wks_category;
BEGIN
  IF nbpar=2 THEN
  BEGIN
    getpar(1,t1,v1);
    getpar(2,t2,v2);
    IF (isident(t1)) AND (isident(t2)) THEN
      IF (iswkstype(v1,lwkstyp)) AND (iswkscateg(v2,lwkscat)) THEN
      OPEN_WORKSTATION(lwkstyp,lwkscat)            (* OK *)
      ELSE rc := 3
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xclosewks;   (*** CLOSE_WORKSTATION ***)
VAR
   t,v : tindex;
   lwkstyp : wks_type;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF iswkstype(v,lwkstyp) THEN
        CLOSE_WORKSTATION(lwkstyp)          (** OK **)
      ELSE
          rc := 3                           (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                           (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                         (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xactivwks;   (*** ACTIVATE_WORKSTATION ***)
VAR
   t,v : tindex;
   lwkstyp : wks_type;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF iswkstype(v,lwkstyp) THEN
        ACTIVATE_WORKSTATION(lwkstyp)       (** OK **)
      ELSE
        rc := 3                             (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                           (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                         (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xdeactivwks;   (*** DEACTIVATE_WORKSTATION ***)
VAR
   t,v : tindex;
   lwkstyp : wks_type;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF iswkstype(v,lwkstyp) THEN
        DEACTIVATE_WORKSTATION(lwkstyp)     (** OK **)
      ELSE
        rc := 3                             (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                           (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                         (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xclearwks;   (*** CLEAR_WORKSTATION ***)
VAR
   t,v : tindex;
   lwkstyp : wks_type;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF iswkstype(v,lwkstyp) THEN
        CLEAR_WORKSTATION(lwkstyp)          (** OK **)
      ELSE
        rc := 3                             (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                           (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                         (** MAUVAIS NB ARGUMENT **)
END;


PROCEDURE xsetclipping;   (*** SET_CLIPPING_INDICATOR ***)
VAR
   t,v : tindex;
   lindic : boolean;
BEGIN
  IF nbpar=1 THEN
  BEGIN
```

```
    getpar(1,t,v);
    IF isident(t) THEN
      IF isboolean(v,lindic) THEN
        SET_CLIPPING_INDICATOR(lindic)     (** OK **)
      ELSE
        rc := 3                            (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetwindow ;    (*** SET_WINDOW ***)
VAR
  t1,v1,t2,v2,t3,v3,t4,v4,t5,v5 : tindex;
BEGIN
  IF nbpar=5 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2);
    getpar(3,t3,v3); getpar(4,t4,v4);
    getpar(5,t5,v5);
    IF (isint(t1)) AND (isreal(t2)) AND (isreal(t3))
    AND (isreal(t4)) and (isreal(t5)) THEN
      SET_WINDOW(getint(v1),getreal(v2),getreal(v3),getreal(v4),getreal(v5))
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xswitchecran;   (*** SWITCH_ECRAN ***)
VAR
   t,v : tindex;
   lbool : boolean;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF isboolean(v,lbool) THEN
        SWITCH_ECRAN(lbool)     (** OK **)
      ELSE
        rc := 3                            (** MAUVAIS ARGUMENT **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetwksview ;    (*** SET_WORKSTATION_VIEWPORT ***)
VAR
  t1,v1,t2,v2,t3,v3,t4,v4,t5,v5 : tindex;
  lwkstyp : wks_type;
BEGIN
  IF nbpar=5 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2);
    getpar(3,t3,v3); getpar(4,t4,v4);
    getpar(5,t5,v5);
    IF isident(t1) THEN
    BEGIN
      IF (iswkstype(v1,lwkstyp)) AND (isreal(t2)) AND
      (isreal(t3)) AND (isreal(t4)) and (isreal(t5)) THEN
        SET_WORKSTATION_VIEWPORT(lwkstyp,getreal(v2),getreal(v3)
                                 ,getreal(v4),getreal(v5))
      ELSE rc := 3;
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xselectnorm;   (*** SELECT_NORMALIZATION_TRANSFORMATION ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SELECT_NORMALIZATION_TRANSFORMATION(getint(v))     (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetlinewidth;   (*** SET_LINEWIDTH_SCALE_FACTOR ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isreal(t) THEN
      SET_LINEWIDTH_SCALE_FACTOR(getreal(v))     (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;
```

```
PROCEDURE xsetlinetyp;   (*** SET_LINETYPE ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_LINETYPE(getint(v))              (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetlinecolor;   (*** SET_POLYLINE_COLOUR_INDEX ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_POLYLINE_COLOUR_INDEX(getint(v))     (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetmarktyp;   (*** SET_MARKER_TYPE ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_MARKER_TYPE(getint(v))           (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xpolyline;     (*** POLYLINE   ***)
VAR
  t1,v1 : tindex;
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t1,v1);
    IF isint(t1) THEN
    BEGIN
      SWITCH_ECRAN(true);
      POLYLINE(getint(v1),tablepoint);
      SWITCH_ECRAN(false);
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xpolymarker;     (*** POLYMARKER   ***)
VAR
  t1,v1 : tindex;
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t1,v1);
    IF isint(t1) THEN
    BEGIN
      SWITCH_ECRAN(true);
      POLYMARKER(getint(v1),tablepoint);
      SWITCH_ECRAN(false);
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xfillarea;     (*** FILL_AREA   ***)
VAR
  t1,v1 : tindex;
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t1,v1);
    IF isint(t1) THEN
    BEGIN
      SWITCH_ECRAN(true);
      FILL_AREA(getint(v1),tablepoint);
      SWITCH_ECRAN(true);
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xpas;     (*** INITIALISATION DE L'ENTIER PAS  ***)
VAR
   t,v : tindex;
```

```
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN PAS := getint(v)
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xinitlocat;    (*** INITIALISE_LOCATOR  ***)
VAR
  t1,v1,t2,v2,t3,v3,t4,v4,t5,v5,t6,v6,t7,v7,t8,v8,t9,v9 : tindex;
  lwkstyp : wks_type;
  ptdep : point;
BEGIN
  IF nbpar = 9 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2); getpar(3,t3,v3);
    getpar(4,t4,v4); getpar(5,t5,v5); getpar(6,t6,v6);
    getpar(7,t7,v7); getpar(8,t8,v8); getpar(9,t9,v9);
    IF isident(t1) THEN
    BEGIN
      IF (iswkstype(v1,lwkstyp)) AND (isint(t2)) AND (isint(t3)) AND
         (isint(t4)) AND (isint(t5)) AND (isreal(t6)) AND
         (isreal(t7)) AND (isreal(t8)) AND (isreal(t9)) THEN
      BEGIN
       ptdep.X := getint(v3);
       ptdep.Y := getint(v4);
       INITIALISE_LOCATOR(lwkstyp,getint(v2),ptdep,getint(v5),getreal(v6),
                          getreal(v7),getreal(v8),getreal(v9));
      END ELSE rc := 3;
    END ELSE RC := 3;
  END ELSE rc := 2;
END;

PROCEDURE xrequlocat;    (*** REQUEST_LOCATOR  ***)
VAR
  t1,v1,t2,v2,t3,v3: tindex;
  li,nbpoint,locnb : integer;
  lwkstyp : wks_type;
  letainput : etat_input;
  lnotrans : integer;
  ptok : point;
  pointsaisi : points_array(1);
BEGIN
  IF nbpar = 3 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2); getpar(3,t3,v3);
    IF isident(t1) THEN
    BEGIN
      IF (iswkstype(v1,lwkstyp)) AND (isint(t2)) AND (isint(t3)) THEN
      BEGIN
        li := 1; nbpoint := getint(v3); locnb := getint(v2);
        WHILE li <= nbpoint DO
        BEGIN
          SWITCH_ECRAN(true);
          REQUEST_LOCATOR(lwkstyp,getint(v2),letainput,lnotrans,ptok);
          CASE letainput OF
            OK : BEGIN
                 tablepoint[li] := ptok;
                 pointsaisi[1] := ptok;
                 POLYMARKER(1,pointsaisi);
                 END;
          NONE : ;
          END;
          SWITCH_ECRAN(false);
          li := li + 1;
        END;
      END ELSE rc := 3;
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xsaisiedirecte;
VAR t1,v1,t2,v2,t3,v3 : tindex;
    lnbpar,lnbpoint,li,lj : integer;
    lerreur : boolean;
BEGIN
  lnbpar := nbpar;
  IF lnbpar >= 3 THEN
  BEGIN
    getpar(1,t1,v1);
    IF isint(t1) THEN
    BEGIN
      lnbpoint := getint(v1);
      IF (lnbpoint = (lnbpar div 2)) THEN
      BEGIN
        li := 2; lj := 1; lerreur := false;
        WHILE (li < lnbpar) AND (not lerreur) DO
        BEGIN
```

```
          getpar(li,t2,v2); getpar(li+1,t3,v3);
          IF isreal(t2) AND isreal(t3) THEN
          BEGIN
            tablepoint[lj].X := getreal(v2);
            tablepoint[lj].Y := getreal(v3);
          END ELSE lerreur := true;
          li := li + 2; lj := lj + 1;
        END;
        IF lerreur THEN rc := 3;
      END ELSE rc := 2;
    END ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xsetmarkercolor;      (*** SET_POLYMARKER_COLOUR_INDEX ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_POLYMARKER_COLOUR_INDEX(getint(v))     (** OK **)
    ELSE rc := 3;                                (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                              (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetmarkersize;       (*** SET_MARKER_SIZE_SCALE_FAVTOR ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isreal(t) THEN
      SET_MARKER_SIZE_SCALE_FACTOR(getreal(v))     (** OK **)
    ELSE rc := 3;                                  (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                                (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xsetcharexp;     (*** SET_CHARACTER_EXPANSION_FACTOR ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isreal(t) THEN
      SET_CHARACTER_EXPANSION_FACTOR(getreal(v))     (** OK **)
    ELSE rc := 3;                                    (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                                  (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xcharspace ;     (*** SET_CHARACTER_SPACING ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isreal(t) THEN
      SET_CHARACTER_SPACING(getreal(v))     (** OK **)
    ELSE rc := 3;                           (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                         (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xtextcolor;    (*** SET_TEXT_COLOUR_INDEX ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_TEXT_COLOUR_INDEX(getint(v))     (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xcharheight;    (*** SET_CHARACTER_HEIGHT ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isreal(t) THEN
      SET_CHARACTER_HEIGHT(getreal(v))     (** OK **)
    ELSE rc := 3;                          (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                        (** MAUVAIS NB ARGUMENT **)
```

```
END;

PROCEDURE xfillareacolor;   (*** SET_FILL_AREA_COLOUR_INDEX ***)
VAR
   t,v : tindex;
BEGIN
  IF nbpar=1 THEN
  BEGIN
    getpar(1,t,v);
    IF isint(t) THEN
      SET_FILL_AREA_COLOUR_INDEX(getint(v))     (** OK **)
    ELSE rc := 3;                               (** MAUVAIS ARGUMENT **)
  END ELSE rc := 2;                             (** MAUVAIS NB ARGUMENT **)
END;

PROCEDURE xtextpath; (*** SET_TEXT_PATH ***)
VAR t,v : tindex;
    ldirect : direction;
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t,v);
    IF isident(t) THEN
      IF isdirection(v,ldirect) THEN
        SET_TEXT_PATH(ldirect)
      ELSE rc := 3
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xcharvector;  (*** SET_CHARACTER_UP_VECTOR ***)
VAR t1,v1,t2,v2 : tindex;
BEGIN
  IF nbpar = 2 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2);
    IF isreal(t1) AND isreal(t2) THEN
      SET_CHARACTER_UP_VECTOR(getreal(v1),getreal(v2))
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xtextfontprecis;  (*** SET_TEXT_FONT_AND_PRECISION ***)
VAR t1,v1,t2,v2 : tindex;
    lqualit : precision;
BEGIN
  IF nbpar = 2 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2);
    IF isident(t2) THEN
      IF isint(t1) AND isprecision(v2,lqualit) THEN
        SET_TEXT_FONT_AND_PRECISION(getint(v1),lqualit)
      ELSE rc := 3
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xtextalign;   (*** SET_TEXT_ALIGNMENT ***)
VAR t1,v1,t2,v2 : tindex;
    lhoriz : horizontal_align;
    lvert : vertical_align;
BEGIN
  IF nbpar = 2 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2);
    IF isident(t1) AND isident(t2) THEN
      IF ishorizontal(v1,lhoriz) AND isvertical(v2,lvert) THEN
        SET_TEXT_ALIGNMENT(lhoriz,lvert)
      ELSE rc := 3
    ELSE rc := 3;
  END ELSE rc := 2;
END;

PROCEDURE xtext;      (*** TEXT ***)
VAR t1,v1 : tindex;
    lchaine : tchcars;
    lg : integer;
BEGIN
  IF nbpar = 1 THEN
  BEGIN
    getpar(1,t1,v1);
    IF getstring(t1,v1,lchaine,lg) THEN
    BEGIN
      SWITCH_ECRAN(true);
      TEXT(tablepoint[1],lchaine);
      SWITCH_ECRAN(false);
    END ELSE rc := 3;
  END ELSE rc := 2;
END;
```

```
PROCEDURE xbox;      (***  BOX  ***)
BEGIN
  IF nbpar=0 THEN
  BEGIN
    SWITCH_ECRAN(true);
    BOX(tablepoint[1],tablepoint[2]);
    SWITCH_ECRAN(false);
  END ELSE rc := 2;
END;

PROCEDURE xarc;      (*** ARC ***)
BEGIN
  IF nbpar = 0 THEN
  BEGIN
    SWITCH_ECRAN(true);
    ARC(tablepoint[1],tablepoint[2],tablepoint[3]);
    SWITCH_ECRAN(false);
  END ELSE rc := 2;
END;

PROCEDURE xinqmaxsurf;    (*** INQUIRE_MAXIMUM_DISPLAY_SURFACE_SIZE ***)
VAR t1,v1,t2,v2,t3,v3,t4,v4,t5,v5,t6,v6,
    t7,v7,t8,v8,t9,v9,t10,v10,t11,v11  : tindex;
    lwkstyp : wks_type;
    lerreur,lgid : integer;
    rep1,rep2,rep3,rep4,rep5,rep6 : boolean;
    lident : tident;
    ldevunit : device_units;
    xu,yu,xr,yr : real;
BEGIN
  IF nbpar = 11 THEN
  BEGIN
    getpar(1,t1,v1); getpar(2,t2,v2); getpar(3,t3,v3);
    getpar(4,t4,v4); getpar(5,t5,v5); getpar(6,t6,v6);
    getpar(7,t7,v7);
    IF isident(t1) THEN
    BEGIN
      IF iswkstype(v1,lwkstyp) AND isvar(t2) AND isvar(t3) AND
      isvar(t4) AND isvar(t5) AND isvar(t6) AND isvar(t7) THEN
      BEGIN
        INQUIRE_MAXIMUM_DISPLAY_SURFACE_SIZE(lwkstyp,lerreur,ldevunit,
                                             xu,yu,xr,yr);
        rep1 := setint(getvar(v2),lerreur);
        rep3 := setreal(getvar(v4),xu); rep4 := setreal(getvar(v5),yu);
        rep5 := setreal(getvar(v6),xr); rep6 := setreal(getvar(v7),yr);
        CASE ldevunit OF
          metres : BEGIN
                     lident := 'metres'; lident[0] := chr(6);
                     lgid := 6;
                   END;
          other  : BEGIN
                     lident := 'other'; lident[0] := chr(5);
                     lgid := 5;
                   END;
        END;
        rep2 := setident(getvar(v3),lident,lgid);
        IF rep1 AND rep2 AND rep3 AND rep4 AND rep5 AND rep6 THEN
          (*** OK ***)
        ELSE fail(rc);       (*** ECHEC ***)
      END ELSE rc := 3;
    END ELSE rc := 3;
  END ELSE rc := 2;
END;


BEGIN (* callext *)
  IF (prn > 0) AND (prn < 37) THEN
    CASE prn OF
      1: xttyin;
      2: xttyout;
      3: xcontrol;
      4: xopengks;
      5: xclosegks;
      6: xopenwks;
      7: xclosewks;
      8: xactivwks;
      9: xdeactivwks;
     10: xclearwks;
     11: xsetclipping;
     12: xsetwindow;
     13: xswitchecran;
     14: xsetwksview;
     15: xselectnorm;
     16: xsetlinewidth;
     17: xsetlinetyp;
     18: xsetlinecolor;
     19: xsetmarktyp;
```

```
      22: xfillarea;
      23: xpas;
      24: xinitlocat;
      25: xrequlocat;
      26: xsaisiedirecte;
      27: xsetmarkercolor;
      28: xsetmarkersize;
      29: xsetcharexp;
      30: xtextcolor;
      31: xcharspace;
      32: xcharheight;
      33: xfillareacolor;
      34: xtextpath;
      35: xcharvector;
      36: xtextfontprecis;
      37: xtextalign;
    END
  ELSE rc := 41;      (* predicat externe inconnu *)
END; (* callext *)

END. { MODULE }
```

Annexe III-1 : Description des fonctions de l'interface

Description complète des fonctions utilisateurs fournies par l'interface.

L'interface seule dispose des commandes usuelles de manipulation du projet. La figure A.19 montre une image du système à un instant donné; on remarquera plus particulièrement la zone de messages qui contient des informations sur le composant sélectionné. Les commandes sont organisées en trois groupes :

1. Le premier groupe concerne la création d'un composant et des fonctionnalités sur l'ensemble du projet. La création de composant (**créer**) fait partie de cette catégorie de commandes car dans tout système de CAO, on ne crée pas un composant, mais on l'instancie à partir de la base de composants du système. Ainsi nous avons considéré que cette instanciation faisait partie des actions concernant le projet en entier. Cependant, il est clair que la création d'un composant nécessite également son placement dans l'espace et donc peut provoquer des conflits géométriques.
   Les autres fonctions de ce groupe sont :

   a. la vérification de la cohérence du projet (nous reviendrons sur ce point par la suite)(**cohérence**),
   b. le pas de grille affiché et permettant un placement précis et simple des composants dans l'espace (**grille**),
   c. le choix d'un composant existant (**choisir**), pour ensuite le manipuler,
   d. le rafraichissement de la fenêtre d'édition (**rafraichir**).

2. Le deuxième groupe de commandes concerne la manipulation de l'enveloppe du rectangle définissant le composant, c'est à dire changer la position du coin haut gauche d'un composant (**coin**), ou les deux coins (**enveloppe**).

3. Enfin le dernier groupe de commandes concerne des opérations plus spécifiques d'une action sur un composant. Il s'agit du déplacement d'un composant (**déplacer**) dans une direction horizontale ou verticale, du déport d'un composant dans une direction quelconque (**déporter**), de la destruction d'un composant (**détruire**).

Nous décrivons maintenant ce qu'est censé faire chacune des commandes citées précédemment. Dans chaque cas nous décrivons l'interaction nécessaire et les effets de bord engendrés par cette commande.

Annexe III-1 : Description des fonctions de l'interface

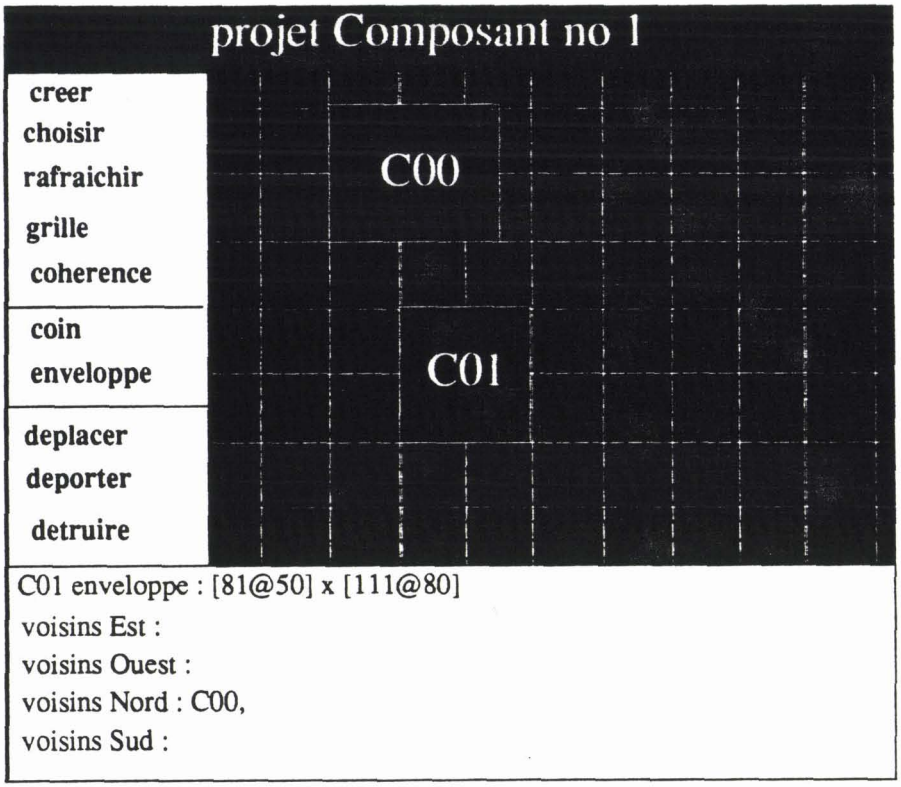


Fig A.19

Création de composant (par instanciation) : créer

lorsque l'utilisateur sélectionne ce choix de menu, le système affiche automatiquement un "pop-up" menu, qui offre à l'utilisateur le choix entre la confirmation de création du composant ou un échappement. S'il confirme son choix, on lui propose dans une fenêtre d'édition un nom de composant qu'il pourra modifier à volonté.
Les noms de composants sont des identificateurs uniques au sein d'un même projet. Ils sont constitués de la lettre 'C', suivie d'un nombre à deux chiffres. Par exemple "C24" est un nom de composant valide, par contre "P12" ne l'est pas, ni "C123".
On lui montre alors un curseur spécialisé qui va lui permettre de placer cette nouvelle instance de composant à l'endroit qu'il désire dans l'espace. Il déplacera ce curseur avec une souris et clicquera sur le bouton gauche pour valider la place du composant à l'endroit courant du curseur.

choix d'un composant à manipuler : choisir

Pour choisr un nouveau composant à manipuler, l'utilisateur sélectionne la case "choisir" .

Le système lui propose alors une liste des noms de composants existant à cet instant précis du projet, sous forme d'un menu "pop-up".

L'utilisateur choisit l'un d'eux et c'est ce composant qui sera alors sélectionné et qu'il pourra manipuler.

Annexe III-1 : Description des fonctions de l'interface

rafraîchissement du schéma : rafraîchir

Lorsque l'utilisateur a pratiqué beaucoup de manipulations sur le schéma, il peut demander au système de redessiner proprement l'état courant du projet, c'est une facilité classique en édition graphique

Pour cela, il choisit l'option "rafraichir" du menu principal.

L'état complet du projet ainsi que la grille de la fenêtre d'édition sont redessinés.

sélection du pas de grille : grille

L'utilisateur peut fixer à son gré le pas de grille représenté dans la fenêtre d'édition.

Pour cela, il sélectionne l'option "grille" du menu principal.

Le système affiche une fenêtre interactive d'édition de texte dans laquelle est affiché la valeur courante du pas de grille. Il peut positionner cette valeur à celle qu'il désire, puis la valider. Le système pratique alors un rafraichissement complet de la zone d'édition avec la nouvelle valeur de pas de grille.

vérification de la cohérence de la base du projet : cohérence

Les différents composants sont liés entre eux par des relations topologiques de voisinage que nous préciserons par la suite. Si l'utilisateur veut vérifier la cohérence des liens entre composants, il sélectionnera l'option "cohérence" du menu principal.

Le système lui présente alors une liste des composants pour lesquels il a détecté une erreur et laquelle.

modification de la forme d'un composant : l'option coin

L'utilisateur peut agrandir ou rétrécir la taille des composants en déplacant le coin haut gauche du composant sélectionné.

Pour cela, il choisit l'option "coin" du menu principal.

L'enveloppe du composant apparait alors en traits pointillés et il peut modiifer à volonté la position du coin haut gauche du composant en déplacant la souris.. Après validation, le nouvel état du composant ainsi que celui de la base est rémis à jour.

modification de la forme d'un composant : l'option enveloppe

L'utilisateur peut également modifier en une seule opération l'enveloppe complète d'un

composant en sélectionnant l'option "enveloppe" du menu principal.

L'enveloppe du composant apparait alors en traits pointillés, il peut la déplacer à l'aide de la souris et changer la forme du composant en validant par un "click" souris le coin haut gauche puis le coin bas droit du composant.

déplacements de composant : l'option déplacer

On peut déplacer le composant dans une direction horizontale ou verticale. Notons que cette opération ne provoque pas de destruction et recréation de l'instance du composant, mais simplement des remises à jour des liens entre composants ainsi que de l'enveloppe et des attributs du composant déplacé.

L'utilisateur doit choisir l'option "déplacer" du menu principal.

Le système demande alors un choix entre les quatres directions suivantes : Est, Ouest, Nord, Sud. Après validation du choix, le composant sélectionné apparait en traits pointillés et peut être déplacé interactivement grâce à la souris et un écho spécialisé représentant son déplacement. Après validation de la nouvelle position, les attributs du composant sont remis à jour, ainsi que les liens entre tous les composants du projet.

déplacements de composant : l'option déporter

Dans notre système, déporter un composant consiste à le déplacer dans l'espace de la carte dans n'importe quelle direction. A la différence de l'action "déplacer", cette opération nécessite une destruction de l'instance courante du composant, et une recréation de celui-ci dans une nouvelle position avec de nouveaux attributs. Nous préciserons plus loin pour quelles raisons cette destruction-recréation est obligatoire.

Lorsque l'utilisateur a choisit l'option "déporter" du menu principal, le composant apparait en traits pointillés et peut alors être "déporté" en n'importe quel endroit de la carte, à l'aide de la souris et d'un écho spécialisé ayant la forme de l'enveloppe du composant, écho qui suit les mouvements de la souris. Les attributs du composant déporté sont alors remis à jour ainsi que les liens entre les composants existant dans le projet en cours.

destruction de composant : l'option détruire

On peut bien sûr détruire une instance de composant à tout moment, en choisissant l'option " détruire" du menu principal. Le composant sélectionné est purement et simplement détruit. La base du projet et les liens entre les composants restant sont remis à jour.

```
Object subclass: #ComposantRectangle
  instanceVariableNames:
    'nom enveloppe voisins projet '
  classVariableNames:
    'Vertical Nord Ouest SortBlocks Est Negatif Positif Horizontal Sud '
  poolDictionaries: ''
ComposantRectangle class methods:
chainePour: orientation
  "convertirOrientation - Renvoie le nom de variable de classe correspondant
   a la valeur orientation. Renvoie nil si orientation ne correspond a rien."

  "teste la validite de orientation."
  (self testerOrientation: orientation)
  ifFalse: [^nil].

  "convertit orientation en chaine."
  (orientation = Est)   ifTrue: [^'Est'].
  (orientation = Ouest) ifTrue: [^'Ouest'].
  (orientation = Nord)  ifTrue: [^'Nord'].
  (orientation = Sud)   ifTrue: [^'Sud'].
-------------------------------------------------------------------------------
est
  "orientation - renvoie l'orientation Est."
  ^Est
-----------------------------------------------
horizontal
  "composantes - renvoie la valeur de Horizontal."
  ^Horizontal
-------------------------------------------------
initialiseBlocksDeTri
  "initialiserClasseComposant - initialise les blocks de tri pour
   les SortedCollection composant les voisinages. Ne renvoie rien."

  "Les blocks de tri sont memorises par un tableau indice par les
   4 orientations possibles."
  SortBlocks:= Array new: 4.

  "a l'est, les composants du voisinage sont tries par ordre croissant
   de leur cote gauche d'enveloppe, et sinon par ordre croissant de
   leur position sur la meme verticale."
  SortBlocks at: Est put:
    [:a1 :a2 |
     (a1 gauche < a2 gauche) or:
     [(a1 gauche = a2 gauche) and: [a1 bas < a2 haut]]
    ].

  "a l'ouest, les composants du voisinage sont tries par ordre decroissant
   de leur cote droit d'enveloppe, et sinon par ordre croissant de
   leur position sur la meme verticale ."
  SortBlocks at: Ouest put:
    [:a1 :a2 |
     (a1 droit > a2 droit) or:
     [(a1 droit = a2 droit) and: [a1 bas < a2 haut]]
    ].

  "au nord, les composants du voisinage sont tries par ordre decroissant
   de leur cote bas d'enveloppe, et sinon par ordre croissant de
   leur position sur la meme horizontale."
  SortBlocks at: Nord put:
    [:a1 :a2 |
     (a1 bas > a2 bas) or:
     [(a1 bas = a2 bas) and: [a1 droit < a2 gauche]]
    ].

  "au sud, les composants du voisinage sont tries par ordre croissant
   de leur cote haut d'enveloppe, et sinon par ordre croissant de
   leur position sur la meme horizontale."
  SortBlocks at: Sud put:
    [:a1 :a2 |
     (a1 haut < a2 haut) or:
     [(a1 haut = a2 haut) and: [a1 droit < a2 gauche]]
    ].
-------------------------------------------------------------------------------
```

```
initialiseComposantesOrientation
  "initialiserClasseComposant - Initialise les 4 variables de classe
   permettant de convertir les orientations en indice de tableaux. Ne
   renvoie rien."

  "Positif et Negatif permettent de differencier Est et Ouest, ainsi
   que Nord et Sud. Indiquent le sens de l'orientation."
  Positif := 1.
  Negatif := 2.

  "Horizontal et Vertical permettent de differencier les directions
   horizontales et verticales."
  Horizontal := 0.
  Vertical := 2.

  "Est indique un sens croissant sur la direction horizontale, ce qui
   est traduit par Est = Horizontal + Positif.
                   Ouest = Horizontal + Negatif.
   Nord indique un sens croissant sur la direction verticale, ce qui
   est traduit par Nord = Vertical + Positif.
                   Sud = Vertical + Negatif.
   On obtient ainsi les indices de tableau par la valeur de
   direction + sens."
-------------------------------------------------------------------------
initialiseOrientations
  "initialiserClasseComposant - initialise les valeurs des 4 variables
   de classe decrivant les couples (direction, sens) possibles .
   Ne renvoie rien."
  Est := 1.
  Ouest := 2.
  Nord := 3.
  Sud := 4.
-------------------------------------------------------------------------
initialiser
  "initialiserClasseComposant - initialise des variables de classe.
     initialise les valeurs d'orientations Est Ouest Nord Sud.
     initialise les valeurs des composantes d'orientations:
      Positif, Negatif; Horizontal, Vertical.
     initialise les blocks de tri pour les voisinages attaches a
     chaque orientation: SortBlocks (Est, Ouest, Nord, Sud).
   Ne renvoie rien."
  self
   initialiseOrientations;
   initialiseComposantesOrientation;
   initialiseBlocksDeTri.
-------------------------------------------------------------------------
negatif
  "composantes - renvoie la valeur de Negatif."
  ^Negatif
-------------------------------------------------
```

```
nom: troisCars enveloppe: unRectangle projet: unProjet
  "creation - Creation d'une instance de ComposantRectangle ayant comme
   attributs ceux passes en parametres. Verifie la syntaxe du nom,
   la coherence du rectangle, la validite du projet. Renvoie ce composant
   si les verifications se sont bien passees, nil sinon ."
  | composant |

  "verifie le nom"
  (troisCars isMemberOf: String)
  ifFalse: [Menu message: 'le nom doit etre une chaine'. ^nil ]
  ifTrue:
    [troisCars size ~= 3
       ifTrue: [Menu message: 'le nom doit comporter 3 caracteres'. ^nil]
       ifFalse:
  [(troisCars at: 1) = $C
         ifFalse: [Menu message: 'Le nom doit commencer par C'. ^nil] ]
    ].

  "verifie l'enveloppe du composant."
  (unRectangle isMemberOf: Rectangle)
  ifFalse: [Menu message: 'l''enveloppe doit etre un rectangle'. ^nil]
  ifTrue:
    [unRectangle corner >= unRectangle origin
     ifFalse:
       [Menu message: 'l''enveloppe doit etre un rectangle Smalltalk'. ^nil]
    ].

  "controle le projet associe."
  (unProjet isMemberOf: ProjetComposant)
  ifFalse:
    [Menu
      message: 'le composant doit etre rattache a un projetComposant'. ^nil ].

  "Ce composant peut etre cree."
  composant := self new
    nom: troisCars enveloppe: unRectangle projet: unProjet.
  composant placement.
  unProjet ajoutComposant: composant.
  ^composant.
-------------------------------------------------------------------------------
nord
  "orientation - renvoie l'orientation Nord."
  ^Nord
-----------------------------------------------
orientationPour: chaine
  "convertirOrientation - renvoie l'orientation correspondant a la
   chaine si elle existe. Renvoie nil sinon."
  (chaine = 'Est')   ifTrue: [^Est].
  (chaine = 'Ouest') ifTrue: [^Ouest].
  (chaine = 'Nord')  ifTrue: [^Nord].
  (chaine = 'Sud')   ifTrue: [^Sud].
  ^nil
-----------------------------------------------------------------
ouest
  "orientation - renvoie l'orientation Ouest."
  ^Ouest
-----------------------------------------------
pointCardinalPour: orientation
  "convertirOrientation - decompose orientation en ses composantes de direction
   et de sens et renvoie celles-ci dans un point. Renvoie nil si orientation
   ne correspond a rien."

  "teste la validite de orientation."
```

```
  (ComposantRectangle testerOrientation: orientation)
  ifFalse: [^nil].

  "renvoie les valeurs dans un point."
  ^((orientation >= 3) ifFalse: [Horizontal] ifTrue: [Vertical]) @
   ((orientation odd)  ifFalse: [Negatif]    ifTrue: [Positif])
-------------------------------------------------------------------------------
positif
  "composantes - renvoie la valeur de Positif."
  ^Positif
-----------------------------------------------
sud
  "orientation - renvoie l'orientation Sud."
  ^Sud
-----------------------------------------------
testerOrientation: dirSens
  "prive - teste si dirSens est une orientation correcte. Renvoie
   true si oui, false sinon."
  ^(dirSens isMemberOf: SmallInteger) and:
   [dirSens between: 1 and: 4]
-------------------------------------------------------------------
```

```
vertical
  "composantes - renvoie la valeur de Vertical."
  ^Vertical
-----------------------------------------------


ComposantRectangle methods:
ajouterVoisin: unComposant pourOrientation: orientation
  "voisinagePrive - ajoute au voisinage(D,S) du receveur le composant
   unComposant. Ce message ne peut etre envoye que par un composant
   ayant detecte le receveur comme nouveau voisin; comme la
   relation est symetrique, unComposant devient voisin du receveur."
  (voisins at: orientation) add: unComposant.
-------------------------------------------------------------------
bas
  "renvoisGeometriques - Renvoie l'ordonnee basse du rectangle
   associe au composant."
  ^enveloppe bottom
-------------------------------------------------------------------
bord: direction
  "renvoisGeometriques - Renvoie sous forme d'un point le bord qui
   correspond a la direction donnee. C'est a dire le segment vertical
   ou horizontal decrivant le cote 'direction' de l'enveloppe du
   receveur.
    i.e: bord horizontal du receveur si direction = Horizontal.
   Renvoie nil si direction n'a pas de sens."

  (direction = Horizontal)
  ifTrue:  [^(enveloppe left @ enveloppe right)].
  (direction = Vertical)
  ifTrue: [^(enveloppe top @ enveloppe bottom)].
  ^nil
-------------------------------------------------------------------
bordAnti: direction
  "renvoisGeometriques - Renvoie sous forme d'un point le bord qui
   correspond a la direction opposee a celle donnee. C'est a dire
   le segment vertical ou horizontal decrivant le cote 'direction
   opposee' de l'enveloppe du receveur.
    i.e: bord vertical du receveur si direction = Horizontal.
   Renvoie nil si direction n'a pas de sens."

  (direction = Horizontal)
  ifTrue:  [^(enveloppe top @ enveloppe bottom)].
  (direction = Vertical)
  ifTrue: [^(enveloppe left @ enveloppe right)].
  ^nil
-------------------------------------------------------------------
```

```
cherchezApparitionsDuesA: unComposant
  disparition: boolDisparition
  orientation: uneOrientation
  zoneLiberee: intervalle
  voisinageAScruter: vDSComposant

  "apparitionsDisparitions - Le receveur execute l'algorithme d'apparition
   de nouveaux voisins de vDSComposant."

  |i                    "un indice de parcours."
   d s                  "composantes de orientation."
   antiS                "le sens oppose a s."
   v                    "un composant temporaire."
   recouvrement         "la portion de bord anti d de self masquee par
                         ses voisins."
   zoneVisible          "la portion de bord anti d de self non masquee
                         par ses voisins jusqu'a unComposant."
   intersection         "entre zoneVisible et bord anti d de unComposant."
   vDS                  "les voisins de self pour orientation."
  |

  vDS := voisins at: uneOrientation.

  vDSComposant isEmpty
  ifTrue:
    "CAS DU VOISINAGE A SCRUTER VIDE."
    [boolDisparition
     ifTrue: "j'ote la reference a unComposant dans mon voisinage."
       [vDS
         remove: (vDS at: (vDS findFirst: [:c | unComposant nom = c nom]) ) ]
    ]
  ifFalse:
    "CAS NORMAUX."
    [d := (ComposantRectangle pointCardinalPour: uneOrientation) x.
     s := (ComposantRectangle pointCardinalPour: uneOrientation) y.
     antiS := "le sens oppose pour l'uneOrientation."
       Positif + Negatif - s.
     recouvrement := "la partie du receveur masquee."
       UniondIntervalles new:0.
     i := 1.

    "RECOUVREMENT DE SELF JUSQU'A unComposant."
    "C'est l'union des parties masquees du receveur par tous ses
     voisins qui se trouvent avant unComposant."
    [(i <= vDS size) and:
     [v := vDS at: i.
      v inferieurA: unComposant dansOrientation: uneOrientation]
    ]
    whileTrue:
      [recouvrement := recouvrement ++ (v interDir: d par: self).
       i := i + 1 ].

    "DETERMINER LA ZONE VISIBLE DU RECEVEUR."
    "C'est la partie reellement visible au receveur jusqu'a et
     y compris unComposant."
    zoneVisible :=
    (recouvrement ++ intervalle "une partie de la zone Liberee
                                 peut etre absorbee par  le recouvrement"
     \ recouvrement)            "on obtient la partie reellement liberee
                                 par unComposant."
    ** (self bordAnti: d).      "On en garde que l'intersection avec self."

    "MODIFICATION OU DESTRUCTION EVENTUELLE DE unComposant COMME VOISIN."
    "si disparition de unComposant, detruire sa reference dans vDS,
     sinon detruire la reference a l'ancien et la remplacer par le
     nouveau."
     i := (vDS findFirst: [:c| c nom = unComposant nom ]).
     vDS remove: (vDS at: i).
     boolDisparition ifFalse: [vDS add: unComposant].

    "REGARDER SI DE NOUVEAUX COMPOSANTS APPARAISSENT."
    "un composant de vDSComposant devient voisin s'il est visible de self"
    vDSComposant do: [:v1 |
      intersection := "entre v1 et la zoneVisible."
        zoneVisible ** (v1 bordAnti: d).
      (intersection notEmpty and:
      [(recouvrement contientUnion: intersection) not])
      ifTrue: "ce composant est en partie 'visible'."
        [(vDS includes: v1) not
         ifTrue: "c'est un nouveau voisin."
           [vDS add: v1.
            "et reciproquement self est un nouveau voisin de v1!"
            v1 ajouterVoisin: self pourOrientation: d + antiS
           ].
```

```
        "mettre a jour la portion masquee de self."
        recouvrement := recouvrement + intersection.
       ].
     ]
   ].

 voisins
   at: uneOrientation
   put: (vDS asSortedCollection:
              (SortBlocks at: uneOrientation) ).
```

```
cherchezDisparitionsDuesA: unComposant
  orientation: uneOrientation
  intersection: intervalle
  "apparitionsDisparitions - le receveur execute l'algorithme de disparition
   de voisins."

  |i                  "un indice de parcours."
   d s                "composantes de uneOrientation."
   antiS antiD        "les sens et directions opposes a s et d."
   v                  "un composant temporairement reference."
   deplacement        "booleen de detection si unComposant est deplace
                       ou cree."
   recouvrement       "la portion de bord d de self masquee par
                       ses voisins."
   intersection       "entre bords d de v et self."
   vDS                "les voisins de self pour uneOrientation."
  |

  vDS := voisins at: uneOrientation.

  vDS isEmpty
   ifTrue:
     "VOISINAGE DU RECEVEUR VIDE."
     [vDS add: unComposant. "On ajoute simplement ce composant." ]
   ifFalse:
     "CAS NORMAUX."
     ["serie d'initialisations."
      recouvrement := UniondIntervalles new:0.
      d := (ComposantRectangle pointCardinalPour: uneOrientation) x.
      s := (ComposantRectangle pointCardinalPour: uneOrientation) y.
      antiD := Horizontal + Vertical - d.
      antiS := Positif    + Negatif  - s.
      i := 1.

      "Transcript cr; show: 'chDispar recouv jusq self'. "
      "RECOUVREMENT DE SELF JUSQU'A unComposant."
      "C'est l'union des parties masquees du receveur par tous ses
       voisins qui se trouvent avant unComposant."
      [(i <= vDS size) and:
       [v := vDS at: i.
        v inferieurA: unComposant dansOrientation: uneOrientation] ]
      whileTrue:
        [recouvrement := recouvrement ++ (v interDir: d par:self).
         i := i + 1 ].

     " Transcript cr; show: 'chDispar recouv zone antiD'."
      "RECOUVREMENT DE self POUR LA ZONE antiD DE unComposant."
      "C'est le recouvrement du bord d de self par tous ses voisins
       qui sont moins proche que unComposant et qui sont en recouvrement
       antiD avec unComposant."
      deplacement := false.
      [(i <= vDS size) and:
       [v:= vDS at: i.
        v estIntersectePar: unComposant pourDirection: antiD ]
      ]
      whileTrue:
        [(v nom = unComposant nom)
         ifTrue: "c'est un deplacement antiD; remplacer v par unComposant a
                  sa nouvelle position."
           [deplacement := true.
            vDS remove: v; add: unComposant.
            v:= unComposant ].
```

```
      "mettre a jour le recouvrement."
      recouvrement :=
        recouvrement ++ (v interDir: d par: self).

      i := i + 1
     ].

   "unComposant ne faisait pas partie des voisins de self, donc
    c'est une creation de composant, et il faut ajouter unComposant
    a la liste des voisins de self, et modifier le recouvrement
    correspondant."
   deplacement
   ifFalse:
     [vDS add: unComposant.
      recouvrement :=
        recouvrement ++ (unComposant interDir: d par: self).
      i := i + 1. "puisque unComposant a ete insere parmi le
                   voisins de la boucle precedente."
     ].

" Transcript cr; show: 'chDispar suppression de voisins'. "
 "SUPPRESSION DE VOISINS DISPARUS."
```

```
      "A ce niveau, les seuls voisins restant a parcourir dans vDS
       sont forcement situes au dela de unComposant, et il peuvent
       donc disparaitre du vDS de self si unComposant les masque a
       self. On pourra arreter le processus aussi quand self est
       completement masque; alors tous les voisins restant seront
       invisibles."
      [(i <= vDS size) and:
       [(recouvrement contient: (self bordAnti: d)) not]
      ]
      whileTrue:
        [v := vDS at: i. "voisin suivant."
         intersection := v interDir: d par: self.  "ce qu'il masque de self."

         (recouvrement contient: intersection)
         ifTrue: "le voisin est devenu invisible, on le supprime."
           ["mais self est aussi invisible pour le voisin."
            (vDS at: i)
              detruireVoisin: self
              pourOrientation: d + antiS.
            vDS remove: (vDS at: i)
            "Note : pour tenir compte de cette destruction, on ne modifie
             pas l'indice de parcours de vDS."
           ]
         ifFalse: "le voisin est toujours visible, on le garde."
           [recouvrement := recouvrement ++ intersection.
            i := i + 1 ]
        ].

      "Si tout le bord d de self est masque, plus aucun composant ne
       peut etre visible; on supprime les restants."
      [i <= vDS size]
      whileTrue:
        [(vDS at: i)
           detruireVoisin: self
           pourOrientation: d + antiS.
         vDS remove: (vDS at: i)
        ].

     ].

   "REORDONNER vDS, MAIS EST CE UTILE ? (INSTANCE DE SORTEDCOLLECTION)."
   voisins
     at:  uneOrientation
     put: (vDS
             asSortedCollection:
               (SortBlocks at: uneOrientation)
          ).
```

```
deplacementVers: orientation de: uneDistance
  "actionsSurComposant - Le receveur execute l'algorithme de deplacement:
    Calcul de sa nouvelle position (=> mise a jour attributs).
    Creation d'un nouveau composant avec ces attributs.
    On ote temporairement le composant de la base.
    On recalcule le voisinage du composant uniquement dans les
     orientations interesssantes (celles opposees au deplacement).
    Decomposition du voisinage en 3 sous ensembles:
      anciensVoisins, nouveauxVoisins, voisinsIdentiques.
    Envois de messages de mise a jour des contextes des voisins:
      apparitions possibles pour anciensVoisins;
      disparitions possibles pour nouveauxVoisins;
      apparitions-disparitions possibles pour voisinsIdentiques.
    Pour les composants voisins du receveur a sa nouvelle position,
    mise a jour de leur voisinage au niveau du composant reference
    (le receveur a sa nouvelle position)."

  |d antiD s antiS   "des directions et sens deduits de orientation."
   translation       "le deplacement a appliquer au composant."
   ds                "une orientation deduite d'un sens et d'une direction."
   antiS2            "localement un sens oppose."
   composant         "le resultant du deplacement."
   zoneLiberee       "zone liberee par le deplacement du composant."
   anciensVoisins    "voisins du composant avant deplacement."
   nouveauxVoisins   "voisins du composant apres deplacement."
   voisinsIdentiques
   voisinsApparus
   voisinsDisparus   "les 3 sous ensembles qui forment la decomposition
                      des anciens et nouveaux voisins."
  |

   "DECOMPOSITION DE L'ORIENTATION EN SES COMPOSANTES."
   d := (ComposantRectangle pointCardinalPour: orientation) x.
   s := (ComposantRectangle pointCardinalPour: orientation) y.
   antiD := Horizontal + Vertical - d.
   antiS := Positif + Negatif - s.

   "CREATION D'UN COMPOSANT AYANT LES NOUVEAUX ATTRIBUTS DE SELF."
   translation :=
     [d = Horizontal
      ifTrue:
        [s = Positif
         ifTrue: [uneDistance @ 0]
         ifFalse: [uneDistance negated @ 0]
        ]
      ifFalse:
        [s = Positif
         ifTrue: [0 @ uneDistance negated]
         ifFalse: [0 @ uneDistance]
        ]
     ] value.
   composant := ComposantRectangle new.
   composant
     nom: nom
     enveloppe: (enveloppe translateBy: translation)
     projet: projet.

   "CALCUL DE LA ZONE LIBEREE PAR LE DEPLACEMENT DU COMPOSANT."
   zoneLiberee :=
     s = Positif
     ifTrue:
       [d = Horizontal
        ifTrue:  [enveloppe left @ composant gauche]
        ifFalse: [composant bas  @ enveloppe bottom]
       ]
     ifFalse:
       [d = Horizontal
        ifTrue:  [composant droit @ enveloppe right]
        ifFalse: [enveloppe top   @ composant haut]
       ].

   "OTER TEMPORAIREMENT  LE COMPOSANT DU PROJET."
   projet oterComposant: self.

   "RECALCULER LE VOISINAGE DU COMPOSANT DEPLACE."
   anciensVoisins := voisins.
   "affectation directe du voisinage, en particulier les inchanges."
   composant     "recalcul du voisinage qui a pu etre modifie."
     determinerVoisinsPourDirection: antiD.
   "self halt."
   Positif to: Negatif do: [:sens|
     (composant voisins)
       at: d + sens
       put: (anciensVoisins at: d + sens)
     ].
```

```
    nouveauxVoisins := composant voisins. "nouveau contexte"
    "self halt."

    "DECOMPOSITION DES ANCIENS ET NOUVEAU VOISINAGE EN 3 SOUS - ENSEMBLES."
    "ces sous-ensembles sont des tableaux de voisisnage a 2 elements,
     car lors d'un deplacement, seuls les voisins dans la direction
     antiD sont modifies."
     voisinsIdentiques := Array new:2.
     voisinsDisparus   := Array new:2.
     voisinsApparus    := Array new:2.

    "On remplit ces voisinages."
    Positif to:Negatif do: [:sens1 |
      voisinsIdentiques at: sens1 put: OrderedCollection new.
      voisinsApparus    at: sens1 put: OrderedCollection new.
      voisinsDisparus   at: sens1 put: OrderedCollection new.

      ds := antiD + sens1.

      "remplit voisinsApparus et voisinsIdentiques."
      (nouveauxVoisins at:ds) do: [:compN |
        ((anciensVoisins at:ds) includes: compN)
        ifTrue:  [(voisinsIdentiques at: sens1) add: compN]
        ifFalse: [(voisinsApparus    at: sens1) add: compN]
        ].

      "remplit voisinsDisparus."
      (anciensVoisins at:ds) do: [:compA|
        ((nouveauxVoisins at:ds) includes: compA)
        ifFalse: [(voisinsDisparus at: sens1) add: compA]
        ].
      ]. "les 3 sous-ensembles sont initialises."

    "self halt."
    "faire les traitements sur voisinage pour les 2 sens."
     Positif to: Negatif do: [:sens2|
       antiS2 := Positif + Negatif - sens2.

       "TRAITER LES VOISINS DISPARUS."
       (voisinsDisparus at: sens2) do: [:vDisparu|
         vDisparu
           cherchezApparitionsDuesA: composant
           disparition:              true   "le composant est disparu pour les anciens vo
isins"
           orientation:              antiD + antiS2
           zoneLiberee:              zoneLiberee
           voisinageAScruter:        (anciensVoisins at: (antiD + antiS2)).
         ].

       "TRAITER LES VOISINS APPARUS."
       (voisinsApparus at: sens2) do: [:vApparu|
         vApparu
           cherchezDisparitionsDuesA: composant
           orientation:               antiD + antiS2
           intersection:              (composant interDir: antiD par: vApparu)
         ].

       "TRAITER LES VOISINS IDENTIQUES."
       (voisinsIdentiques at: sens2) do: [:vIdentique|
         vIdentique
           cherchezApparitionsDuesA: composant
           disparition:              false  "composant reste visible a vIdentique"
           orientation:              antiD + antiS2
           voisinageAScruter:
             (anciensVoisins at: (antiD + antiS2));

           cherchezDisparitionsDuesA: composant
           orientation:               antiD + antiS2
           intersection:              (composant interDir: antiD par: vIdentique).
       ]
     ].

"POUR LES COMPOSANTS VOISINS DU RECEVEUR A SA NOUVELLE POSITION,
 MISE A JOUR DE LEUR VOISINAGE AU NIVEAU DU COMPOSANT REFERENCE
 (REFERENCER LE RECEVEUR A SA NOUVELLE POSITION)."
Positif to: Negatif do: [:sens3 |
  (nouveauxVoisins at: d + sens3) do: [:comp|
    comp
      remplacerVoisin:  self
      par:              composant
      pourOrientation: d + (Positif + Negatif - sens3)
    ]
  ].

"REINSERER LE COMPOSANT DANS LA BASE DU PROJET."
```

```
   projet ajoutComposant: composant.
-------------------------------------------------------------------------------
-------
destruction
  "actionsSurComposant - Realise l'algorithme de destruction
   d'un composant. Les voisins sont avertis et executent
   l'algorithme de disparition de voisins."
  | direction sensInverse|

  Est to: Sud do: [:ds |
    "Initialiser les sens et directions pour designer mon orientation
     pour mes voisins."
    direction := (ComposantRectangle pointCardinalPour: ds) x.
    sensInverse :=
      "astuce de calcul pour obtenir le sensInverse."
      Positif + Negatif - (ComposantRectangle pointCardinalPour: ds) y.

    "le receveur avertit ses voisins ds qu'il est detruit
     et les 'invitent' a effectuer l'algorithme d'apparition
     de voisins."
    (voisins at: ds) do: [:voisinDS |
      voisinDS
        cherchezApparitionsDuesA: self
        disparition: true
        orientation: (direction + sensInverse)
        zoneLiberee:
          (voisinDS interDir: direction par: self)
        voisinageAScruter:
          (voisins at: direction + sensInverse)
      ]
    ]
-------------------------------------------------------------------------------
```

```
determinerVoisins
  "constructionVoisins - determine et affecte le voisinage du receveur.
   On calcule d'abord les intersectants du receveur sur chaque direction,
   puis on en extrait les voisins."

  |i                  "un indice."
   d                  "une direction temporaire."
   bordAntiD          "le bord anti d de self."
   intersection       "intersection temporaire entre self et un composant."
   interOrientation   "intersectants, puis voisins pour une orientation."
   intersectants      "intersectants, puis voisins de self."
   partieCachee       "ce que les voisins cachent de self. temporaire."
   ci                 "un composant choisi temporairement."
  |

  "DETERMINATION DES INTERSECTANTS."
  intersectants := self intersectants.

  "pour chaque orientation, traiter le probleme."
  Est to: Sud do: [:orientation |
    "INITIALISATION DES VARIABLES TEMPORAIRES."
    i := 1.
    d := (ComposantRectangle pointCardinalPour: orientation) x.
    bordAntiD := self bordAnti: d.
    interOrientation := intersectants at: orientation.
    partieCachee := UniondIntervalles new: 0.

   "SELECTION DES VOISINS."
   "le principe est le suivant: tant que le bordAntiD n'est pas
    completement masque par les voisins choisis au fur et a mesure
    du parcours de interOrientation, determiner pour le composant
    intersectant choisi s'il est 'visible' de self, si oui c'est un
    voisin de self pour orientation, et en deduire la nouvelle partie
    cachee de self."
    [(i <= interOrientation size) and:
     [(partieCachee contient: bordAntiD) not] ]
    whileTrue:
      [ci := interOrientation at: i. "l'intersectant suivant."
       intersection :=  "la zone en intersection avec self."
         ci interDir: d par: self.

       "la partie de controle; on agit directement sur interOrientation."
       (partieCachee contient: intersection)
       ifTrue: "le composant est invisible de self, l'oter des intersectants."
         [interOrientation remove: ci ]
       ifFalse: "le composant est visible, modifier  la partie cachee de self."
         [partieCachee := partieCachee ++ intersection.
          i := i + 1 ].
      ].

    "s'il reste des composants non testes, ils sont invisibles
     car le bordAntiD de self est completement masque par les voisins.
     On retire de interOrientation ces voisins."
    [(i <= interOrientation size)]
    whileTrue:
      [interOrientation remove: (interOrientation at: i)].

    "REORDONNER CORRECTEMENT LES VOISINS."
    intersectants
      at: orientation
      put: (interOrientation
              asSortedCollection:
                (SortBlocks at: orientation) )
```

```
    ].

  voisins := intersectants.
-------------------------------------------------------------------------------
```

```
determinerVoisinsPourDirection: uneDirection
  "constructionVoisins - determine et affecte le voisinage du receveur
   pour la direction uneDirection. On calcule d'abord les intersectants
   du receveur pour cette direction, puis on en extrait les voisins."

  |i                  "un indice."
   bordAntiD          "le bord anti d de self."
   intersection       "intersection temporaire entre self et un composant."
   interOrientation   "intersectants, puis voisins pour une orientation."
   intersectants      "intersectants, puis voisins de self."
   partieCachee       "ce que les voisins cachent de self. temporaire."
   ci                 "un composant choisi temporairement."
  |


  "DETERMINATION DES INTERSECTANTS."
  intersectants := self intersectantsPour: uneDirection.

  Positif to: Negatif do: [:sens |
    "INITIALISATION DES VARIABLES TEMPORAIRES."
    i := 1.
    bordAntiD := self bordAnti: uneDirection.
    interOrientation := intersectants at: sens.
    partieCachee := UniondIntervalles new:0.

    "SELECTION DES VOISINS."
    "le principe est le suivant: tant que le bordAntiD n'est pas
     completement masque par les voisins choisis au fur et a mesure
     du parcours de interOrientation, determiner pour le composant
     intersectant choisi s'il est 'visible' de self, si oui c'est un
     voisin de self pour orientation, et en deduire la nouvelle partie
     cachee de self."
    [(i <= interOrientation size) and:
     [(partieCachee contient: bordAntiD) not] ]
    whileTrue:
      [ci := interOrientation at: i.   "l'intersectant suivant."
       intersection := "son recouvrement avec self."
         ci interDir: uneDirection par: self.

       "on travaille directement sur les intersectants."
       (partieCachee contient: intersection)
       ifTrue: "l'intersectant est invisible, l'oter de interOrientation."
         [interOrientation remove: ci]
       ifFalse: "c'est un voisin, modifier partie cachee de self."
         [partieCachee := partieCachee ++ intersection.
          i:= i+ 1 ]
      ].

    "s'il reste des composants non testes, ils sont invisibles
     car le bordAntiD de self est completement masque par les voisins.
     On retire de interOrientation ces voisins."
    [(i <= interOrientation size)]
    whileTrue:
      [interOrientation remove: (interOrientation at: i)].

    "REORDONNER LES VOISINS ET MODIFIER LE VOISINAGE DU COMPOSANT."
    voisins
      at:  (uneDirection + sens)
      put: ( interOrientation
             asSortedCollection:
               (SortBlocks at: (uneDirection + sens) ) )
```

```
-------------------------------------------------------------------------
detruireVoisin: unComposant pourOrientation: orientation
  "voisinagePrive - retire du voisinage(D,S) du receveur le composant
   unComposant. Ce message ne peut etre envoye que par un composant
   ayant detecte la disparition du receveur comme voisin; la relation
   etant symetrique, unComposant n,'est plus voisin du receveur."
  (voisins at: orientation) remove: unComposant.
-------------------------------------------------------------------------
droit
  "renvoisGeometriques - Renvoie l'abscisse droite du rectangle
   associe au composant."
  ^enveloppe right
-------------------------------------------------------------------------
enveloppe
  "renvoisGeometriques - Renvoie le rectangle associe au composant."
  ^enveloppe
-------------------------------------------------------------------------
estIntersectePar: unComposant pourDirection: direction
 "predicatsTopologiques - Teste s'il y a intersection verticale
  ou horizontale entre le receveur et unComposant.
   i.e: Id(receveur, argument) ?<> 0.
  Renvoie true si il y a intersection, false sinon."
 ^(self interDir: direction par: unComposant) notNil
-------------------------------------------------------------------------
gauche
  "renvoisGeometriques - Renvoie l'abscisse gauche du rectangle
   associe au composant."
  ^enveloppe left
-------------------------------------------------------------------------
haut
  "renvoisGeometriques - Renvoie l'ordonee haute du rectangle
   associe au composant."
  ^enveloppe top
-------------------------------------------------------------------------
inferieurA: composant dansOrientation: orientation
  "predicatsTopologiques - Teste si le receveur precede ou non le receveur
   dans orientation."
  (orientation = Est)   ifTrue: [^enveloppe left   < composant gauche].
  (orientation = Ouest) ifTrue: [^enveloppe right  > composant droit].
  (orientation = Nord)  ifTrue: [^enveloppe bottom > composant bas].
  (orientation = Sud)   ifTrue: [^enveloppe top    < composant haut].
-------------------------------------------------------------------------
interDir: direction par: unComposant
  "calculRecouvrement - Renvoie l'intervalle de recouvrement entre
   unComposant et le receveur suivant direction:
    i.e: Id(receveur, argument).
   Renvoie nil si recouvrement vide.
   Attention:
    le systeme de coordonnees de Smalltalk en y est oriente vers le bas.
    L'orientation Nord etant celle de sens positif, les inegalites
    ci-dessous sont correctes car dans ce systeme de coordonnees, pour
    les instances de Rectangle, on a top < bottom."

 (direction = Horizontal)
 ifTrue:
   ["existe t'il un recouvrement?"
    ((enveloppe top > unComposant bas) or:
     [enveloppe bottom < unComposant haut])
    ifTrue: ["non !" ^nil]
    ifFalse:
      ["oui, le recouvrement est un segment vertical. On renvoie
        l'intervalle le decrivant avec en x son ordonnee basse,
        en y son ordonnee haute ."
       ^(enveloppe top max: unComposant haut) @
        (enveloppe bottom min: unComposant bas)]
   ]
 ifFalse:
   [((enveloppe left > unComposant droit) or:
     [enveloppe right < unComposant gauche])
    ifTrue: ["non !" ^nil]
    ifFalse:
      ["oui, le recouvrement est un segment horizontal. On renvoie
       l'intervalle le decrivant avec en x son abscisse gauche,
       en y son abscisse droite."
       ^(enveloppe left max: unComposant gauche) @
        (enveloppe right min: unComposant droit)]
   ]
-------------------------------------------------------------------------
```

```
intersectants
  "constructionVoisins prive - Determine, pour un composant donne
   (le receveur), les composants du projet qui sont en intersection
   horizontale ou verticale avec ce composant. C'est une etape
   intermediaire a la determination de voisins.
   Pour chaque composant du projet hormis le receveur, regarde s'il
   est en recouvrement avec le receveur dans l'une des 4 orientations.
   si Oui, classe ce composant l'ensemble d'intersectants correspondant.
   Renvoie dans un tableau indice par les orientations, les intersectants
   verticaux et horizontaux du receveur."
  |interOrientation      "le tableau des 4 ensembles d'intersectants."
   interEst             "les intersectants a l'est."
   interOuest           "les intersectants a l'ouest."
   interNord            "les intersectants au nord."
   interSud             "les intersectants au sud."
  |

  "INITIALISATIONS."
  interOrientation := Array new: 4.
  interOrientation at: Est   put: (interEst   := OrderedCollection new).
  interOrientation at: Ouest put: (interOuest := OrderedCollection new).
  interOrientation at: Nord  put: (interNord  := OrderedCollection new).
  interOrientation at: Sud   put: (interSud   := OrderedCollection new).

  "DETERMINATION DES INTERSECTIONS."
  projet do: [:composant|
   (self estIntersectePar: composant pourDirection: Vertical)
    ifTrue: "cas d'intersection verticale."
     [(composant situeAu: Nord de: self)
      ifTrue:  [interNord  add: composant]
      ifFalse: [interSud add: composant]
     ]
    ifFalse: "cas d'intersection horizontale."
     [(self estIntersectePar: composant pourDirection: Horizontal)
      ifTrue:
       [(composant situeAu: Est de: self)
        ifTrue:  [interEst  add: composant]
        ifFalse: [interOuest add: composant]
       ]
     ]
   ].

  "ORDONNER CORRECTEMENT LES INTERSECTIONS."
  Est to: Sud do: [:orientation |
    interOrientation
      at: orientation
      put:
        ( (interOrientation at:orientation)
            asSortedCollection:
              (SortBlocks at: orientation) )
    ].

  ^interOrientation
-------------------------------------------------------------------------
```

```
intersectantsPour: uneDirection
  "constructionVoisins prive - Calcule, pour la direction horizontale
   ou la direction verticale, les 2 ensembles de composants du projet
   en intersection avec le receveur. Renvoie ces ensembles dans un
   tableau indice par les sens sur cette direction."
  |interD          "le tableau des intersectants pour uneDirection"
   interDPlus      "les intersectants de sens Positif."
   interDMoins     "les intersectants de sens Negatif."
  |

  "INITIALISATIONS."
  interD := Array new: 2. "seuls les sens Positif et Negatif comptent."
  interD at: Positif put: (interDPlus  := OrderedCollection new).
  interD at: Negatif put: (interDMoins := OrderedCollection new).

  "DETERMINATIONS ET AFFECTATIONS DES INTERSECTANTS SUIVANT LE SENS."
  projet do: [:composant|
    (self estIntersectePar: composant pourDirection: uneDirection)
    ifTrue:
      [(composant situeAu: (uneDirection + Positif) de: self)
       ifTrue:  [interDPlus  add: composant]
       ifFalse: [interDMoins add: composant]
      ]
    ].

  "ON REORDONNE CORRECTEMENT interD."
  Positif to: Negatif do: [:sens |
    interD
      at:  sens
      put:
        ( (interD at: sens)
            asSortedCollection:
              (SortBlocks at: (uneDirection + sens) )  )
    ].

  "RENVOI DES INTERSECTANTS TRIES."
  ^interD
-------------------------------------------------------------------------
nom
  "caracteristiques - Renvoie le nom du composant."
  ^nom
-------------------------------------------------------
nom: unNom enveloppe: unRectangle projet: unProjet
  "caracteristiques - modifie directement les attributs du receveur."
  nom := unNom.
  enveloppe := unRectangle.
  projet:= unProjet.
  voisins := Array new: 4.
-------------------------------------------------------------------------
placement
  "actionsSurComposant - Realise l'algorithme de placement pour le
   receveur: determination du voisinage du receveur parmi la base;
   puis modification du contexte des voisins (disparitions possibles)."
  | direction sensInverse |

  "calcul du voisinage du receveur."
  self determinerVoisins.

  "modification du contexte des voisins."
  Est to:Sud do: [:ds|
    "Initialiser les sens et directions pour designer mon orientation
     pour mes voisins."
    direction := (ComposantRectangle pointCardinalPour: ds) x.
```

```
    sensInverse :=
      "astuce de calcul pour obtenir le sensInverse."
      Positif + Negatif - (ComposantRectangle pointCardinalPour: ds) y.

    "envoyer le message de modif de contexte a tous
     les voisins (disparitions possibles)."
    (voisins at: ds) do: [:voisinDS |
      voisinDS
        cherchezDisparitionsDuesA: self
        orientation: (direction + sensInverse)
        intersection: (self interDir: direction par: voisinDS).
      ]
    ]
```

```
remplacerVoisin: ancienEtatDeVoisin
 par: nouvelEtatDeVoisin
 pourOrientation: orientation
 "voisinagePrive - remplace les anciennes caracteristiques
  d'un voisin du receveur par les nouvelles. Ne doit etre envoye que par
  un composant qui a ete deplace sans conflits, et dont le receveur est
  reste voisin. Utile car l'algorithme de deplacement de composant
  cree une nouvelle instance de ComposantRectangle ayant les nouvelles
  caracteristiques de l'envoyeur."
 (voisins at: orientation)
   remove: ancienEtatDeVoisin;
   add:    nouvelEtatDeVoisin.
```

```
showCaracteristiques
  "caracteristiques - Renvoie la description texte des attributs
   du receveur dans une chaine de caracteres destinee a etre affichee."
  | text|
  "creation du texte avec le nom du composant."
  text := WriteStream with: nom, ' '.

  "description de l'enveloppe."
  text
    nextPutAll:
      ' enveloppe: [', enveloppe origin printString,
      '] x [', enveloppe corner printString,
      ']'; cr.

  "description du voisinage."
  Est to: Sud do: [:ds|
    text
      nextPutAll:
        'voisins ',(ComposantRectangle chainePour: ds),': '.
    (voisins at: ds) do: [:c| text nextPutAll: c nom, ','].
    text cr].

  "renvoi de la chaine descriptive."
  ^text contents
```

```
situeAu: orientation de: unComposant
  "predicatsTopologiques - Teste si le receveur est place suivant
   orientation par rapport a unComposant. Si orientation n'a pas
   de sens, renvoie nil."

  "self a l'est de unComposant ?"
  (orientation = Est)
  ifTrue: [^(enveloppe left > unComposant droit)].

  "self a l'ouest de unComposant ?"
  (orientation = Ouest)
  ifTrue: [^(enveloppe right < unComposant gauche)].

  "self au nord de unComposant ?"
  (orientation = Nord)
  ifTrue: [^(enveloppe bottom < "(>)" unComposant haut)].

  "self au sud de unComposant ?"
  (orientation = Sud)
  ifTrue: [^(enveloppe top > "(<)" unComposant bas)].

  ^nil
```

```
voisins
  "voisinagePrive - renvoie le voisinage du receveur."
```

```
  ^voisins
-------------------------------------------------------
voisins: unVoisinage
  "voisinagePrive - affecte directement le voisinage du receveur."
  voisins := unVoisinage.
-------------------------------------------------------------------
```

```
Object subclass: #ProjetComposant
  instanceVariableNames:
    'baseProjet nom image pasDeGrille composantSelectionne commandeSelectionnee graphPan
e flotDeRenvois '
  classVariableNames:
    'PoliceDessin '
  poolDictionaries:
    'FunctionKeys '
ProjetComposant class methods:
deNom: uneChaine
 "Cree un nouveau projet de nom uneChaine "
 ^self new nom: uneChaine
-----------------------------------------
new
 "Cree une nouvelle base de donnee projet "
 | temporaire |
 PoliceDessin isNil ifTrue: [PoliceDessin := Font eightLine].
 ComposantRectangle initialiser.
 temporaire := super new.
 temporaire baseDuProjet: Dictionary new.
 temporaire pasDeGrille: 10.
 ^temporaire
-----------------------------------------------------------------


ProjetComposant methods:
ajoutComposant: unComposant
  "gestionBaseProjet - ajoute un composant a la base du projet."
  baseProjet add:
    (Association
      key: unComposant nom
      value: unComposant)
-----------------------------------------------------------------
baseDuProjet
 " Renvoit la base de donnee du projet en cours (receveur)."
 ^baseProjet
-----------------------------------------------------------------
baseDuProjet: dico
 " Affecte le dictionnaire du projet a dico."
 baseProjet := dico.
-----------------------------------------
choisirComposant
 " permet a l'utilisateur de selectionner un composant par son nom."
 | name noms i|
 i := 0.
 noms := Array new: self baseDuProjet size.
 self baseDuProjet keysDo:
 [:clef| noms at: (i := i + 1) put: clef].
 (name :=
  (Menu
    labelArray: noms
    lines: Array new
    selectors: noms) popUpAt: Cursor offset)
 notNil ifTrue:
  [composantSelectionne := self baseDuProjet at: name.
   self changed: #renvois.
   self changed: #commandes.
  ].
-------------------------------------------------------------------------
```

```
coherenceBase
 " verifie que le graphe des relations entre composant est
  complet."
 | incoherents tabDirOpposees |
 incoherents:= (OrderedCollection with: 'incoherence(s) pour:').

 tabDirOpposees :=
  (Array new: 4)
   at: ComposantRectangle est put: ComposantRectangle ouest;
   at: ComposantRectangle ouest put: ComposantRectangle est;
   at: ComposantRectangle nord put: ComposantRectangle sud;
   at: ComposantRectangle sud put: ComposantRectangle nord;
   yourself.

self baseDuProjet do:
  [:c|
   ComposantRectangle est to: ComposantRectangle sud do:
    [:ds|
     (c voisins at: ds) do:
     [:v|
      ((v voisins at: (tabDirOpposees at: ds))
       includes: c)
      ifFalse:
       [incoherents last = c nom
        ifFalse: [incoherents add: c nom]
       ]
     ]
    ]
  ].

 (Menu
    labelArray: incoherents
    lines: #()
    selectors: incoherents)
  popUpAt: Cursor offset.
-------------------------------------------------------------------------
coinComposant
 " permet a l'utilisateur de modifier la taille du composant en
  changeant l'origine de celui-ci."
 | point rect|
 composantSelectionne isNil
 ifTrue: [Menu message:'pas de composant selectionne'.^nil].

 graphPane form: (self dessinDestructionDe: composantSelectionne).
 point := self getCoin: composantSelectionne enveloppe.
 rect := composantSelectionne enveloppe origin corner: point.
 (self
   verifierEnveloppe: rect
   nom: composantSelectionne nom )
 ifTrue:
  [self modifierTailleDe: composantSelectionne
        nouveauCoin: point.
  ].
 composantSelectionne :=
  self baseDuProjet at: composantSelectionne nom.
 graphPane form: (self dessinCreationDe: composantSelectionne).
 self changed: #renvois.
-------------------------------------------------------------------------
commande: aString
 " demande a l'utilisateur les arguments de la commande et fait
  executer cette commande par le receveur."

 (aString = 'creer')       ifTrue: [ self creerComposant.].
```

```
(aString = 'choisir')     ifTrue: [self choisirComposant.].
(aString = 'rafraichir') ifTrue: [self rafraichirEcran.].
(aString = 'grille')      ifTrue: [self grilleEcran.].

(aString = 'coin')        ifTrue: [self coinComposant].
(aString = 'enveloppe')   ifTrue: [self enveloppeComposant].

(aString = 'deplacer') ifTrue: [self deplacerComposant].
(aString = 'deporter') ifTrue: [self deporterComposant].
(aString = 'detruire') ifTrue: [self detruireComposant].
(aString = 'coherence') ifTrue: [self coherenceBase].
-------------------------------------------------------------
```

```
commandes
 " Renvoie la liste des commandes disponibles. "
 ^#('creer'
    'choisir'
    'rafraichir'
    'grille'
    'coherence'
    ' '
    'coin'
    'enveloppe'
    ' '
    'deplacer'
    'deporter'
    'detruire')
-------------------------------------------------
creerComposant
 " message graphique de creation de composant Rectangle."
 | select nomComposant nomPropose index rect ct|
 [select :=
   (Menu labels:'creer composant\echapper' withCrs
    lines: Array new
     selectors: #( Oui Non))
   popUpAt: Cursor offset.
  select = #Non]
 whileFalse:
  [index := 1.
   nomPropose := 'C00'.
   [self baseDuProjet includesKey: nomPropose]
   whileTrue:
    [nomPropose :='C',
      [(index < 10)
       ifTrue: ['0', index printString]
       ifFalse: [index printString].
      ] value.
     index := index + 1.
    ].

   nomComposant :=
    Prompter prompt:'nomduComposant' default: nomPropose.

   (self baseDuProjet includesKey: nomComposant)
   ifTrue:[Menu message:'composant deja existant'.]
   ifFalse:
    [rect := self getEnveloppe: ComposantRectangle new.
     (self verifierEnveloppe: rect nom: nomComposant)
     ifTrue:
      [composantSelectionne :=
       self creerComposantRectangle: nomComposant
           enveloppe: rect.
       composantSelectionne notNil ifTrue:
        [graphPane form:
         (self dessinCreationDe: composantSelectionne).
        ].
      ].
    ].
   self changed: #renvois.
  ].
 self changed: #commandes.
-------------------------------------------------------------
creerComposantRectangle: unNom enveloppe: unRectangle
 "Creation d'un nouveau ComposantRectangle associe au receveur.
  Les tests de validite du nouveau comosant sont fait a l'essai
  de creation dans la classe ComposantRectangle: nom correct,
```

```
  pas d'autre composant de meme nom, rectangle correct.
  Renvoie le composant cree."

 ^ComposantRectangle
   nom: unNom
   enveloppe: unRectangle
   projet: self
-----------------------------------------------------------------
```

```
deplacerComposant
 " permet a l'utilisateur de deplacer unComposant horizontalement
  ou verticalement."
 | rect oldEnveloppe dirSens ds |
 composantSelectionne isNil
 ifTrue: [Menu message: 'pas de composant selectionne'.^nil].

 dirSens :=
    (Menu
       labels: 'vers Est\vers Ouest\vers Nord\vers Sud' withCrs
       lines: Array new
       selectors: #( Est Ouest Nord Sud)
    ) popUpAt: Cursor offset.

 dirSens isNil
 ifTrue: [^nil]
 ifFalse:
  [dirSens := ComposantRectangle orientationPour: dirSens asString].

 ds := ComposantRectangle pointCardinalPour: dirSens.

 oldEnveloppe := composantSelectionne enveloppe.
 graphPane form: (self dessinDestructionDe: composantSelectionne).

 rect := self moveEnveloppe: oldEnveloppe toDirection: dirSens.
 (self verifierEnveloppe: rect nom: composantSelectionne nom)
 ifTrue:
  [self
   deplacerComposantRectangle: composantSelectionne
   direction: ds x
   sens: ds y
   distance:
    ((ds x == ComposantRectangle horizontal)
     ifTrue: [ (rect left - oldEnveloppe left) abs]
     ifFalse: [(rect top - oldEnveloppe top) abs]).
  ].
 composantSelectionne :=
  self baseDuProjet at: composantSelectionne nom.
 graphPane form: (self dessinCreationDe: composantSelectionne).
 self changed: #renvois.
-----------------------------------------------------------------
deplacerComposantRectangle: unComposant
 direction: uneDirection
 sens: unSens
 distance: uneDistance
 " Envoie a unComposant l'ordre de se deplacer dans la direction
  uneDirection, avec le sens unSens, et une grandeur de deplacement
  uneDistance. unComposant realise lui meme les mises a jours
  necessaires parmi les voisinages des composants."
 unComposant
  deplacementVers: (uneDirection + unSens)
  de: uneDistance
-----------------------------------------------------------------
deporterComposant
 " permet a l'utilisateur de deporter le composant selectionne a
  l'endroit de son choix dans la fenetre graphique."
 | rect oldEnveloppe |
 composantSelectionne isNil
 ifTrue: [Menu message: 'pas de composant selectionne'.^nil].

 oldEnveloppe := composantSelectionne enveloppe.
 graphPane form: (self dessinDestructionDe: composantSelectionne).
 rect := self moveEnveloppe: oldEnveloppe toDirection: nil.
```

```
(self verifierEnveloppe: rect nom: composantSelectionne nom)
ifTrue:
 [ self deporterComposantRectangle: composantSelectionne
     en: rect origin.
 ].
composantSelectionne :=
 self baseDuProjet at: composantSelectionne nom.
graphPane form: (self dessinCreationDe: composantSelectionne).
self changed: #renvois.
```

```
deporterComposantRectangle: unComposant
 en: unPoint
 " extirpe le composant de son contexte local pour le placer en unPoint."
 | rect |
 unComposant destruction.
 self baseDuProjet removeKey: unComposant nom.
 rect := unPoint
         corner:
          (unPoint +
           (unComposant enveloppe width @ unComposant enveloppe height)).
 ComposantRectangle
  nom: unComposant nom
  enveloppe: rect
  projet: self.
```

```
dessin: unFrame
 " initialise le contenu du dessin."
 | unPen unCharScan unePolice |
 (image isNil or:
  [(image width ~= unFrame width) or: [image height ~= unFrame height] ]
 )
 ifTrue:
  [image :=
    Form new
    width: unFrame width
    height: unFrame height
    initialByte: 16r00.
  ].

 unPen := Pen new: image.
 pasDeGrille <= 0
  ifFalse: [ unPen gray; grid: pasDeGrille.].
 unPen white.

 unCharScan :=
  (CharacterScanner new)
    initialize: (image offset extent: image extent)
    font: (unePolice := Font eightLine)
    dest: image.
 unCharScan
  mask: Form white;
  combinationRule: Form over.

 self baseDuProjet do:
  [:c |
   self drawRectangle: c enveloppe pen: unPen.
   unCharScan
    display: c nom
    from: 2
    to: 3
    at: (c enveloppe center x - unePolice width + 1) @
        (c enveloppe center y - (unePolice height // 2 - 1) ).
  ].
 image displayAt: unFrame origin.
 ^image
```

```
dessinCreationDe: unComposant
 " met a jour l'image apres creation de unComposant ."
 | unPen unCharScan unePolice |
 unPen := (Pen new: image) white.
 unCharScan :=
  (CharacterScanner new)
```

```
    font: (unePolice := Font eightLine)
    dest: image.
 unCharScan
  mask: Form white;
  combinationRule: Form over.
 unPen
  place: unComposant enveloppe origin;
  goto: unComposant droit @ unComposant haut ;
  goto: unComposant droit @ unComposant bas;
  goto: unComposant gauche @ unComposant bas;
  goto: unComposant enveloppe origin.
 unCharScan
  display: unComposant nom
  from: 2
  to: 3
  at: (unComposant enveloppe center x - unePolice width + 1) @
      (unComposant enveloppe center y - (unePolice height // 2 - 1)).

 image displayAt: graphPane frame origin.
 ^image
-------------------------------------------------------------------------
```

```
dessinDestructionDe: unComposant
 " met a jour l'image apres destruction de unComposant ."
 | unPen unCharScan unePolice |
 unPen := (Pen new: image) black.
 unCharScan :=
  (CharacterScanner new)
    initialize: (image offset extent: image extent)
    font: (unePolice := Font eightLine)
    dest: image.
 unCharScan
  mask: Form black;
  combinationRule: Form over.
 unPen
  place: unComposant enveloppe origin;
  goto: unComposant droit @ unComposant haut ;
  goto: unComposant droit @ unComposant bas;
  goto: unComposant gauche  @ unComposant bas;
  goto: unComposant enveloppe origin.
 unCharScan
  display: unComposant nom
  from: 2
  to: 3
  at: (unComposant enveloppe center x - unePolice width +1) @
      (unComposant enveloppe center y - (unePolice height // 2 - 1)).

 image displayAt: graphPane frame origin.
 ^image
-------------------------------------------------------------------------
detruireComposant
 " permet a l'utilisateur de detruire un composant de la base."
 | str |
 composantSelectionne isNil
 ifTrue: [Menu message: 'pas de composant selectionne']
 ifFalse:
  [str :=
    (Menu
      labels: 'tuer composant\echapper' withCrs
      lines: Array new
      selectors: #( Oui Non))
    popUpAt: Cursor offset.

   (str = #Oui)
   ifTrue:
    [self detruireComposantRectangle: composantSelectionne.
     graphPane form: (self dessinDestructionDe: composantSelectionne). ].
     composantSelectionne := nil.
     self changed: #renvois.
  ].
-------------------------------------------------------------------------
detruireComposantRectangle: unComposant
 " Detruit le composant de nom dans la base du projet vise."
 unComposant destruction.
 self baseDuProjet removeKey: unComposant nom.
-------------------------------------------------------------
do: unBlock
  "gestionBaseProjet - execute unBlock sur tous les elements de
   la base du projet."
  baseProjet do: unBlock.
-------------------------------------------------------------
drawMove: position rectangle: rect pen: crayon
 " trace rect en joignant au prealable position a rect origin."
 crayon
  place: position;
```

```
  goto: rect origin;
  goto: rect right @ rect top;
  goto: rect right @ rect bottom;
  goto: rect left @ rect bottom;
  goto: rect origin.
-----------------------------------------------------------------
drawRectangle: rect pen: crayon
 " Trace rect sur la vue graphique avec la plume crayon."
 crayon
  place: rect origin;
  goto: rect right @ rect top;
  goto: rect right @ rect bottom;
  goto: rect left @ rect bottom;
  goto: rect origin.
-----------------------------------------------------------
```

```
enveloppeComposant
 " Permet a l'utilisateur de modifier l'enveloppe du composant
  selectionne."
 | rect |
 composantSelectionne isNil
 ifTrue: [Menu message:'pas de composant selectionne'.^nil].

 graphPane form:
  (self dessinDestructionDe: composantSelectionne).

 rect := self getEnveloppe: composantSelectionne.

 (self
   verifierEnveloppe: rect
   nom: composantSelectionne nom )
 ifTrue:
  [self modifierTailleDe: composantSelectionne
    par: rect.
  ].
 composantSelectionne :=
    self baseDuProjet at: composantSelectionne nom.

 graphPane form: (self dessinCreationDe: composantSelectionne).
 self changed: #renvois.
-----------------------------------------------------------------
getCoin: unRectangle
 " permet a l'utilisateur saisir une nouvelle enveloppe de composant
  en modifiant son coin."
  | unCar frame  unPen rect arret
    initialPosition oldPosition rightMin bottomMin unPoint |
  frame := graphPane frame.
  unPen := Pen new
   combinationRule: Form reverse;
   mask: Form gray.
  rect := unRectangle origin + frame origin extent:
          unRectangle extent.
  rightMin := rect left + (PoliceDessin width * 2 ).
  bottomMin := rect top + PoliceDessin height.
  initialPosition := Cursor offset: rect corner.
  oldPosition := Cursor offset.

  self drawRectangle: rect pen: unPen.
  arret := false.
  [arret]
  whileFalse:
   [unCar := Terminal read.
    unPoint := Cursor offset.
    ((unPoint x ~= oldPosition x or: [unPoint y ~= oldPosition y])
    and:[unCar == 255 asCharacter])
    ifTrue:
     [((unPoint x > rightMin) & (unPoint y > bottomMin)
       and:[graphPane frame containsPoint: unPoint])
      ifTrue:
       [self drawRectangle: rect  pen: unPen.
        rect := rect origin corner: unPoint.
        self drawRectangle: rect  pen: unPen.]
      ifFalse:
       [Cursor offset: rect corner.].
      oldPosition := unPoint.
     ]
     ifFalse:
      [(MouseEvent
        ifTrue: [unCar == EndSelectFunction]
```

```
        ifFalse: [false])
      ifTrue: [arret := true].
    ].
  ].

^rect corner moveBy: graphPane frame origin negated.
```

```
getEnveloppe: unComposant
 " permet de saisir graphiquement un rectangle de composant.
  Renvoie nil si le rectangle saisi est incorrect."
 | estSaisie frame rect unCar|

 estSaisie := false.
 frame := graphPane frame.
 unComposant enveloppe isNil
 ifTrue: [ rect := 0@0 extent: 1@1.]
 ifFalse: [rect := unComposant enveloppe deepCopy].
 rect moveBy: frame origin.

 [estSaisie]
  whileFalse:
   [Cursor offset: rect origin.
    rect :=
     PointDispatcher rectangleFromUserOfSize:
      (PoliceDessin width * 2 + 1) @ (PoliceDessin height + 1)
     initSize: rect extent.

    (frame origin < rect origin) & (frame corner > rect corner)
     ifTrue:[estSaisie := true.].
   ].
 ^rect moveBy: frame origin negated
```

```
grilleEcran
 " dessine sur le graphPane une grille dont le pas est fixe
  par l'utilisateur."
 | composant commande grille|
 composant := composantSelectionne.
 image := nil.
 grille :=
  Prompter
   prompt: 'pas de grille choisi (<100) ?'
   defaultExpression: pasDeGrille printString.
 grille > 100 ifTrue: [grille := 10].
 self pasDeGrille: grille.
 self rafraichirEcran.
 self changed: #commandes.
 self changed: #renvois.
```

```
interface
 "cree l'interface graphique pour le projet composant receveur."
 | aTopPane textPane rapport maxSize|
 rapport := 5.
 aTopPane := TopPane new
  label: ' ProjetComposant ', self nom;
  minimumSize:
   (ListFont width *
    [maxSize := 1.
     self commandes do:
      [:s|( s size > maxSize)
          ifTrue: [maxSize := s size]
      ].
     maxSize.
    ] value * rapport + 22)   @
   (LabelFont height +
    (ListFont height + 1 * self commandes size)  +
    (TextFont height  * 5)
    );
  yourself.

 aTopPane addSubpane:
  (ListPane new
    model: self;
    name: #commandes;
    change: #commande:;
    framingBlock:
     [:box|
      box origin extent:
      (box width // rapport) @
       (box height - (TextFont height * 5))
     ]
  ).

 aTopPane addSubpane:
  (graphPane := GraphPane new
     model: self;
     name: #dessin:;
     change: #selectionComposant:;
     framingBlock:
      [:box |
       box origin + (box width // rapport @ 0) extent:
        (box width * (rapport - 1) // rapport) @
         (box height - (TextFont height * 5))
      ]
   ).
 aTopPane addSubpane:
  (textPane := TextPane new
    model: self;
    name: #renvois;
    framingBlock:
     [:box|
      box origin + (0 @ (box height - (TextFont height * 5)))
      corner: box corner]
  ).
 flotDeRenvois := textPane dispatcher.

 aTopPane dispatcher open scheduleWindow.
-------------------------------------------------------------------
modifierTailleDe: unComposant
 nouveauCoin: unPoint
 " Change la forme (agrandit, retrecit, allonge) de unComposant en prenant
 ((unPoint x <= unComposant gauche) or: [unPoint y <= unComposant haut])
 ifFalse:
  [unComposant destruction.
   self baseDuProjet removeKey: unComposant nom.
   ComposantRectangle
    nom: unComposant nom
    enveloppe: (unComposant enveloppe origin corner: unPoint)
    projet: self.
  ]
 ifTrue: [self error: 'enveloppe de composant incorrecte'].
-------------------------------------------------------------------
modifierTailleDe: unComposant
 par: unRectangle
 " Change la forme (agrandit, retrecit, allonge) de unComposant par
  unRectangle."
 unComposant destruction.
 self baseDuProjet removeKey: unComposant nom.
 ComposantRectangle
  nom: unComposant nom
  enveloppe: unRectangle
  projet: self.
-------------------------------------------------------------------
```

```
moveEnveloppe: unRectangle toDirection: uneDirection
 " permet a l'utilisateur de deplacer  l'enveloppe d'unComposant
  suivant une uneDirection. si celle ci n'est pas precisee,
  l'utilisateur peut se deplacer dans toutes les uneDirections."
  | origine frame
    unPen unPoint deplacement unCar arret
    rect initialPosition oldPosition
    leftLimit rightLimit topLimit bottomLimit
    vDS rectVoisin
    hauteur largeur|
  frame := graphPane frame.
  unPen := Pen new
   combinationRule: Form reverse;
   mask: Form gray.
  rect := unRectangle translateBy: frame origin.
  Cursor offset: rect origin.
  initialPosition := rect origin deepCopy.

  largeur := rect width.
  hauteur := rect height.
  rightLimit := frame right - largeur.
  leftLimit := frame left.
  topLimit := frame top.
  bottomLimit := frame bottom - hauteur.
  uneDirection notNil
  ifTrue:
   [vDS := composantSelectionne voisins at: uneDirection.
    vDS notEmpty ifTrue:
     [rectVoisin := (vDS at: 1) enveloppe translateBy: frame origin.
      (uneDirection == ComposantRectangle est) ifTrue:
       [rightLimit := rectVoisin right - largeur - 1].
      (uneDirection == ComposantRectangle ouest) ifTrue:
       [leftLimit := rectVoisin left + 1].
      (uneDirection == ComposantRectangle nord) ifTrue:
       [topLimit := rectVoisin top + 1].
      (uneDirection == ComposantRectangle sud) ifTrue:
       [bottomLimit := rectVoisin bottom - hauteur - 1].
     ].
   ].
  oldPosition := initialPosition.
  self drawMove: initialPosition
       rectangle: rect
       pen: unPen.
  arret := false.
  [arret]
  whileFalse:
   [unCar := Terminal read.
    unPoint := Cursor offset.
    ((unPoint x ~= oldPosition x or: [unPoint y ~= oldPosition y])
    and:[unCar == 255 asCharacter])
    ifTrue:
     [uneDirection isNil
      ifFalse:
       [deplacement := 0@0.
        (((uneDirection == ComposantRectangle est) |
          (uneDirection == ComposantRectangle ouest))
         and: [unPoint x ~= rect left])
         ifTrue:
          [deplacement x: unPoint x - rect left.
           (uneDirection == ComposantRectangle est)
           ifTrue:
            [(deplacement x < 0)
             ifTrue:
              [deplacement x:
                (deplacement x max:(initialPosition x - rect left))]
            ]
           ifFalse:
            [(deplacement x > 0)
             ifTrue:
              [deplacement x:
                (deplacement x min:(initialPosition x - rect left))]
            ].
          ].

        (((uneDirection == ComposantRectangle nord) |
          (uneDirection == ComposantRectangle sud))
         and: [unPoint y ~= rect top])
        ifTrue:
         [deplacement y: unPoint y - rect top.
          (uneDirection == ComposantRectangle nord)
          ifTrue:
           [(deplacement y > 0)
            ifTrue:
             [deplacement y:
               (deplacement y min:(initialPosition y - rect top))]
```

```
          ]
         ifFalse:
          [(deplacement y < 0)
           ifTrue:
            [deplacement y:
              (deplacement y max:(initialPosition y - rect top))]
          ].
        ].
       ]
      ifTrue: [deplacement := unPoint - rect origin].

      unPoint := rect origin + deplacement.

      (deplacement x > 0)
      ifTrue:[ unPoint x: (unPoint x min: rightLimit)].
      (deplacement x < 0)
      ifTrue:[ unPoint x: (unPoint x max: leftLimit )].
      (deplacement y > 0)
      ifTrue:[ unPoint y: (unPoint y min: bottomLimit)].
      (deplacement y < 0)
      ifTrue:[ unPoint y: (unPoint y max: topLimit )].

      Cursor offset: unPoint.
      self drawMove: initialPosition
           rectangle: rect
           pen: unPen.
      rect := unPoint extent: rect extent.
      self drawMove: initialPosition
           rectangle: rect
           pen: unPen.
     oldPosition := Cursor offset.
     ]
     ifFalse:
      [(MouseEvent ifTrue: [unCar == EndSelectFunction])
       ifTrue: [arret := true].
    ].
   ].

 ^rect translateBy: graphPane frame origin negated.
-------------------------------------------------------------------
nom
 " Renvoie le nom du projetComposant visualise."
 ^nom
-------------------------------------------------
nom: uneChaine
 " affecte le nom de projet a uneChaine."
 ^nom := uneChaine
-----------------------------------------
oterComposant: composant
  "gestionBaseProjet - supprime un composant de la base du projet."
  baseProjet removeKey: composant nom.
-------------------------------------------------------------------
pasDeGrille: unEntier
 " affecte la variable d'instance pasDeGrille a unEntier."
 pasDeGrille := unEntier.
-----------------------------------------------------------
rafraichirEcran
 " remet a jour l'ensemble des vues."
 graphPane form: (self dessin: graphPane frame).
 composantSelectionne := nil.
 self changed: #renvois.
 self changed: #commandes.
-----------------------------------------------
 " affiche dans la fenetre de renvois les caracteristiques du
  composant selectionne."

 composantSelectionne isNil
 ifTrue:
  [^String new ]
 ifFalse:
  [^composantSelectionne showCaracteristiques].
-----------------------------------------------------------
selectionComposant: unPoint
 " selectionne graphiquement un composant qui contient unPoint."
 | composant  |
 composant :=
  self baseDuProjet
   detect: [:c | c enveloppe
                  containsPoint: unPoint - graphPane frame origin]
   ifNone: [nil].
 composantSelectionne := composant.
 self changed: #renvois.
-------------------------------------------------------------------
```

```
verifierEnveloppe: unRectangle nom: unNom
 " controle l'existence des conflits entre enveloppes de
  composants. unRectangle intersecte t'il l'un des composants
  de la base sauf celui de nom unNom."
 | cmpsEnConflit |
  cmpsEnConflit := OrderedCollection new.
  self baseDuProjet do:
  [:cmp|
   ((cmp nom ~= unNom) and:
    [cmp enveloppe origin <=  unRectangle corner
      and: [unRectangle origin <= cmp enveloppe corner]])
   ifTrue: [cmpsEnConflit add: cmp nom].
  ].
  cmpsEnConflit isEmpty ifTrue: [^true].
  cmpsEnConflit addFirst:'conflits avec:'.
  (Menu
   labelArray: cmpsEnConflit
   lines: Array new
   selectors: cmpsEnConflit) popUpAt: Cursor offset.
  ^false.
```

```
OrderedCollection variableSubclass: #UniondIntervalles
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
UniondIntervalles class methods:


UniondIntervalles methods:
* uneUnion
 "intersection - renvoie l'intersection entre unionRec et uneUnion.
  C'est a dire l'union de tous les recouvrements d'intervalles entre
  unionRec et uneUnion. On renvoie le resultat dans une nouvelle
  UniondIntervalles.
  ex: unionRec = [0,2] [4,6] [11,13]
      uneUnion =       [3,5] [10,   14]
      unionRec * uneUnion = [4,5] [11,13]."
 | inter |
 inter := UniondIntervalles new.
 uneUnion do: [:i | inter addAll: (self ** i)].
 ^inter
-------------------------------------------------------------------------
** unInt
 "intersection - renvoie l'intersection entre l'unionRec et unInt :
  i.e. une UniondIntervalles contenant des intervalles qui decrivent
  l'intersection.
  ex: unionRec = [0,2] [4,6] [11,13]
      unInt    =       [3,      12]
      unionRec ** unInt = [3,6] [11,12].
  On renvoie donc l'union des recouvrements entre les intervalles de
  unionRec et unInt. une nouvelle uniondIntervalles est creee."
 ^(self intersectantsDe: unInt)
  collect:
   [:i |
    ((i x > unInt x) ifTrue: [i x] ifFalse: [unInt x]) @
    ((i y < unInt y) ifTrue: [i y] ifFalse: [unInt y])
   ]
-------------------------------------------------------------------------
+ uneUnion
  "superposition - realise l'operation de superposition entre unionRec
   et uneunion. C'est la generalisation de la superposition precedente
    ex: U1 : [2,5] [9,15] [20,22]
        U2 : [3,8]      [18,23]      [27,30]
        U1 + U2 = U3 : [2,8] [9,15] [18,23] [27,30]
   resultat construit par recurrence."
 | res  "contient les R0,R1,R21,...,Rn sucessifs. Rn = resultat" |
 "creation d'une nouvelle UniondIntervalles contenat unionRec"
 res := self deepCopy.
 "creation des Ri successifs"
 uneUnion do: [:i | res := res ++ i].
 "Rn = resultat final renvoye"
 ^res
-------------------------------------------------------------------------
```

```
++ unInt
 "superposition - Renvoie la superposition de unionRec et de unInt.
  Cette operation consiste a determiner la plus petite
  UniondIntervalles contenant a la fois unionRec et unInt. Le resultat
  est tel que:
   uneUnion estContenueDans: (uneUnion ++ unInt)
   unInt estContenuDans: (uneUnion ++ unInt)
  D'un point de vue geometrique ou l'on considere les intervalles
  comme des segments et les uniond'intervalles comme des ensemble
  de segments de la droite reelle, il s'agit de la superposition d'un
  segment sur un ensemble de segments. Le resultat est renvoye dans
  une nouvelle UniondIntervalles."
 | intersectants "contient les intersectants de self avec unInt"
   superposition "contient le resultat renvoye"
   indice        "indice de parcours de collection quelconque"
   size          "la taille de self"
   englobant     "la superposition des intersectants et de unInt"
 |

 "copie de self dans une nouvelle UniondIntervalles"
 superposition := self deepCopy.
 size := self size.

 "cas a: unInt est contenu dans self => renvoyer uneUnion"
 (self contient: unInt) ifTrue: [^superposition].

 "determine les intersectants de unInt"
 intersectants := self intersectantsDe: unInt.

 "cas generaux a traiter"
 intersectants isEmpty
  ifTrue:
    ["cas b: aucun elt de unInt n'appartient a self"
     indice := 1.

     "on cherche la position d'insertion de unInt"
     [(indice <= size) and: [(self at: indice) y < unInt x]]
     whileTrue: [indice := indice + 1].

     "insertion de unInt"
     (indice > size)
     ifTrue: [superposition add: unInt]
     ifFalse: [superposition add: unInt before: (self at: indice)].
    ]
  ifFalse:
    ["cas c: uneUnion ** unInt non vide sauf cas a"

     "determiner les bornes de l'intervalle englobant"
     englobant :=
       ((intersectants first x) min: (unInt x)) @
       ((intersectants last y)  max: (unInt y)).

     "on determine la position du premier des intersectants"
     indice := self indexOf: intersectants first.

     "suppression des intersectants"
     superposition removeAll: intersectants.

     "intersection de l'englobant"
     (indice > 1 )
      ifTrue:
        [superposition add:englobant
```

```
      ifFalse: [superposition addFirst: englobant].
    ].
 ^superposition
-------------------------------------------------------------------------
\ uneUnion
  "difference - realise l'operation de difference entre unionRec et
   uneUnion. C'est la generalisation de l'operateur \\ aux
   UniondIntervalles.
    ex: self : [2,5] [7,9] [12,16]
        uneUnion : [1, 7.5] [8, 13] [18, 21
        self \ uneUnion renvoie [7.5,  8] [13, 16]
   On obtient le resultat par recurrence. On renvoie une nouvelle
   UniondIntervalles contenant le resultat."
   | res  "contient le resultat intermediaire et final" |
   "creation d'une nouvelle UniondIntervalles contenant unionRec"
   res := self deepCopy.
   "creation des Ri successifs"
   uneUnion do: [:i | res := res \\ i].
   "Rn = resultat final, renvoi"
   ^res
-------------------------------------------------------------------------
```

```
\\ unInt
 "difference - realise l'operation de difference entre unionRec
  et unInt. Elle consiste a oter, de tous les intervalles de unionRec
  qui intersectent unInt, la partie de unInt qu'ils coupent. Cas
  particulier: si unInt et l'intervalle de unionRec considere sont
  contigus, l'operation n'a pas d'influence sur cet intervalle (vient
  du fait que ce sont des intervalles fermes).
   ex: self  : [2,5] [7,9] [12,15]
       unInt :  [3 ,         14]
       self \\ unInt : [2,3] [14,15]."
 | inters "les intersectants de self avec unInt"
   aDiffI "le resultat de l'operation"
   indice "indice de parcours qcq" |

 "creation d'une nouvelle UniondIntervalles contenant self"
 aDiffI := self deepCopy.
 "determiner les intersectants de unInt"
 inters := self intersectantsDe: unInt.

 "modification - recherche de la difference"
 inters isEmpty
   ifTrue:  [^aDiffI]
   ifFalse:
     [inters do:
       [:i |
        indice := aDiffI indexOf: i. "indice de i dans aDiffI"
        "cas 1 et 2"
        (i x < unInt x)
          ifTrue:
            ["le resultat pour cet intervalle commence a unInt x"
             (aDiffI at: indice) y: unInt x.
             "cas 2"
             (i y > unInt y)
             ifTrue:
               ["rajouter la partie restante droite de cet intervalle"
                aDiffI add:   (unInt y @ i y)
                       after: (aDiffI at: indice) ]
            ]
          "cas 3 et 4"
          ifFalse:
            [(i y > unInt y)
             ifTrue:
               {"le resultat pour cet intervalle commence a unInt y"
                (aDiffI at: indice) x: unInt y}
             ifFalse:
               ["intervalle contenu dans unInt; a oter de self"
                aDiffI remove: i}
            ]
       ].
     "renvoie le resultat"
     ^aDiffI
     ]
-------------------------------------------------------------------------
contenantDe: unInt
 "prive - renvoie l'intervalle de unionRec qui contient unInt,
  c'est a dire l'intervalle unique  i verifiant l'assertion:
  (I x <= unInt x) ^ (I y >= unInt y)
  Renvoie nil si cet intervalle n'existe pas dans unionRec "

 self do:
   [:i |
    (i x > unInt x)
    ifTrue: [ ^nil]
```

```
    ifFalse: [ (i y >= unInt y) ifTrue: [^i] ].
   ].
   ^nil
-----------------------------------------------------------------
contient: unInt
 "inclusion - teste si unionRec contient unInt, renvoie true si vrai
  false sinon."
 ^(self contenantDe: unInt) notNil
-----------------------------------------------------------------------
contientUnion: uneUnion
 "inclusion - teste si unionRec contient uneUnion, c'est a dire si
  tous les intervalles de uneUnion sont contenus dans unionRec."
 uneUnion isEmpty
   ifTrue: [^true]
   ifFalse:
    [ uneUnion do:
      [:i | (self contient: i) ifFalse: [ ^false] ]
    ].
   ^true
-----------------------------------------------------------------
```



```
deepCopy
 "prive - renvoie une copie de l'unionRec telle que tous les
  elements de unionRec sont copies par valeurs dans cette copie.
  En sortie il existe deux objets differents de contenus identiques
  references differement mais representant la meme valeur semantique:
   U (i1, i2, ... iN) , U1 (j1, j2, ... jN) et
   pour tout m appartenant a [1..n] im.x = jm.x et im.y = jm.y."
 | copie |
 copie _ UniondIntervalles new: (self size).
 self do: [ :i | copie add: (i x @ i y)].
 ^copie
-----------------------------------------------------------------------
intersectantsDe: unInt
 "prive - Renvoie, pour l'unionRec, la liste des intervalles qui
  coupent (intersectent) unInt; cette liste est rangee par ordre
  croissante dans une UniondIntervalles creee pour l'occasion."
 ^self select:
   [:i |
    ( (unInt x > i x) and: [ unInt x < i y] ) or:
    [ (unInt y >= i x) and: [ unInt x <= i x] ]
   ]
-----------------------------------------------------------------
```

Résumé:

Cette thèse comporte deux études portant sur l'intégration de techniques d'intelligence artificielle (I.A.) en conception assistée par ordinateur (C.A.O.).

La première porte sur le problème logiciel de base qui consiste à interfacer une partie logicielle écrite grâce à un langage spécialisé en IA (LISP, PROLOG, SMALLTALK, ...), avec une partie écrite à l'aide d'un langage procédural classique (C, PASCAL, ADA, FORTRAN, ...). Nous présentons une description générale des interfaces possibles et les résultats de multiples expérimentations. Nous classifions les interfaces logicielles en deux catégories : les "interfaces directes" sont basées sur un couplage fort entre les parties logicielles et favorisent la performance, et les interfaces "indirectes" sont basées sur un couplage faible et favorisent l'aisance de réalisation. Le choix d'un type d'interface dépendra donc de la priorité accordée à l'efficacité ou à la facilité de réalisation.

La deuxième partie porte sur l'étude d'un système d'aide intelligent dédié au placement interactif de composants géométriques dans un espace à deux dimensions. Un modèle d'architecture multi-agents y est décrit, où chaque composant est considéré comme un acteur et sait déterminer seul ses accointances géométriques, les aspects fonctionnels et sémantiques du domaine d'application n'étant pas traités ici. Nous avons de plus étudié deux "moteurs de résolution de conflits", basés sur le caractère décentralisé du modèle et pouvant être implantés indépendamment sur chaque composant. Ce modèle, appelé MAPS (**M**ulti-**A**gent model for interactive **P**lacement **S**ystems) tire parti de la nature modulaire des composants et du caractère souvent local des conflits géométriques. Il montre que l'on peut envisager de réaliser un système d'aide à la conception intelligent dédié aux problèmes de schématique, basé sur les techniques d'IA distribuée, et dans lequel l'utilisateur interagit avec des entités logicielles actives indépendantes.

MAPS consistue ainsi un premier pas vers des systèmes d'aide à la conception intelligents dans des domaines où les tâches d'ordonnancement spatial sont importantes, répétitives, et difficiles, comme c'est souvent le cas en schématique et plus généralement en CAO.

Mots clé:

CAO et schématique, interface logicielle, interface homme-machine, aide à la conception, intelligence artificielle distribuée, ordonnancement spatial, accointance, multi-agents, architecture décentralisée, acteur, comportement social, voisinage géométrique, conflit géométrique, contraintes géométriques, topologiques, fonctionnelles, résolution distribuée, SMALLTALK, classe, objet.