Base de données et gestion de configurations dans un atelier de génie logiciel

Saïd Ghoul

THESE

présentée à

l'Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR INGENIEUR
«informatique»

par

Saïd GHOUL

BASE DE DONNEES ET GESTION DE CONFIGURATIONS DANS UN ATELIER DE GENIE LOGICIEL.

Thèse soutenue le 21 décembre 1983 devant la commission d'examen.

S. KRAKOWIAK	Président
M. ADIBA	Examinateurs
R. JACQUART	
J. MOSSIERE	
B. ROUGEOT	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

Je tiens à remercier,

Monsieur S. Krakowiak, professeur à l'U.S.M.G., et Monsieur J. Mossiere, professeur à l'I.N.P.G., pour la formation et les conseils qu'ils m'ont offerts, pour la lecture soigneuse du manuscrit, et pour avoir accepté, le premier de présider mon jury, le second d'y participer.

L'intérêt, les conseils, et les corrections systématiques que m'a offerts Monsieur S. Krakowiak, durant toute la période de réduction, m'ont beaucoup encouragé et aidé. Je lui en suis reconnaissant.

Monsieur M. Adiba, professeur à l'U.S.M.G., Monsieur R. Jacquart, ingénieur à l'O.N.E.R.A., Centre d'Etude et Recherche de Toulouse, et Monsieur B. Rougeot, ingénieur au C.N.E.T. Lannion, pour leur participation au jury. La critique faite par Monsieur R. Jacquart sur mon manuscrit m'a beaucoup aidé.

Le sous-groupe "Modularité et archivage" de l'équipe ADELE (°), et particulièrement Monsieur J. Estublier, ingénieur au C.N.R.S., avec qui j'ai étroitement travaillé. J'ai trouvé auprès de lui un soutien moral, et une grande volonté qui ont été des facteurs importants pour la réalisation de ce travail. La plupart des idées qui ont présidé à la conception de la base ADELE et qui sont présentées dans ce travail lui sont dues.

Toute l'équipe ADELE, pour l'amitié que ses membres m'ont toujours témoignée ; en particulier ceux qui ont critiqué et corrigé le manuscrit : J.L Cheval, J. Chassin De Kergommeaux, et C. Lenne.

Monsieur E. André, directeur du projet CONCERTO du C.N.E.T. Lannion, pour l'intérêt et l'apport financier offerts au projet ADELE.

Je tiens à remercier également J. et J.L Cheval, M. et C. Lenne, et B. et L. Héziani, pour l'attention particulière et l'hospitalité qu'ils m'ont très souvent offertes. Celles-ci m'apportèrent un soutien inestimable, et j'en garderai un souvenir agréable. Je leur en suis très reconnaissant.

Je remercie enfin le service de reprographie de l'IMAG pour la réalisation matérielle de cet ouvrage.

Que tous, au delà de l'aspect formel des mots, trouvent ici l'expression de ma reconnaissance et de mes remerciements.

Saïd Ghoul.

(°) Le sous-groupe "modularité et archivage" :

J.L Cheval, J. Estublier, S. Ghoul, et S. Krakowiak.

A ma famille.

TABLE DES MATIERES

1 INTRODUCTION

1.1 Environnement de l'étude

L'étude dont il est question dans cette thèse est réalisée dans le cadre du projet ADELE. Le projet ADELE (Atelier de DEveloppement de LogicIEl) a pour but d'étudier divers aspects de la conception et de la réalisation d'ateliers intégrés de logiciels, et de valider cette étude par la réalisation d'un prototype expérimental au laboratoire IMAG (BRI 81, EST 83a).

Compte tenu de son expérience, acquise au cours des projets antérieurs (KRA 76, CHV 80), et de sa connaissance de l'état de l'art (KRA 82), l'équipe a été amenée à étudier les quatre aspects suivants :

- les principes de conception de l'interface du poste de travail : gestion de fenêtres, édition d'interfaces, et affichage des informations (HER 82),

- la décomposition modulaire et l'archivage des programmes dans une base de données, la gestion de versions multiples de modules et de configurations, et le maintien de la cohérence, qui font l'objet de cette thèse,

- la représentation interne des programmes sous forme intermédiaire, construite autour d'un arbre abstrait syntaxique, et les outils qui manipulent cette représentation (constructeur syntaxique et interpréteur) (MOS 82),

- la génération de code exécutable à partir de la représentation intermédiaire de programmes, ainsi que le paramétrage de cette génération par les caractéristiques statiques de la machine cible, exprimées dans un langage de description adéquat (SAN 83).

L'ensemble des outils d'ADELE sont intégrés autour d'une base de données qui contient l'ensemble des objets (entités et associations) de l'atelier, exprimant la composition de systèmes à partir de modules, et la cohérence des systèmes à versions multiples (fig.1.1).

Les phases du cycle de vie de logiciel supportées se réduisent à la phase de programmation (conception de programmes, leur codage, et mise au point).

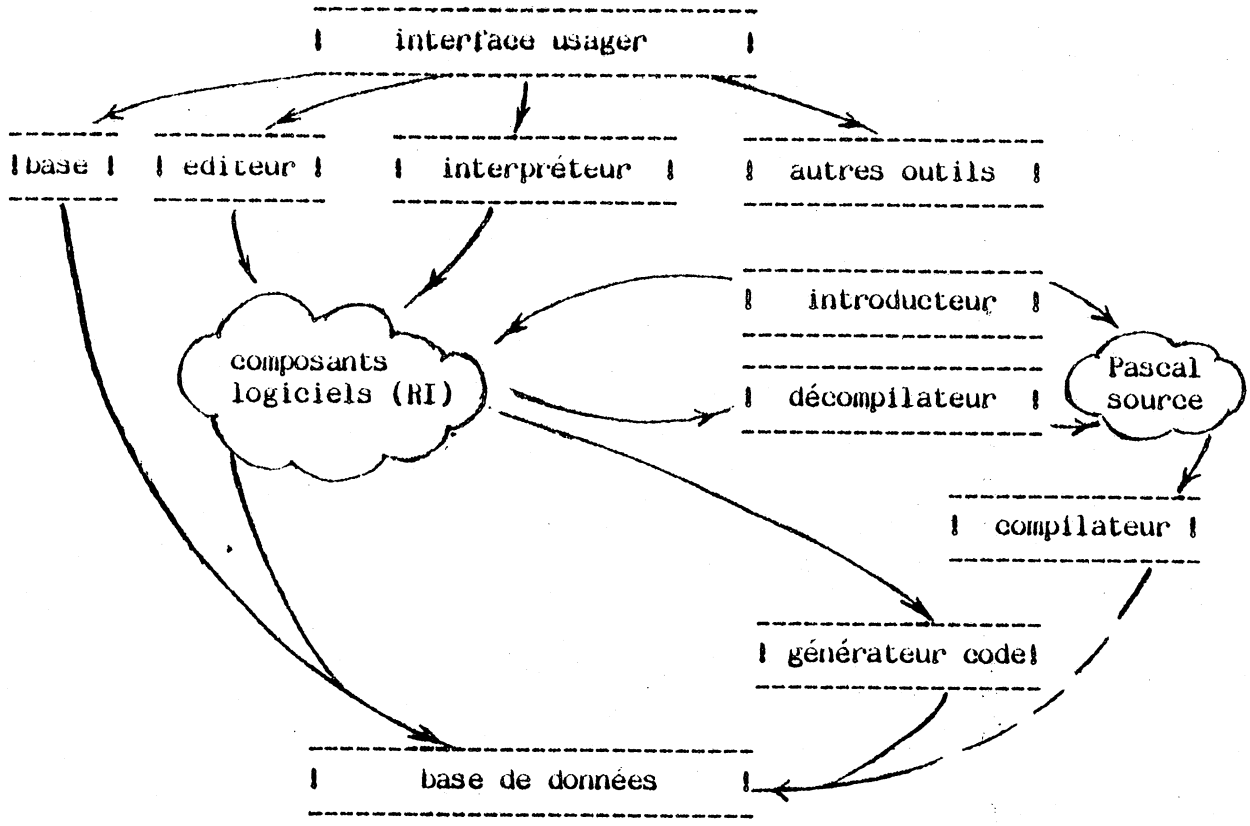


Fig.1.1 Architecture générale de l'atelier ADELE.

1.2 Objectifs et motivations de la recherche

Cette thèse est une étude de l'aspect base de données et gestion de configurations dans un Atelier de Génie logiciel. L'importance de cet aspect (cf. 2.1.1) est capitale puisque la base doit constituer le noyau autour duquel doivent être intégrés les différents outils de cet atelier.

Cependant plusieurs questions se posent :

- l'utilisation des bases de données existantes. Ces bases sont conçues pour supporter la gestion de données élémentaires (chaines de caractères, chiffres, ...) ayant une sémantique simple, ce qui n'est pas le cas des données logicielles (modules, configurations, documents, ...). Il faut donc identifier et résoudre les problèmes que pose l'utilisation des bases de données existantes, et évaluer ces solutions,

- la modélisation des logiciels par un formalisme intégrant la description des données et leurs traitements. Ce formalisme doit être clair et concis pour faciliter la compréhension des logiciels. En effet le modèle d'un logiciel représente sa vue externe par les utilisateurs,

- l'expression des règles de décomposition, de composition, et d'évolution des logiciels et leur gestion automatique par la base,

- le maintien de l'intégrité de la base quand ces règles et logiciels évoluent.

On assiste actuellement à divers travaux de recherche et d'expérimentation dans ce domaine (des références seront données au cours de cette thèse) ; nos efforts s'orientent dans ce sens.

1.3 Plan de la thèse

Nous commencerons (au chapitre 2) par un survol de l'état actuel de l'art en étudiant l'aspect base de données dans quelques ateliers representatifs. Cette étude nous permettra d'identifier les besoins d'un atelier de génie logiciel, les problèmes qui se posent pour les réaliser, et les solutions adoptées.

Les chapitres 3, et 4 sont consacrés à la description de l'approche prise dans ADELE pour répondre aux besoins identifiés au chapitre 2 : les modèles, l'intégrité, et l'évolution des logiciels. Le chapitre 5 décrit la réalisation de cette approche dans ADELE, et le chapitre 6 décrit son expérimentation et son évaluation.

Nous terminerons par une une conclusion (chapitre 7) en particulier sur l'apport et les perspectives de la base ADELE, et en général sur les perspectives des bases de données pouvant supporter les besoins d'un Atelier de Génie Logiciel.

A la thèse sont associées trois annexes : un exemple complet de construction de configurations (A1), les grandes lignes des algorithmes qui permettent la construction et l'évolution de ces configurations (A2), et enfin quelques commandes simplifiées d'utilisation de la base (A3).

2 BASES DE DONNEES DANS LES ATELIERS DE GENIE LOGICIEL

2.1 Généralités

2.1.1 Importance d'une base de données dans un atelier de logiciel

L'Atelier de Génie Logiciel (AGL), le plus simple et primitif, se réduit à un système d'exploitation classique, composé d'un ensemble d'outils (compilateurs, chargeurs, éditeurs, ...). Chacun de ces outils manipule ses propres données mémorisées par des utilitaires (sgf, bibliothécaires simples, ...). Ces utilitaires n'assurent aucune cohérence de ces données. Cet environnement ne supporte que la phase de programmation du cycle de vie d'un logiciel (cf. fig.2.1) ; fonctionnellement, il correspond à l' "environnement 1" décrit par HOWDEN (HOW 82).

Par contre un Atelier de Génie Logiciel, dans l'état actuel des besoins (voir par exemple (STN 80)), de l'art et de la technologie (voir par exemple (KRA 82)), doit supporter, ne serait-ce que partiellement, toutes les phases du cycle de vie d'un logiciel. Ses différents outils doivent être intégres et construits autour d'une base de données qui constituera le noyau de l'atelier. Dans sa proposition d'une terminologie et d'axes de comparaison des Ateliers de Génie Logiciel, CHEATHAM (CHT 81) place la base de données comme composante principale d'un AGL, autour de laquelle peuvent être intégrés les différents outils.

HOWDEN (HOW 82) définit cette base comme moyen de conserver et de retrouver les produits de chaque phase du cycle de vie, de modéliser et de documenter ces produits. Cette base doit être construite autour de trois concepts : un objet, ses propriétés et ses relations avec d'autres objets. Un objet peut être un code objet, un plan de test, un document de conception, un graphique de planification de la réalisation du logiciel, une spécification des besoins, etc. HOWDEN a également cité quelques relations entre ces objets : l'appel d'une procédure par une autre, la précédence d'une tâche du projet par rapport à une autre, la relation entre une configuration d'un logiciel et l'ensemble de composants qui la constituent, etc.

Le problème de l'intégrité (cf. 2.1.2.2) de cette base doit alors être résolu. Les utilitaires classiques n'étant plus suffisants deviennent le noyau de systèmes plus performants, les Systèmes de Gestion de Bases de Données. Parmi les fonctions d'un SGBD on cite : la prise en charge des problèmes liés à la mémorisation, à l'accès et à la sécurité de l'information. L'outil doit connaitre toutes les informations nécessaires à son fonctionnement, sans avoir à se soucier de la manière de les retrouver, mais en étant assuré de leur cohérence. Le SGBD permet ainsi une communication indirecte et contrôlée entre les différents outils de l'AGL.

Le rôle important d'une base de données dans un AGL fait qu'elle est un domaine de recherche d'intérêt considérable, actuellement et dans les années à venir. OSTERWEIL (OST 81), dans sa réflexion sur la recherche dans le domaine des AGL, place la base de données parmi les axes importants et propose de grandes lignes pour cette recherche. Nous pouvons citer : étude et création de modèles précis et détaillés des tâches logicielles (modèles des entités logicielles, ...) ce qui est actuellement inexistant. Sur ces modèles seront conçus les schémas des bases de données et les systèmes d'informations seront construits. Il faut également déterminer le type des requêtes qui peuvent être adressées à une base de données. Ces expériences permettront de trouver des réponses aux problèmes de stockage et d'accès dans les bases de données.

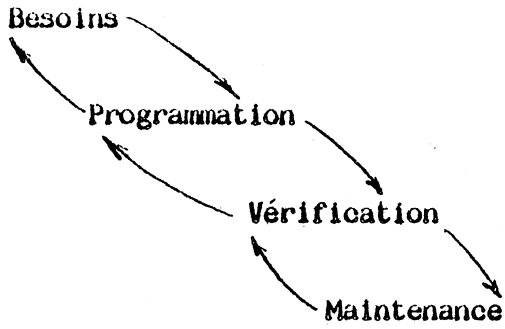


Fig.2.1 Cycle de vie d'un logiciel.

2.1.2 Aspects étudiés

Dans notre étude des bases de données dans les AGL, nous focalisons notre intérêt sur tous les aspects de l'atelier ayant une influence directe sur la conception et la réalisation de son SGBD. Ces aspects peuvent constituer des paramètres pour la conception d'un SGBD général, pouvant supporter les besoins de divers ateliers. Nous les examinons dans un ordre de détails croissant.

2.1.2.1 Généralités sur l'atelier

Plusieurs caractéristiques d'un atelier définissent les grandes lignes de son SGBD ; nous pouvons citer :

- Les phases du cycle de vie d'un logiciel que l'atelier assiste en automatisant, par des outils appropriés, les activités de chaque phase. Nous trouvons dans la littérature (voir par exemple (MUR 81, WAS 81)) la spécification de chaque phase et de ses principales activités. La complexité des entités à gérer, ainsi que leurs dépendances, et la complexité du maintien de la cohérence de ces informations ont un impact considérable sur la conception du SGBD en entier,

- la longévité des logiciels produits par un atelier exige un SGBD apte à suivre aisément l'évolution des logiciels (évolution des textes sources, évolution structurelle des entités) et à protéger les informations contre toute altération,

- le volume des logiciels supportés influe sur les capacités de stockage, la recherche des informations et la capacité du SGBD à refléter la répartition des membres du projet ainsi que celle des moyens,

- le domaine d'application supporté influe sur la structure des entités et la nature de leurs relations. Un SGBD spécialisé dans la gestion d'applications dans un domaine particulier peut être plus efficace qu'un SGBD général.

2.1.2.2 Expression des besoins

Pour supporter le développement de logiciels en l'assistant par automatisation du maximum d'activités, l'atelier a besoin de connaitre certaines informations vitales sur le logiciel à produire. Ces informations peuvent être paramétrées pour un SGBD général, ou figées pour un SGBD spécialisé. Nous citons :

- le modèle du logiciel à produire : le SGBD doit connaitre les différentes entités à gérer, leurs propriétés (caractéristiques), et les relations qui les relient. Un SGBD spécialisé, où le modèle des logiciels supportés est figé, est plus performant en étant plus adapté à cette structure ; alors qu'un SGBD général, paramétré par ce modèle, permettra des modèles divers, au détriment de sa performance.

Il existe trois grandes catégories de modèles de données (DEL 82), selon la nature des associations qu'ils modélisent : le modèle réseau, le modèle hiérarchique, et le modèle relationnel. Les deux premiers modèles ont donné naissance à la plupart des principaux systèmes commercialisés actuellement (IMS, IDS, ...), car ils ont été conçus dans la période 1965-1970.

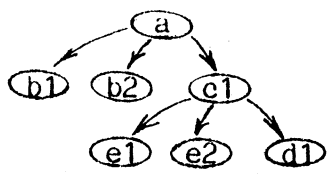
- le modèle réseau : le modèle réseau est basé sur deux concepts :

  - le type d'enregistrement logique : il décrit les attributs caractérisant un ensemble d'entités ; une entité correspond à un enregistrement logique,

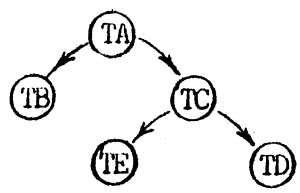
  - le lien : un lien peut se définir comme la représentation d'une association. L'association est une perception abstraite de la réalité, alors que le lien est sa matérialisation. Un lien L est défini entre deux types d'enregistrements R et S, dans une direction donnée. Il offre un mécanisme d'accès qui, à partir d'un enregistrement r de type R, permet d'accéder aux enregistrements s1, s2, ..., sn de type S qui lui sont reliés.

- le modèle hiérarchique : les éléments de base de ce modèle sont des enregistrements logiques reliés entre eux pour construire une arborescence valuée. Nous supposons connues les notions d'arborescence et de forêt qui sont utilisées dans ce modèle, et nous définissons la notion d'arborescence valuée.

Une arborescence valuée est une arborescence dont les noeuds sont valués par des enregistrements logiques. Elle est définie à partir d'une arborescence dont les noeuds sont des types d'enregistrements logiques. Cette arborescence est bien souvent appelée schéma hiérarchique.



arborescence valuée



arborescence des types d'enregistrements

- le modèle relationnel : c'est à partir de 1970 qu'a été introduit le modèle relationnel. Il marque une étape décisive dans l'évolution des bases de données. Ce modèle est muni d'un formalisme mathématique permettant de définir les entités et les relations.

Une base de données est décrite par un schéma noté :

$$S = ((R1, R2, \ldots, Rn), \Sigma)$$

qui consiste en un ensemble de relations (Ri) et un ensemble de propriétés inter-relations (Σ).

Chaque relation (ou intention) est composée d'un identificateur R, d'un ensemble fini d'attributs (A1, A2, ..., An), et d'un ensemble de propriétés intra-relations (Σr). Une relation R est alors notée :

$$R = (A1, A2, \ldots, An), \Sigma r$$

Si D1, D2, ..., Dn représentent des ensembles de valeurs (appelés domaines) respectivement de A1, A2, ..., An, l'extension de R (ou simplement la relation R) est définie comme un sous-ensemble du produit cartésien de ces domaines :

$$R (A1, A2, \ldots, An) \subseteq D1xD2x\ldots xDn$$

Pour représenter au mieux la réalité modélisée, les deux ensembles Σ et Σr , de règles d'intégrité complètent le schéma de la base. Ces règles permettent de réduire et de contrôler l'ensemble des états pris par la base.

Chaque relation est généralement représentée sous forme d'un tableau à n colonnes dont chacune est appelée attribut. Chaque ligne de R, appelée n-uplet est identifiée de manière unique, par une clé primaire qui correspond à la valeur d'un (ou plusiuers) de ses attributs.

- l'intégrité : nous ne rappelons pas les diverses définitions de l'intégrité des données qui ont été données dans la littérature, mais nous disons simplement que l'intégrité des données est leur correspondance à la réalité. Un SGBD doit offrir à l'utilisateur la possibilité de définir des règles qui permettent le maintien de cette intégrité. Ces règles sont appelées contraintes d'intégrité. Elles correspondent à des propriétés qui devront toujours être vérifiées dans la base de données quelles que soient les valeurs enregistrées.

l'intégrité recouvre deux notions :

- la protection des données : contrôle de l'accès concurrent, droits d'accès, sécurité de la base (en cas d'incident logique ou matériel), ...

- l'intégrité sémantique : possibilité d'assurer que les informations conservées dans la base sont cohérentes par rapport à leur signification : c'est ce genre d'intégrité qui nous intéresse dans notre étude. Nous considérons deux grandes catégories de Contraintes d'Intégrité Sémantiques (CIS) :

  - les CIS structurelles : concernent les structures des données, et donc leur modèle, il s'agit des dépendances hiérarchiques, des types d'objets, ... Elles évoluent avec le schéma de la base de données. Ce sont des CIS statiques,

  - les CIS par valeur : concernent les données (état de la base) et leurs transition (passage d'un état à un autre). Elles peuvent être définies ou détruites à tout moment, même si le schéma de la base reste stable. Ce sont des CIS dynamiques.

Ces contraintes constituent essentiellement la méthode du développement des logiciels supportés (règles de décomposition, de composition, et d'évolution des logiciels). Le SGBD doit vérifier la cohérence de la base après chaque manipulation de ces contraintes,

Nous trouvons plus de détails sur les CIS dans (FER 83).

- l'évolution des logiciels gérés : souvent au cours du développement ou même de l'utilisation d'un logiciel, la modification, la suppression ou l'ajout de fonctions réalisant un besoin quelconque s'impose. Ceci peut se traduire par la modification, la suppression ou l'ajout de structures de données (évolution structurelle) et/ou par la modification des textes des entités pour les adapter aux nouveaux besoins (évolution des textes). La capacité d'un SGBD à faciliter cette évolution implique sa souplesse dans d'autres aspects tels que la modification du modèle de la structure d'un système supporté sans remettre en cause les informations déjà existantes.

2.1.2.3 Réalisation

Le mode de réalisation d'un SGBD est très important, dans la mesure où il permet l'évaluation de la complexité du SGBD en fonction des facilités offertes. Nous examinons deux aspects que nous jugeons fondamentaux :

- l'utilisation du SGBD : l'interface d'un SGBD avec les différents outils, avec l'usager ou même avec des fonctions du système d'exploitation délimite les possibilités de son utilisation, d'ajout d'autres outils ou de leur modification. L'indépendance de l'interface du SGBD de la nature des informations contenues dans la base rend le SGBD universel, ce n'est malheureusement pas souvent le cas,

- l'implantation du SGBD : certains aspects sont importants pour un SGBD : les ressources mises en oeuvre (personnel, logiciel, ressources physiques de la machine), les méthodes et techniques employées (modèle de données, enregistrement physique des données, indépendance de l'utilisation du SGBD et de l'implantation physique des données), la portabilité du SGBD pour son utilisation dans d'autres contextes, etc.

2.1.3 Critères du choix des ateliers retenus pour l'étude

Les dix dernières années ont vu fleurir un nombre important d 'AGL (HAU 81a). Certains sont développés dans un cadre industriel, d'autres constituent des expériences et des validations d'idées de chercheurs souvent accentuées sur certains aspects particuliers. Dans la littérature (HAU 81b) nous trouvons des études d'AGL basées sur un grand nombre d'aspects (motivations, modèles de cycles de vie, concepts, méthodes, outils, etc.).

Comme notre étude est orientée par l'aspect base de données, nos critères de sélection sont en conséquence :

- l'importance de l'aspect base de données dans l'atelier : la plupart des ateliers utilisent des structures de stockage qui ne sont pas vraiment des bases de données mais de simples systèmes de gestion de fichiers. Les implications de l'utilisation d'une base commencent juste à être explorées (HUF 81). Nous constatons que certains ateliers, tels que PWB/UNIX (MIT 81), utilisent le système de gestion des fichiers du système d'exploitation et n'ont donc pas de bases de données. Certains autres, tels que GANDALF (HAB 82, KAI 82a) et CDL2/LAB (BAY 81), intègrent les fonctions de manipulation de la base dans leurs outils et n'ont donc pas de base de données en tant que composante indépendante, avec un système qui s'interface avec les différents outils. Enfin d'autres ont conçu et/ou utilisé des SGBD en tant que composants indépendants et essentiels. Notre intérêt se porte sur ce dernier type d'ateliers,

- les phases du cycle de vie et le domaine d'applications supporté qui sont décrits au (2.1.2.1),

- les modèles et méthodes du développement des logiciels, détaillés au (2.1.2.2).

2.2. état de l'art

Pour plus de clarté, nous avons préféré présenter l'étude atelier par atelier en examinant les aspects précédemment fixés, au lieu d'étudier aspect par aspect tous les ateliers choisis. Une synthèse de l'ensemble des présentations est donnée en (2.3).

2.2.1 CADES : COMPUTER AIDED DESIGN EVALUATION SYSTEM

2.2.1.1. Généralités

Pour supporter le développement du système d'exploitation VME/B, pour sa gamme d'ordinateurs 2900, ICL a développé l'Atelier de Génie Logiciel CADES. CADES permet de prendre en compte toutes les caractéristiques des grands systèmes d'exploitation (complexité de la structure, grande équipe, longue vie, maintenance, ...) (SNO 81).

Parmi les objectifs de CADES on peut citer : le support d'une méthode de travail (équipe hiérarchisée) qui permet une communication et documentation formelle entre les membres de l'équipe, le support d'une méthode de développement (descendante par transformation de données), et la validation des performances (de la structure).

Toutes les phases du cycle de vie sont assistées par un ensemble d'outils intégrés autour d'une base de données (fig.2.2).

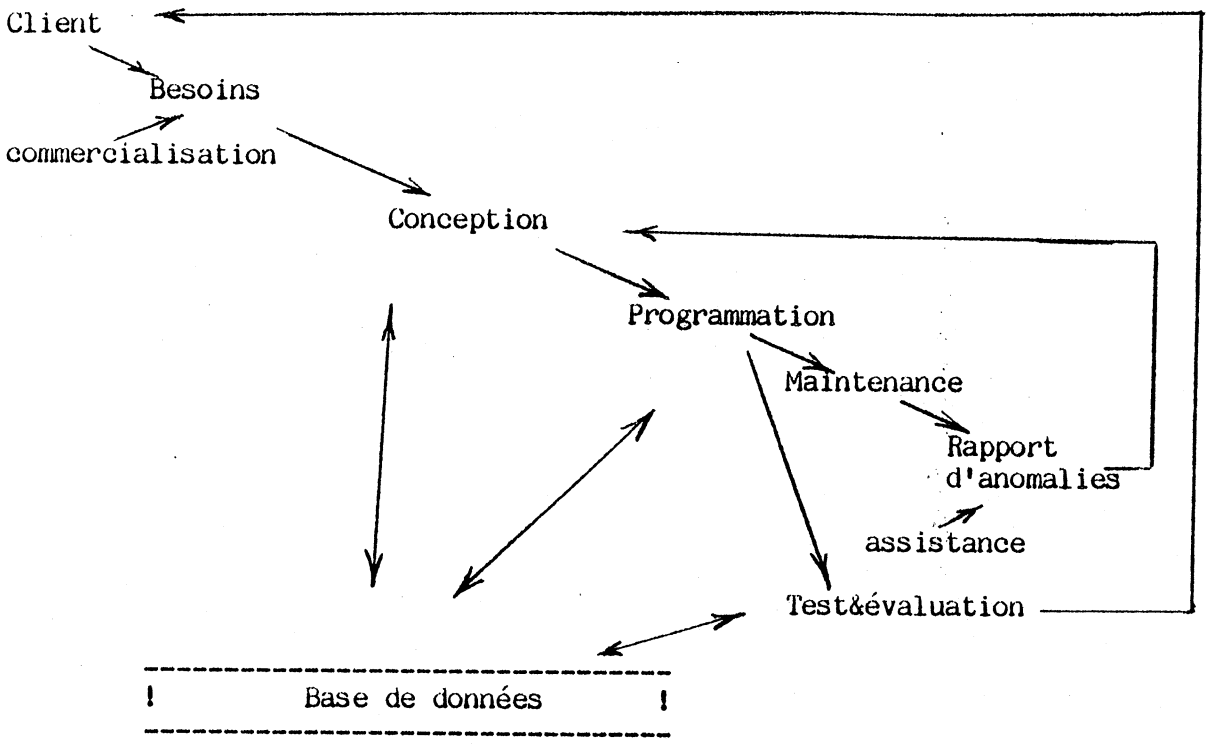


Fig.2.2 Le rôle de la base de données dans CADES.

Toutes les informations de ce cycle et d'autres (gestion de projet, etc.) sont conservées dans cette base dont le rôle est vital pour CADES aussi bien pour sa conception que pour sa réalisation. Tous les outils peuvent prendre leurs données dans la base et y déposer leurs résultats.

2.2.1.2 Expression des besoins

- Modèle et méthode de développement : le système CADES consiste en une methodologie et en un ensemble de mécanismes pour supporter cette méthodologie. La règle fondamentale de CADES est de modéliser le produit d'une manière formelle avant de l'implanter. Ce modèle sera conservé dans la base de données pour son utilisation ultérieure (implantation, modification, ...). La méthode de modélisation utilisée, appelée "modélisation structurée", est une itération descendante (McG 79).

On commence, au niveau le plus haut, par la définition générale du produit. Cette définition est alors décomposée, au niveau suivant, pour fournir des détails sur chaque composant. Chaque niveau de cette structure définit le produit total dont les détails sont précisés lorsque l'on descend dans la hiérarchie.

Chaque niveau de la structure est composé de deux types d'entités : les données (manipulées) et les "holons", processus qui contrôlent et utilisent ces données. L'analyse des données, pour choisir un modèle de logiciel à produire, a conduit au choix du modèle relationnel pour la description des entités et de leurs dépendances. La structure est formée d'arborescences de données et de holons. Chaque holon peut avoir ses propres données temporaires allouées dynamiquement ; mais son utilisation des données non temporaires, partagées ou locales au holon, est controlée via l'interaction du holon et de la hiérarchie des données. Un holon peut manipuler des données "filles" de celles manipulées par son "père" et qui sont propres au holon lui-même ou spécialement rendues disponibles par le holon auquel elles appartiennent.

Nous avons donc une hiérarchie de données, dont le niveau le plus bas définit les données au niveau d'implantation, sous la forme nécessitée pour sa manipulation par un langage de haut niveau.

Nous avons aussi la hiérarchie des holons dont les hauts niveaux définissent, sous forme de spécification formelle, la répartition des fonctions des holons entre leurs divers composants (holons) et leur utilisation de fonctions externes de même niveau d'abstraction. Le niveau le plus bas définit les fonctions au niveau d'implantation, dans une forme qui peut être compilée, comme un langage de haut niveau.

Nous avons également l'interaction holon-données, définie dans la base de données, ainsi que des informations structurelles (protection, ...).

Ces définitions et spécifications sont exprimées au moyen d'un langage de définition de systèmes (SDL), dérivé du langage S3 (S3 est un langage basé étroitement sur ALGOL 68). SDL contient de nouveaux concepts (tel que le holon, plutôt que procédure). La base de données est formée en extrayant des informations structurelles des définitions et des spécifications. Par compilation du niveau le plus bas de la hiérarchie de la structure, d'un sytème en cours de production, on aura le produit (fig.2.3, fig.2.4).

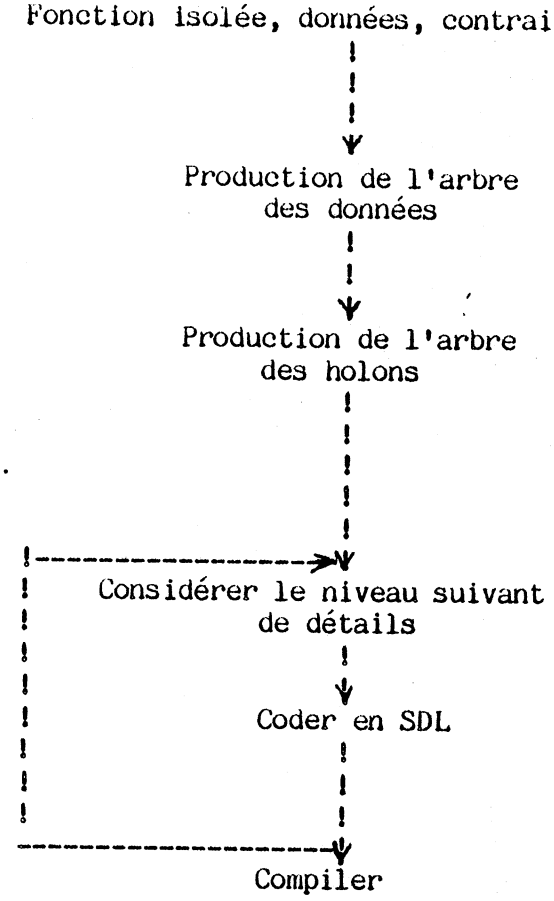


Fig.2.3 Méthode de développement d'un système sous CADES.

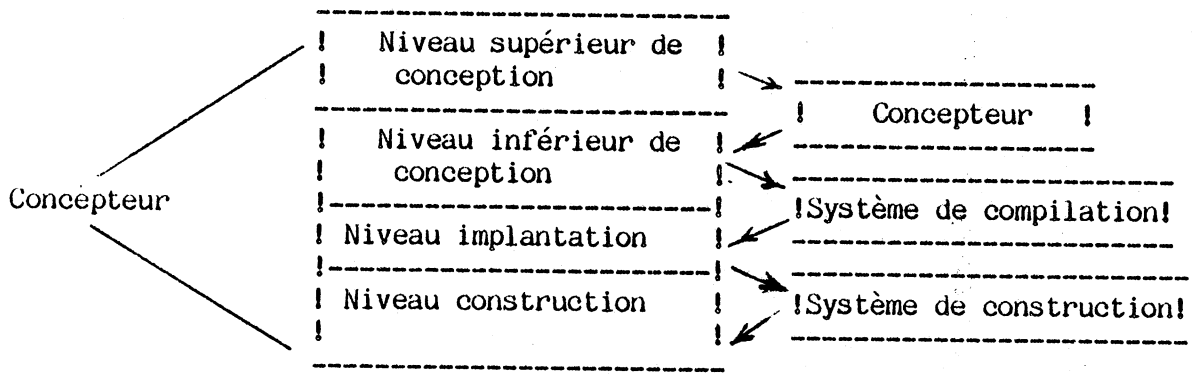


Fig.2.4 Représentation du produit et outils.

- Contraintes d'intégrité : dans le système CADES, le problème d'intégrité n'est pas traité en tant que tel, mais dilué dans les couches du logiciel construit autour de la base de données (gestion de versions multiples, vue relationnelle, ...). La possibilité de se définir dynamiquement des contraintes d'intégrité dans la base, par exemple sur la composition du produit (à partir de certaines versions) ou sur sa structure, n'est pas apparente. Il semble que ces contraintes sont figées dans le modèle et la méthode utilisés. Parmi ces contraintes, figurent celles qui consistent en la propagation des effets d'une décision (de toute opération) introduite à un niveau donné sur les niveaux inférieurs et supérieurs, en testant sa cohérence avec le reste des décisions déjà prises. Le degré de l'automatisation de cette propagation et son coût n'ont pas été exposés.

La structure hiérarchique d'un système en cours de développement permet de répartir les membres de l'équipe sur cette structure. Le projet est donc découpé en régions logiques qui constituent un mécanisme de protection puissant. Tout membre de l'équipe de développement ne peut travailler que dans la région logique à laquelle il a été affecté par l'administration du projet. On ne peut donc altérer quoi que ce soit dans une autre région. Ce mécanisme de région est transparent à l'utilisateur.

- Evolution des logiciels : parmi les caractéristiques des logiciels qui peuvent être supportés par CADES, figure la longue durée de vie. Le produit n'est pas statique, mais soumis à des développements continus par extensions et augmentations des fonctionnalités. Cette évolution est permise par le mécanisme de versions multiples.

Chaque entité et relation dans la base de données peut exister en différentes versions. Chaque fois qu'une nouvelle version est créée, elle hérite des attributs de la précédente, de sorte qu'il suffit de décrire la modification. La nouvelle version du produit est constituée des dernières versions des entités qui le composent. Chaque région logique a sa numérotation de versions indépendante des autres régions. Une nouvelle version du produit est alors créée en incorporant une version particulière de chaque région (à l'aide d'un système de construction). De cette manière un sous-système particulier du produit, correspondant à une région logique, peut être développé indépendamment et incorporé au produit quand il est prêt.

L'aspect dynamique et automatique de cette incorporation n'a pas fait l'objet d'un intérêt particulier.

2.2.1.3 Réalisation

- Utilisation : l'interface usager est un langage pour l'usager final, spécialement conçu pour représenter la structure choisie pour la description du produit. La structure est composée d'"enregistrements-cibles" et de leurs relations. Ce langage est "navigationnel", il permet de spécifier des enregistrements-cibles et de "naviguer" à travers les relations entre ces enregistrements. Le même langage est utilisé pour modifier et interroger la base, simplifiant ainsi l'interface usager.

La base s'interface également avec des outils de l'atelier, tels que le compilateur et le constructeur de sytèmes, par un logiciel qui connait la structure et le contenu de la base et présente aux outils des entrées standards. Différents processeurs (système de compilation, système de construction) préparent l'environnement nécessaire à chaque outil (compilateur, constructeur de configuration) en extrayant de la base des informations.

- Implantation : des formes primitives de bases de données ont été proposées au début du projet CADES, fournissant au concepteur un système de recherche d'informations pour la représentation de la structure du logiciel.

Ainsi, dans le premier développement de CADES, était utilisé un système hiérarchique de gestion de base de données PEARL (Part Explosion And Retrieval Language) d'ICL. La base contient seulement des informations structurelles du modèle, alors que les parties algorithmiques (code source, code objet, ...), et les documents qui decrivent le produit sont conservés sur des fichiers séparés de la base.

Le modèle relationnel a servi de base pour la conception et l'implantation d'une base de données sur une des machines 2900. Le système de base de données IDMS d'ICL a été choisi, surtout pour des raisons pratiques. Il fournit des fonctions nécessaires pour l'administration et le développement de la base. IDMS est une implantation d'un sous-ensemble des services de la base de données CODASYL (1975), proposée par le comité des langages de programmation CODASYL. Ces comités ont proposé des langages standards pour définir le contenu et l'organisation d'une base de données ainsi que les fonctions de manipulation à fournir aux programmeurs COBOL.

la vue relationnelle n'étant pas supportée par IDMS, un logiciel additionnel construit autour d'IDMS a donc été développé pour supporter cette vue relationnelle, ainsi que d'autres concepts tels que la répartition de la base en régions, la manipulation des versions multiples.

IDMS fournit beaucoup d'éléments pour l'efficacité du travail de la base : l'accès concurrent, la reprise, les fonctions d'archivage et quelques primitives essentielles pour la maintenance de la base de données. A cause des différences conceptuelles entre la base CADES et IDMS, il a fallu adapter les facilités d'IDMS aux besoins de CADES. Dans la base CADES, les informations sont réparties par régions logiques plutôt que par types d'informations. Ainsi les actions IDMS (reprise, verrouillage) qui sont orientées par le type de l'information doivent être modifiées pour fournir des actions équivalentes pour les concepts de CADES.

2.2.2 CEDAR

2.2.2.1 Généralités

CEDAR est un nouvel atelier de logiciel de "XEROX PARC", construit autour du langage MESA. C'est un environnement distribué, ayant pour support un réseau local de type ETHERNET. Les programmeurs peuvent collaborer à la production d'un logiciel en se partageant des programmes dans différents états de développement. La version récente de CEDAR (SCH 82) est essentiellement composée d'un système de gestion de programmes, le modéliseur. Le modéliseur, par des outils appropriés, gère des configurations ("systems model") et des modules en utilisant des bases de données locales et des serveurs de fichiers du réseau (fig.2.5). La phase supportée est celle de la programmation (édition, compilation, et mise au point). Nous pouvons trouver dans (HOR 79) des détails sur le contexte matériel et logique du projet.

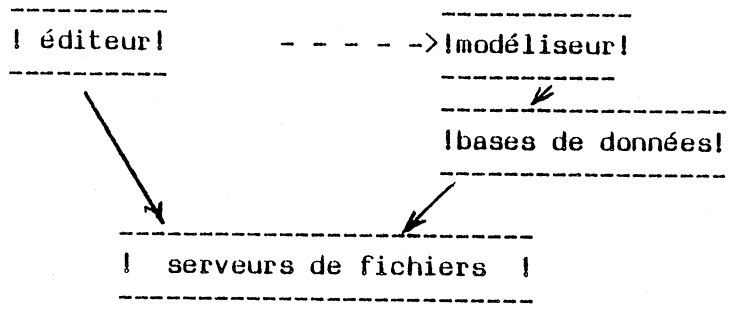


Fig.2.5 Structure générale de CEDAR.

2.2.2.2 Expression des besoins

- Modèle et méthode de développement : les programmes gérés par CEDAR sont construits selon la décomposition modulaire permise par MESA (MIC 79). On y distingue trois types de modules :

  - implantation : ce type de modules contient l'implantation des types, des variables locales ou globales, et des procédures. Ces modules peuvent être interconnectés pour former des systèmes complets. Cette interconnexion est réalisée par l'importation et l'exportation d'interfaces,

  - interface : elle est composée d'un ensemble de définitions (constantes, types, procédures) implantées par le (s) module (s) implantation qui les exporte (nt). Ce module définit un type. Pour appeler une procédure définie dans un autre module implantation (implanteur) l'appelant (client) doit importer un module interface I où est définie cette procédure. Ce module interface I doit être exporté par le module implanteur. Les deux modules implanteur et client dépendent de I (fig.2.6). Si I est recompilée, l'implanteur et le client doivent l'être aussi. La compilation d'un module dépend donc du source de ce module et des codes objets des modules interfaces que ce module importe ou exporte,

- configuration (appelée "system model") : elle est composée de modules implantations ou d'autres modules configurations. Ses composants, ainsi que leurs connexions, sont exprimés dans un langage d'interconnexion de modules (SML), dérivé de C/MESA (SCH 82). Ce module est analysé par le modéliseur qui automatise la phase du cycle de vie supportée. Le modéliseur construit une structure de données interne qui permet une recompilation optimale des modules, extrait des informations qu'il met dans la base de données locale (de l'utilisateur), localise les modules, ...

Chaque module CEDAR est représenté par un fichier source (dont le nom se termine par ".Mesa"), le compilateur CEDAR produit un fichier objet (dont le nom se termine par ".Bcd").

Chaque source ou objet peut exister en plusieurs versions. La version d'un source est exprimée par sa date de création, alors que celle d'un objet est exprimée par une estampille (qui le rend donc unique). L'estampille d'un fichier objet est une fonction du fichier source et des estampilles des fichiers objets dont il dépend.

Les relations induites par la décomposition modulaire et la méthode de développement utilisée peuvent se résumer comme suit :

- relation de dépendance (importation) d'un module d'un autre,

- type d'un module : le SML permet des expressions désignant le source d'un module. Comme toute expression est d'un certain type, le source d'un module en définit donc un. Ce type est caractérisé essentiellement par les types d'interfaces qu'il importe et qu'il exporte. Ces interfaces constituent les paramètres de ces expressions. Ces caractéristiques essentielles sont liées au source par une relation et stockées par le modéliseur dans la base locale de l'utilisateur,

- transformation d'un module par un outil (par exemple la compilation): cette relation associe à chaque module source les valeurs de ses caractéristiques et le résultat de la transformation,

- localisation : CEDAR étant distribué, et le serveur de fichiers non centralisé, il est nécessaire de pouvoir localiser chaque module (source et objet) en donnant son nom dans le réseau.

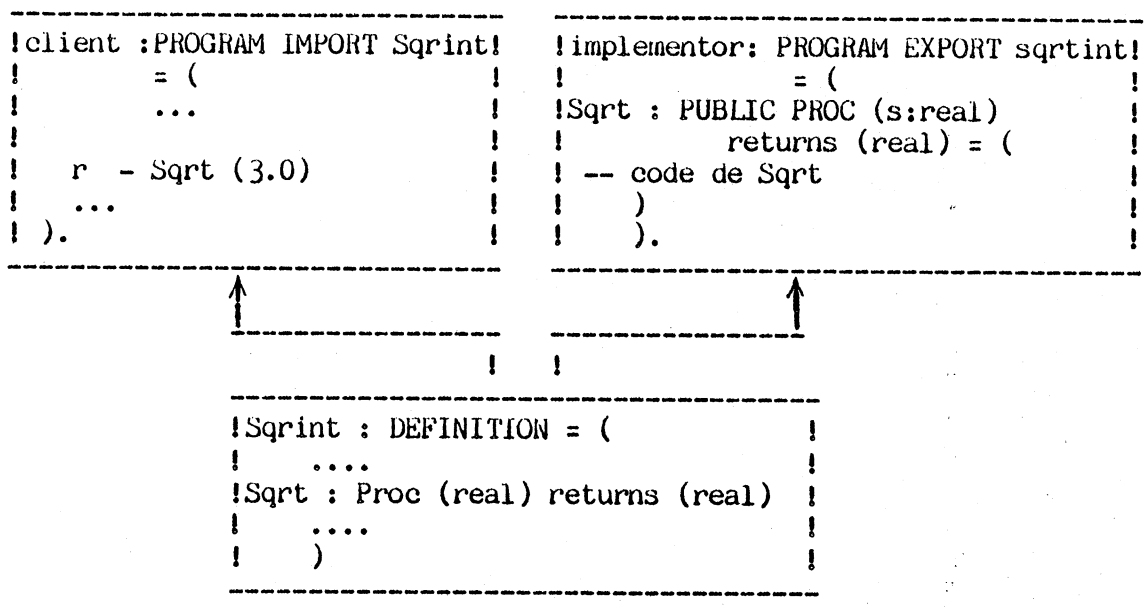


Fig.2.6 Structure modulaire d'un système sous CEDAR.

- Contraintes d'intégrité : les sources des modules et leurs objets sont des fichiers répartis sur le réseau et chaque fichier peut être manipulé individuellement sans que la cohérence des modules dépendants de lui, ni celle des configurations qui le contiennent, ne soit assurée.

C'est seulement lors de la manipulation (édition, compilation, chargement, ...) d'une configuration que le modéliseur vérifie l'intégrité de celle-ci.

L'automatisation de la gestion d'une configuration et le maintien de son intégrité sont basés sur deux mécanismes essentiels :

- les objets sont immuables : un objet a un nom unique et son contenu ne change pas depuis sa création. Il peut cependant être détruit. Chaque modification d'un objet est un autre objet identifié par sa version,

- l'estampille : la manière dont est calculée l'estampille d'un fichier objet (en fonction de son source et des estampilles des fichiers objets dont il dépend) permet de détecter, à tout instant, les inconérences dues au changement de versions de fichiers objets.

Pour accélérer son travail le modéliseur utilise les relations précédemment citées. Si elle contiennent déjà les informations qu'il désire, il les utilisera directement et terminera son travail ; sinon il calcule ces relations, les enregistre, pour une utilisation future, dans la base locale au site où il est invoqué, utilise ces informations, et termine son travail. Le rôle de ces relations étant l'accélération du travail, elles contiennent donc toutes les informations utiles sur les modules.

- Evolution des logiciels : le mécanisme de versions multiples, de l'archivage des modules indépendamment des objets qui leurs sont attachés ou qui en dépendent, et de la configuration qui regroupe tous les objets à utiliser dans un logiciel (même les documents), permet dans ce contexte réparti de faire évoluer les logiciels structurellement et textuellement. Les objets composants d'un logiciel ne sont pas figés, il suffit de donner leurs noms réseau.

2.2.2.3 Réalisation

- Utilisation : le système de gestion de programmes CEDAR peut être utilisé interactivement ou non. Les programmes qu'il gère ne peuvent être écrits que dans le langage CEDAR, car le modéliseur les analyse pour en extraire les informations des relations précédentes. C'est donc un produit local.

- Implantation : les procédures du modéliseur peuvent être classées comme suit :

  - procédures d'analyse des fichiers sources et de construction de l'arbre interne d'analyse (graphe de dépendance). Cette structure est stockée dans le fichier "objets" de chaque configuration,

  - procédures qui analysent le source CEDAR et déterminent son type,

  - procédures qui maintiennent dans la base locale une table qui contient les relations entre les fichiers sources et les fichiers objets,

  - procédures qui maintiennent une table qui donne les informations sur les fichiers de la configuration et leur localisation,

En conclusion, les objets (source, code objet, document) sont distribués sur les machines du réseau. Les informations (relations entre objets, informations sur les objets, ...) accélérant le travail du modéliseur sont stockées dans des tables qui constituent des bases de données locales. Ces bases sont accédées par les procédures spécialisées précédentes.

2.2.3 GALAAD: GESTION AUTOMATIQUE DE LOGICIEL AVEC ARCHIVAGE DES DONNEES

2.2.3.1 Généralités

GALAAD (SAL 80) est conçu pour faciliter le suivi technique et la maintenance des logiciels, notamment dans le domaine des télécommunications (GIC 79). GALAAD supporte la phase de programmation et celle de la maintenance des logiciels, en offrant un langage de commandes algorithmique, le Langage de Gestion de Programmes (LGP) (ROU 78). Ce LGP permet de formaliser toutes les procédures de gestion en manipulant des informations rangées dans une base de données (et par extension en bibliothèque pour les fichiers sources et binaires) et en offrant la possibilité d'activer de manière dynamique des processeurs spécifiques (fig.2.7).

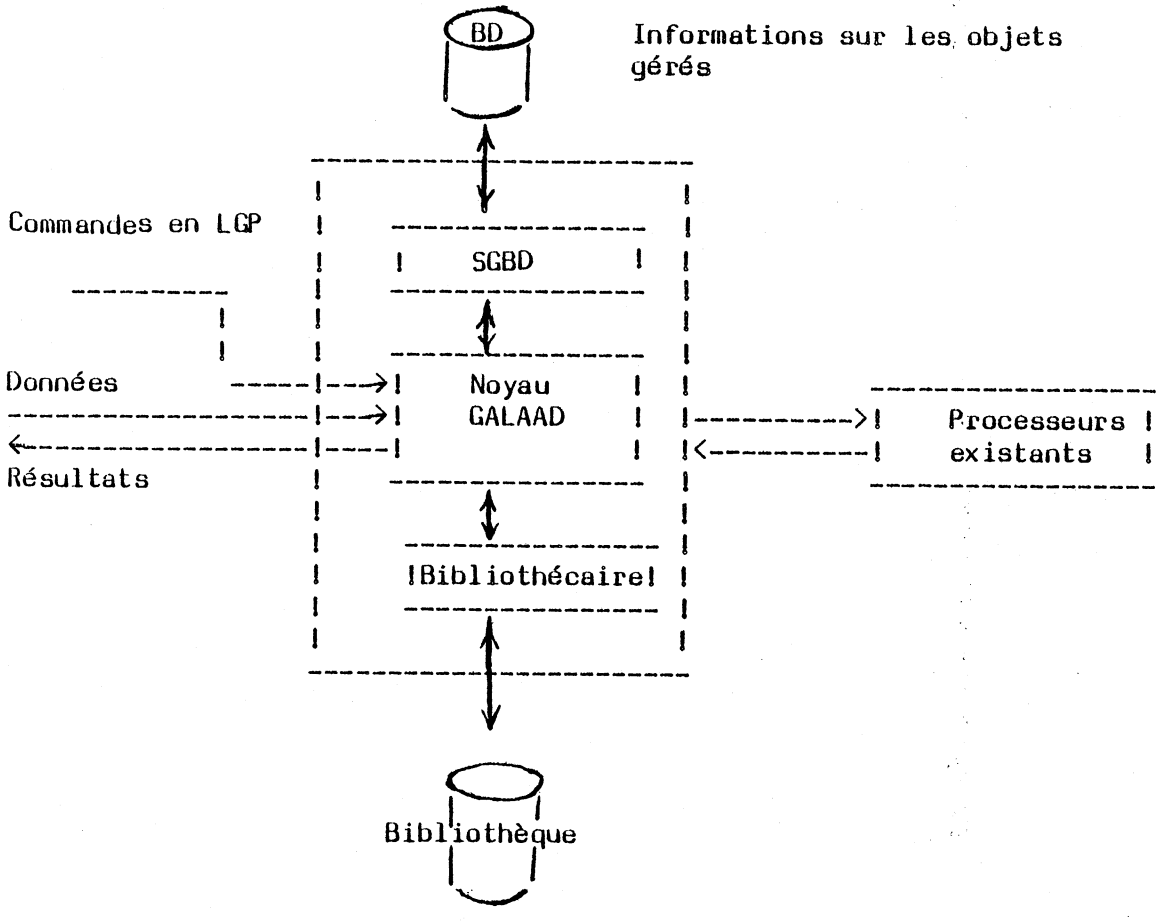


Fig.2.7 Architecture de GALAAD.

2.2.3.2 Expression des besoins

- Modèle et méthode de développement : GALAAD supporte (fig.2.8) un modèle hiérarchique pour le développement d'un projet (ROU 77). Un projet peut être décomposé récursivement en d'autres projets (sous-projets) et en entités. Une entité est un objet sur lequel porte la gestion, il pourra s'agir d'un programme source, de module exécutable ou d'un fichier, etc.

Chaque entité est composée d'un ensemble de caractéristiques, ses attributs, qui peuvent être de types différents (nombre, chaine de caractères, référence, bloc de code source - pavé source - ou binaire - pavé binaire-). L'ensemble des attributs d'une entité définit le type de cette entité. Un type est modélisé par une entité particulière, prédéfinie, la description de type. Il appartient à l'utilisateur de définir les divers types d'objets sur lesquels il souhaite faire porter la gestion.

Une entité peut comporter, à un instant donné, plusieurs variantes qui diffèrent selon les valeurs prises par leurs attributs ; ces variantes sont appelées qualifications (versions). Une qualification peut évoluer à son tour dans le temps (corrections d'anomalies, amélioration de performances, ...) ; cette évolution peut être traduite par un ensemble d'éditions (révisions).

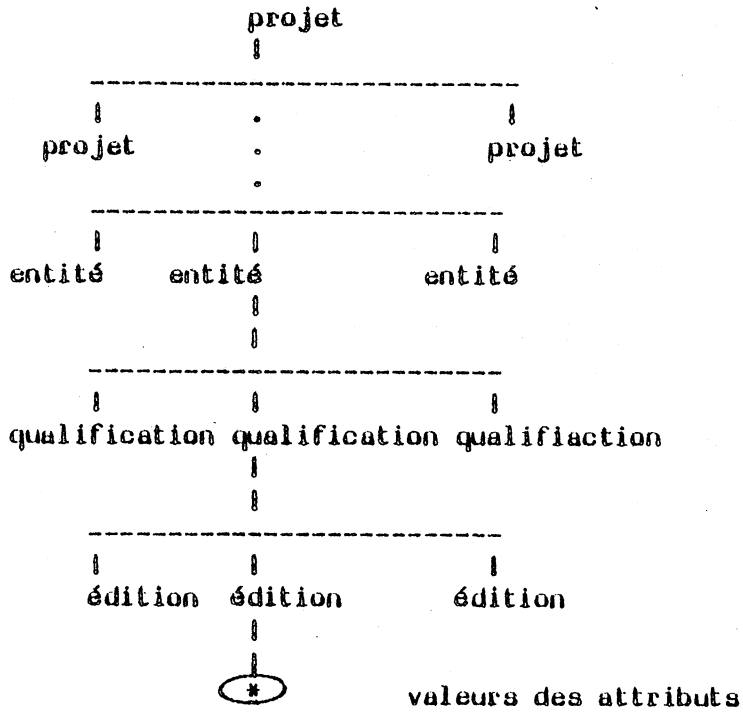


Fig.2.8 Organisation hiérarchique des logiciels sous GALAAD.

Cette organisation hiérarchique est figée dans GALAAD ; elle permet l'interprétation des noms et l'automatisation du maintien de l'intégrité de la structure d'un logiciel dans la base. En plus des relations structurelles qui décrivent la hiérarchie d'un projet, l'usager peut expliciter d'autres relations qui expriment des liens logiques entre les entités. Parmi ces relations nous citons celles qui existent entre les éditions (mère, fille, ...). Ces relations permettent le maintien d'une intégrité sémantique générale des logiciels dans la base.

- Contraintes d'intégrité : le problème des contraintes d'intégrité, dans GALAAD, se pose d'une manière simple, vu la généralité de GALAAD. Chaque application, bâtie autour de GALAAD, doit assurer elle même l'intégrité de ses données. Néanmoins GALAAD assure un minimum de cohérence dans la base, notamment en ce qui concerne :

  - la modification de type d'entité : la modification ou la suppression de type d'entité n'est prise en compte que si des contraintes prédéfinies sont satisfaites. Pour supprimer un attribut d'un type, toutes ses valeurs dans les éditions de ce type doivent être vides ; pour détruire un type, aucune entité de ce type ne doit exister, etc.

  - la modification des procédures LGP (procédures d'applications écrites sous GALAAD) : la modification d'un pavé source entraine une série de messages indiquant à l'usager quels sont les pavés sources déjà compilés et faisant référence au pavé source modifié et qui sont donc susceptibles de subir à leur tour des modifications.

La protection de la base (et de la bibliothèque) contre tout incident matériel ou logique est assurée par des mécanismes classiques de sauvegarde et de journalisation.

- Evolution des logiciels : le mécanisme de description de type d'entité, et la répercussion automatique des effets de ses modifications, permet l'évolution structurelle des entités gérées par GALAAD. Le mécanisme des qualifications et éditions permet l'évolution de leurs textes.

2.2.3.3 Réalisation

- Utilisation : pour utiliser GALAAD, il est nécessaire de développer des applications écrites en LGP pour pouvoir désigner les divers objets sur lesquels porte la gestion, accéder à leur contenu et spécifier les traitements souhaités. Grâce à l'interface avec le système d'exploitation, il est possible d'activer dynamiquement des processeurs par des procédures systèmes.

L'utilisation du SGBD nécessite donc la définition de structures de données, la saisie des données et l'exploitation des procédures de gestion. Une procédure LGP est modélisée par une entité particulière et prédéfinie.

Les logiciels à gérer peuvent être écrits dans n'importe quel langage compilable par la machine support. C'est donc un outil qui peut être utilisable dans tout environnement sur lequel il peut être porté.

- Implantation : les informations gérées par GALAAD sont de deux sortes: les informations introduites par l'utilisateur et qui font l'objet de la gestion (contenu des attributs), et les informations nécessaires à GALAAD pour assurer la gestion des informations introduites par l'usager. Les valeurs des attributs de type pavé (plusieur centaines de lignes de 80 caractères) sont stockées sur des volumes distincts de ceux sur lesquels sont stockées les autres valeurs. Les volumes supportant les pavés constituent une bibliothèque, alors que les autres volumes constituent la base de données. Les volumes de la base de données sont définis au moment de la création de la base et ne changent plus, ils doivent être montés avant toute utilisation de GALAAD ; alors que les volumes de la bibliothèque ne le sont pas nécessairement. On note également que les descriptions des types sont conservées ensemble sous un projet spécial que gère la compilateur de GALAAD.

Le LGP permet à l'utilisateur d'exprimer de manière non ambiguë les règles de gestion qu'il souhaite mettre en oeuvre. Le texte source écrit en LGP est traité par un ensemble de processeurs développés en Pascal, ainsi que les compilateurs qui lui sont associés. La nature des informations à manipuler implique l'extension de Pascal pour permettre la gestion d'une base de données. Ces extensions portent essentiellement sur :

- l'accès aux éléments d'un tableau d'une manière uniforme, qu'ils soient en mémoire principale ou en mémoire secondaire,

- la création (destruction) dynamique d'objets complexes, de types variés, d'une manière homogène sans avoir un sous-programme pour chaque type.

Cette extension de Pascal s'appelle ATOME (GIC 81). Les concepts introduits dans ATOME sont cependant ceux relatifs à une base de données de type réseau, et sont donc utilisables par une large gamme d'applications. Les programmes écrits en ATOME sont transformés en Pascal standard par une phase de précompilation.

2.2.4 SDEM/SDSS : SOFTWARE DEVELOPMENT ENGINEER'S METHODOLOGY/ SOFTWARE DEVELOPMENT SUPPORT

2.2.4.1 Généralités

Ce projet est conduit par une équipe qui, durant ces dernières années, a essayé d'appliquer plusieurs méthodes pour automatiser le développement d'applications (quelques centaines de projets pouvant atteindre des centaines de milliers d' instructions). Cette équipe a été amenées à standardiser la méthode de développement de logiciels en spécifiant SDEM, qui définit un cycle de vie des logiciels assez clair en établissant les objectifs, les contenus et les procédures des activités de chaque phase (NAK 78).

Durant ce cycle de vie, l'assistance est fournie aux usagers par deux systèmes recouvrant chacun une partie de ce cycle de vie. ISDOS (PSL/PSA) est utilisé pour les phases de planification de projet et de conception de systèmes (TEI 77). PSL/PSA assiste l'analyse, la documentation et la preparation des spécifications fonctionnelles des systèmes de traitement d'informations. Le système SDSS supporte les étapes allant de la phase de conception de modules jusqu'à la maintenance. La phase qui constitue l'interface entre les deux systèmes n'est pas assistée (spécification extérieure du système, structure de modules, ...) ; mais doit être fournie par une description manuelle à partir des produits des phases antérieures (MUR 81) (fig.2.9). Les deux systèmes sont donc indépendants, l'un ISDOS travaille sur une base de données de type CODASYL, l'autre SDSS sur une bibliothèque gérée par un bibliothècaire.

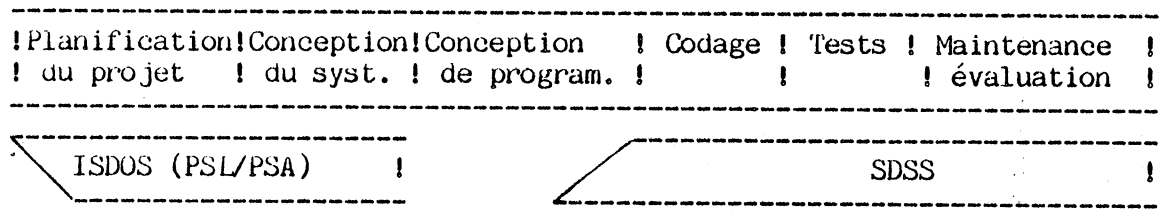


Fig.2.9 Support du cycle de vie par ISDOS et SDSS.

Pour la cohérence de l'exposé, nous examinerons ISDOS en parallèle avec SDSS.

2.2.4.2 Expression des besoins

Modèle et méthode de développement

- ISDOS : dans la conception de logiciels, assistée par ordinateur, l'objectif est de produire , comme dans le processus manuel, un rapport de définition d'un système. PSL est un langage pour la description d'un système, alors que PSA (problem statement analyser) permet de conserver une telle description dans une base de données, de la modifier incrémentalement, de faire des analyses, et de produire des rapports.

PSL est basé en premier lieu sur le modèle d'un système général et en second lieu sur la spécialisation du modèle en système d'informations. Le modèle général est relativement simple : un système consiste en un ensemble d'objets pouvant avoir des propriétés, et en un ensemble de relations établies entre ces objets. Le modèle spécialisé permet seulement l'utilisation d'un nombre limité et prédéfini d'objets (collection d'informations, processus, interfaces, ...), de propriétés, de relations (contenu dans, consiste en, utilisé par, reçoit, génère, modifie, ...), et un texte libre.

- SDSS : un système est décomposé en plusieurs sous-systèmes. Un sous-système est composé d'un ou plusieurs modules. Un module est spécifié en MDL (Module description langage). Cette spécification est analysée par le MDA (Module Description Analyser) et archivée dans la bibliothèque du projet par le bibliothécaire. Les entités manipulées sont donc essentiellement des modules qui ont un certain nombre d'attributs (identité, interface, source, ...) et de relations entre eux (utilise, ...).

Contraintes d'intégrité

- ISDOS : en plus de sa fonction de manipulation des entités de la base (objets, propriétés, et relations) PSA réalise des tests prédéfinis, tels que la détection de données non définies, la non correspondance entre l'entrée d'un processus et la sortie d'un autre, etc.

- SDSS : le MDA permet d'analyser le texte de la spécification d'un module et de confirmer si cette spécification et le texte du module sont cohérents en vérifiant les attributs décrits formellement. Il produit également quelques documents (table de références croisées entre modules, ...) qui peuvent aider à la maintenance d'un module. Cette maintenance n'est donc pas assurée automatiquement.

- Evolution des logiciels : les entités supportées par ISDOS peuvent évoluer (du point de vue structure et texte) en suivant le modèle entité-relation de PSA ; alors que l'évolution structurelle des entités dans SDSS n'est pas possible (le modèle des entités est figé dans le MDL et le MDA).

2.2.4.3 Réalisation

Utilisation

- ISDOS : l'utilisation de ISDOS ne dépend pas d'une structure particulière d'un processus de développement de système ou de format et contenu standard des documents. Il est compatible avec les procédures courantes dans la plupart des organisations qui supportent le développement et la maintenance des logiciels. Il peut donc être utilisé par un usager avec sa méthode spécifique. La disponibilité d'un méta-système dans ISDOS permet la définition de nouveaux langages de documentation ; le méta-système génère alors une nouvelle instance du système qui gère et analyse des documents écrits dans ce langage, avec des facilités similaires à celles du système original. Certains utilisateurs (LAM 83) le trouvent très utile.

- SDSS : l'utilisation se fait par un langage de commandes qui permet un accès facile aux éléments de la bibliothèque, ou par les MDL et MDA et les outils de tests. L'important dans cet aspect est sa capacité de gérer des programmes écrits en plusieurs langages.

Implantation

- ISDOS : pour manipuler les descriptions de logiciels l'usager dispose de commandes écrites dans un langage de commandes. Toutes les manipulations de la base sont écrites en PSA qui constitue le système de gestion de la base de données ; il donne une vue entité-relation alors que la base de données est du type CODSYL (fig.2.10).

- SDSS : le sous-système bibliothécaire de projet supporte la gestion unifiée des programmes sources, codes objets, données de tests, ... attachés à chaque projet. Cette bibliothèque n'est pas seulement un archivage physique d'informations, mais offre aussi quelques services utiles en adoptant une hiérarchie logique qui reflète le projet pratique (fig.2.11).

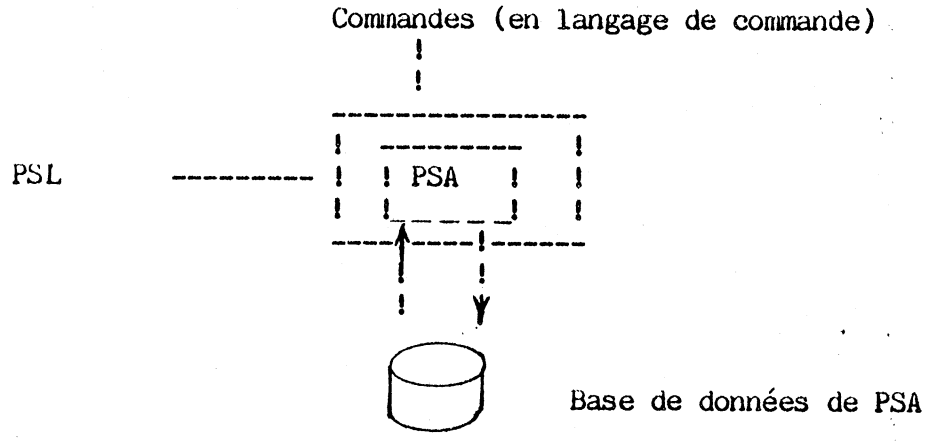


Fig.2.10 Base de données PSA.

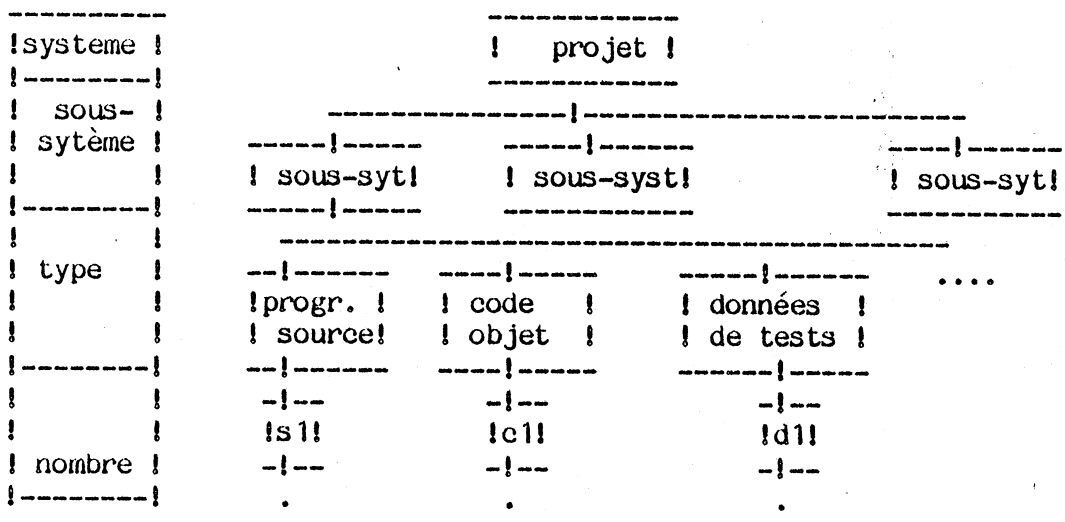


Fig.2.11 Bibliothèque de SDSS.

2.2.5 SPRAC: SYTEME DE PROGRAMMATION PAR REUTILISATION ASSISTEE DE CONNAISSANCES

2.2.5.1 Généralités

Le projet de conception de programmes assistée par ordinateur SPRAC consiste à développer un système d'aide à la conception, à la réalisation, et à la validation de logiciels (FOI 80a). La première version de SPRAC, celle qui est exposée, assiste la spécification formelle des fonctions, leur réalisation et leurs tests.

Le système (fig.2.12) s'articule autour :

- d'une base de connaissances (BDC) contenant des connaissances sur le domaine modélisé sous forme de deux classes : les fonctions et les types abstraits. Les justifications de cette base et ses objectifs sont exposés dans (FOI 80b),

- d'une base de données projet (BDP) dont le contenu permet d'offrir une assistance active au développement. Cette base joue le rôle d'un espace de travail pour l'utilisateur du système. Vide au départ elle s'enrichit au fur et à mesure de la conception, à partir de la base de connaissances et des connaissances propres à l'utilisateur en vérifiant un ensemble de règle de cohérence.

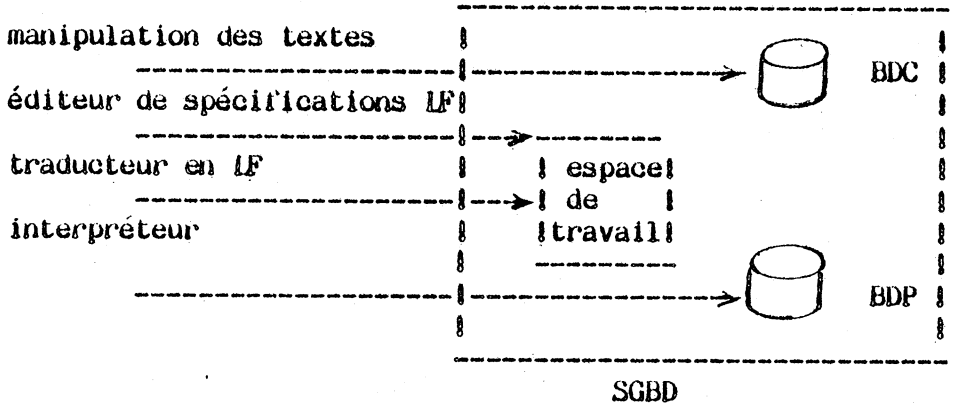


Fig.2.12 Structure simplifiée de SPRAC.

2.2.5.2 Expression des besoins

- Modèle et méthode de développement : la description des logiciels sous SPRAC est supportée par un graphe de développement. Ce graphe permet d'une part de visualiser une solution (complète ou partielle) dans son aspect structurel (liens problèmes-sous-problèmes et liens entre niveaux d'abstraction) ; d'autre part le graphe constitue un point d'ancrage à partir duquel toutes les informations relatives à la description et au développement de la solution pourront être retrouvées ou transformées par des outils adéquats. Ce graphe est défini à l'aide d'entités et de relations entre elles. Il permet un développement descendant mais (dans ses premières spécifications) ne l'impose pas.

Un mécanisme de règles de contrôle (de la structure ou de son évolution) permet de rendre conforme le développement d'un projet à une certaine méthode. Une partie de ces règles est considérée comme paramètre du système, de façon à pouvoir modifier le schéma général de description ainsi que la méthode de développement.

Des informations pouvant passer d'une base à une autre, ces bases ont donc en commun des entités et des relations : fonction (F) qui est une abstraction ; algorithme (A), représentation algorithmique de la fonction ; module (M), concrétisation d'algorithme ; type abstrait (ADT), ensemble des fonctions définissant un type abstrait de données ; représentation des ADT (RADT), ensemble d'algorithmes pour les fonctions de l'ADT, et concrétisation des ADT, ensemble de concrétisation des RADT.

Ces entités sont reliées par des relations structurelles qui définissent des liens directs entre elles(fig.2.13) et des relations qui permettent de définir les attributs des entités. Ces attributs (domaine, définitions, ...) sont manipulables par des outils du système (traducteur, interpréteur, ...).

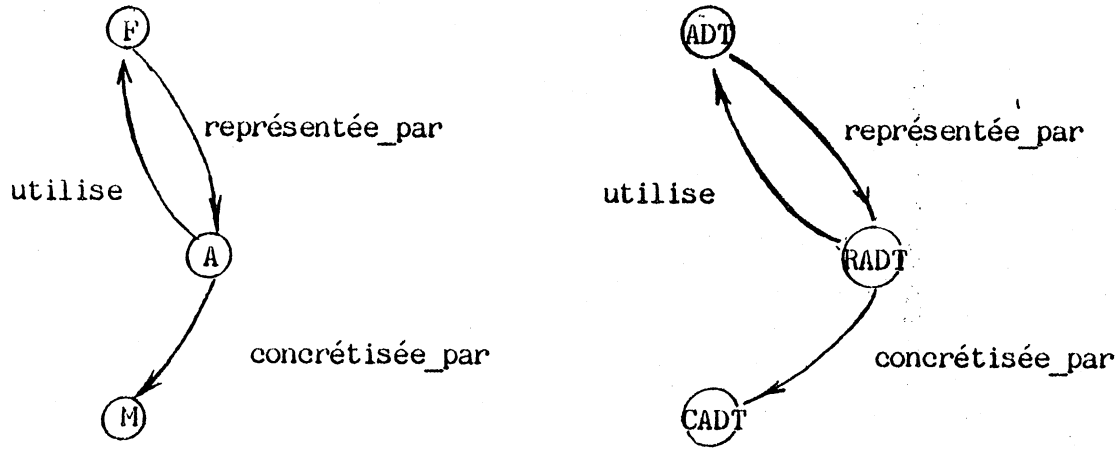


Fig.2.13 Exemple de relations structurelles.

La BDP, qui a d'autres fonctions que le stockage de ces entités, a besoin d'autres structures répondant chacune à une fonction :

- Le graphe de développement, constitué des entités citées et de leurs relations, supporte la fonction de description,

- la version et l'agenda courants supportent l'aide au développement qui consiste à guider l'usager, à tout instant du développement, en lui indiquant les actions à faire. Le système doit lui présenter des informations pertinentes à l'étape de travail courante,

- l'historique supporte le suivi du développement. Des solutions et des choix antérieurs sont conservés par le sytème dans la BDP, de telle façon que l'usager puisse reconsidérer ces solutions, revenir sur ces choix et développer sa solution dans des voies différentes,

- les règles de cohérence supportent le contrôle du développement, elles sont :

  - statiques (règles de la structure du graphe de développemnt) : chaque état de la base doit être cohérent en lui même et conforme à un certain schéma conceptuel descriptif préétabli. Elles ne sont pas modifiables,

  - dynamique : le passage d'un état cohérent de la BDP à un autre état cohérent est soumis à des règles de transition qui permettent de rendre conforme le développement d'un projet à une certaine méthode.

- Contraintes d'intégrité : un des objectifs de SPRAC était de supporter des projets aux schémas de description et méthodes de développement variés. SPRAC devrait donc avoir comme paramètres ces deux aspects. Les règles de contrôle du développement, préservant son intégrité sont donc aussi des paramètres fournis par l'usager. Mais dans sa première version SPRAC a figé la méthode de développement par la décomposition descendante. Les contraintes d'intégrité dynamiques sont maintenues par un module qui manipule le graphe de développement et les spécifications. Elle sont donc figées.

Par exemple, la description d'un module (M) qui concrétise une fonction nécessite la description de cette fonction et sa représentation. L'intégration d'une spécification dans une version nécessite le test de sa compatibilité, la mise à jour de l'historique et la génération d'une nouvelle version.

- Evolution des logiciels : le modèle relationnel des deux bases (BDC) et (BDP) facilite l'évolution structurelle des logiciels par la modification des schémas des relations touchées par cette évolution. Cette facilité dépend fortement du (des) SGBD utilisé(s) pour la gestion des deux bases. L'évolution des textes sources des entités est supportée par le mécanisme des versions multiples.

2.2.5.3 Réalisation

- **Utilisation** : l'utilisateur dialogue avec SPRAC au travers du Superviseur de la base qui utilise un menu hiérarchisé des différentes tâches que l'usager peut accomplir ou des services offerts.

Le choix de la base de connaissances se traduit par la mise à la disposition de l'usager des connaissances de la BDC. Elle peut être accédée à l'aide d'un langage d'interrogation spécialisé, qui consiste en un ensemble de questions prédéfinies. Le résultat de l'interrogation est rangé dans un espace de travail temporaire (qui constitue le brouillon de l'usager) manipulable par les outils associés à la BDP ou par des outils généraux (éditeurs syntaxiques, éditeurs de textes, ...).

Le choix de la BDP se traduit par la possibilité de consultation (édition de l'état : graphe du développement ; édition d'objets : spécifications, algorithmes, ... ; édition de l'historique ; ...), de modification (ajout d'un objet à une version, modification de version, ...) et de changement (changement de versions, combinaison de versions, ...).

Le contenu des attributs de définition des entités étant analysé, le mécanisme de gestion de la base de données dépend fortement des languages de spécification des fonction (LF) (FOI 82b), de description algorithmique (LA) et de représentation (LM).

- **Implantation** : l'environnement logique de SPRAC étant MULTICS, et ayant choisi le modèle relationnel pour les deux bases, l'utilisation de MRDS (MULTICS Relational Data Store) disponible sous MULTICS était souhaitable. Une première maquette a été réalisée en utilisant MRDS (MRD 81) comme système de gestion de base de données.

De nombreuses difficultés pratiques ont été dues à l'inadaptation de MRDS (comme d'ailleurs de tous les sytèmes de gestion de bases de données actuels) à ce type d'applications (pour plus de détails voir (FOI 82a)). Nous citons :

- MRDS permet de gérer des bases à schéma conceptuel statique, c.à.d des objets à structure statique et rend difficile la gestion d'objets à structure dynamique (modification structurelle de leurs types),

- les choix effectués (caractères de contrôle du langage d'interrogation, format des éditions des réponses par tableaux relationnels, ...) s'ils n'intérdisent pas l'utilisation de MRDS dans ce contexte (texte indenté sur plusieurs lignes, structure de données variées, ...) imposent des étapes préalables ou suivantes à l'utilisation de MRDS. Un texte de définition (qui est un attribut d'une relation) doit, par exemple, être transformé par suppression de l'indentation, des caractères retour_chariot (interprétés comme fin de n-uplet).

Dans le cadre d'une expérimentation limitée d'outils de gestion à la fois plus sophistiqués et plus simples, il a été envisagé d'utiliser des bases de données PROLOG ou LISP, vu que le langage support est LISP. Le schéma conceptuel déjà défini et le langage d'interrogation associé sont conservés.

2.2.6 SSP : SYSTEME SUPPORT DE PROGRAMMATION

2.2.6.1 Généralités

S'inscrivant dans le cadre des Ateliers de Génie Logiciel, SSP est orienté vers des applications de type "contrôle et commande de processus en temps réel". Il peut être utilisé lors de développement de grands projets en langage LTR sur MITRA 125.

Les phases du cycle de vie supportées par SSP sont celles de la conception détaillée, de codage, de tests et d'intégration de programmes. SSP est composé d'un ensemble d'outils qui s'articulent autour d'une base de données relationnelle, conçue et réalisée pour cet effet (LAP 81) (fig.2.14).

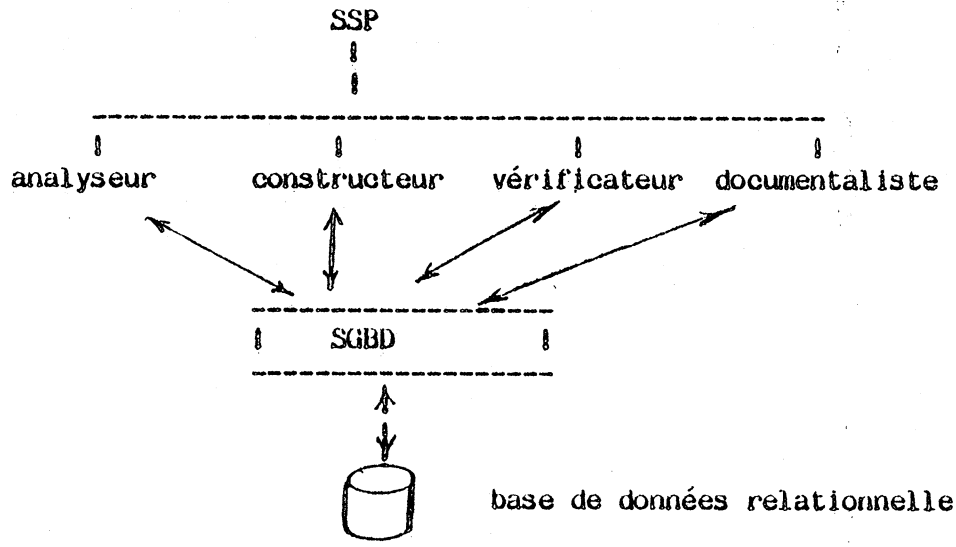


Fig.2.14 Structure de l'atelier SSP.

2.2.6.2 Expression des besoins

- Modèle et méthode de développement : un projet développé sous SSP doit appliquer la méthode de conception par hiérarchie de machines abstraites dans un environnement temps partagé. Il doit se décomposer en un ensemble de programmes, de processus, de machines abstraites et de fonctions dont la combinaison fournit une solution au problème posé (fig.2.15).

Un ensemble d'objets élémentaires a été prédéfini (processus, ressources, ...). Les objets élémentaires qui prennent leurs valeurs dans un même ensemble de définition ont été regroupés dans une même classe (Projet, programme, ...). Tout ensemble de classes ayant une même sémantique constitue un regroupement (projet, programmes, processus, machine, fonction interne, et fonction externe).

Des relations entre les classes d'objets (voir par exemple fig.2.16) sont exprimées pour conserver des traces de la spécification et effectuer des vérifications. D'une part il s'agit d'exprimer les relations entre entités d'un groupement qui traduisent sa structure syntaxique, et d'autre part d'exprimer celles qui traduisent la cohérence du groupement. Les entités et relations sont spécifiées dans un langage de spécification de conception temps réel, SPECTRE (AZZ 80). L'analyseur de projet (MOI 81) valide et conserve cette spécification dans la base de données, selon le schéma, prédéfini de l'outil. La structure d'un logiciel sous SSP est schématisée par la figure (fig.2.15).

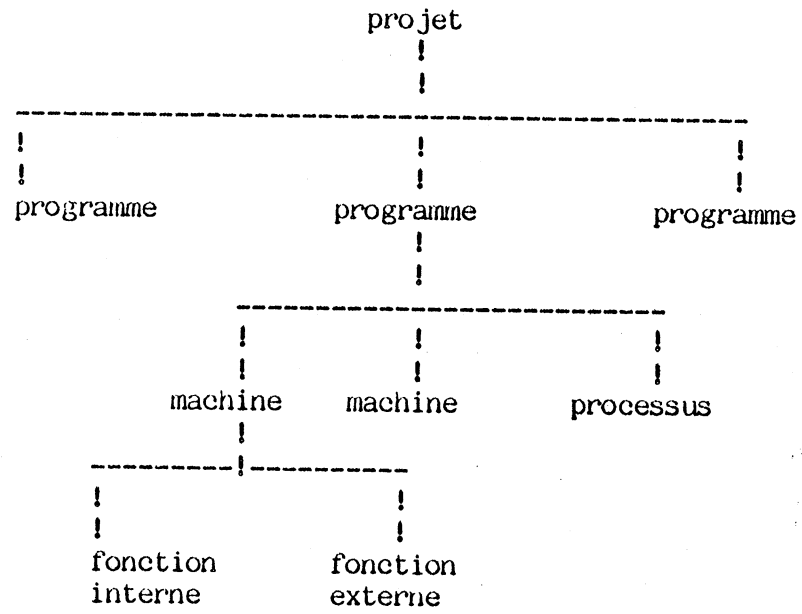


Fig.2.15 Organisation d'un système sous SSP.

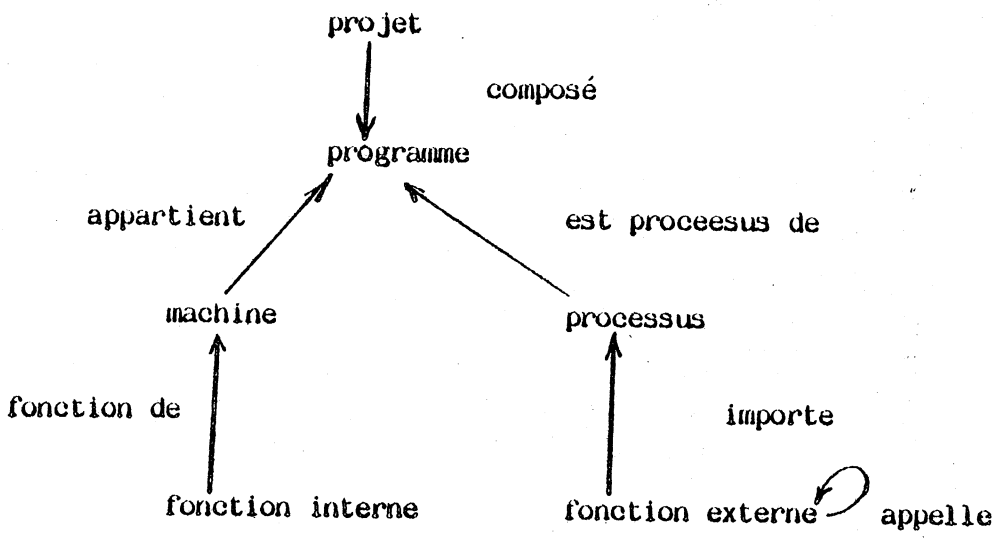


Fig.2.16a Exemple de relations structurelles.

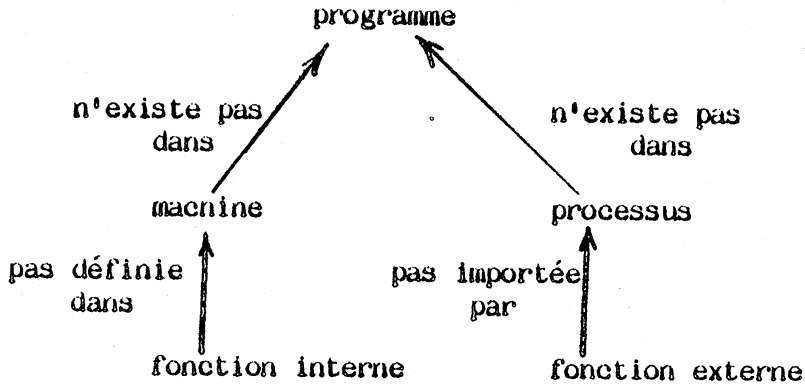


Fig.2.16b Exemple de relation de cohérence .

- Contraintes d'intégrité : les informations envoyées à la base par les outils (données codées déduites des éléments de spécification ou de programmation, résultat de tests, ...) sont conformes aux règles définies lors de la mise en place des outils, avant leur transfert dans la base.

Cette analyse consiste à vérifier la correction de la syntaxe de l'entité en cours de description. De plus quelques vérifications de cohérence sont effectuées sur l'entité (cf. fig.2.16).

Chaque outil est responsable d'un ensemble d'informations créées par lui et qu'il peut partager avec d'autres ; de ce fait la base de données est invisible à l'usager. La protection de l'information repose sur un mécanisme de droit d'accès par les outils aux informations de la base. Des mécanismes de journalisation et de sauvegarde sont utilisés.

- Evolution des logiciels : l'expression de nouvelles exigences (évolution structurelle des entités) d'un utilisateur de SSP peut amener le responsable de ce dernier à modifier le rôle d'un outil. Nous pouvons distinguer deux types de modifications :

  - La modification impliquant un seul outil s'exprime par la définition de nouvelles relations, modification ou suppression de relations dans le schéma conceptuel associé à l'outil. Le SGBD devra, en fonction de ces modifications, définir, modifier ou supprimer des relations dans le schéma conceptuel existant associé à l'outil,

  - la modification impliquant plusieurs outils (fournisseurs) : L'outil à modifier (demandeur) s'exprime par l'utilisation des relations existantes dans des schémas conceptuels associés à d'autres outils. L'expression des nouveaux besoins se traduit par la modification du schéma conceptuel initialement défini par le réalisateur de l'outil (demandeur) et par l'introduction de nouvelles classes d'objets et de liaisons, dans les schémas d'autres outils (les fournisseurs), auxquels il fait référence.

Le modèle relationnel utilisé permet de prendre facilement en compte l'intégration d'un nouvel outil et les changements apportés aux fonctions d'un outil existant.

2.2.6.3 Réalisation

- Utilisation : la base de données est invisible aux usagers, elle ne peut être utilisée qu'à travers des outils. Le SGBD offre une interface souple (primitives de manipulation de schéma conceptuel indépendantes des outils), ce qui permet la modification ou l'introduction de nouveaux outils.

- Implantation : le SGBD de SSP est relationnel. Les attributs des relations sont des chaines élémentaires de caractères ou d'entiers (identificateurs codés). Les grandes chaines (commentaires, documents, textes sources, ...) ne sont pas stockés dans la base.

SSP est implanté sur MITRA 125 et sous MMT2, le système de gestion de fichiers FMS2 assure l'ensemble des services permettant la création et l'exploitation des données sur MITRA 125.

L'espace des données est découpé en pages, de façon à amener en mémoire centrale une page lors d'une entrée sortie. Les données d'une relation sont implantées sur une ou plusieurs pages. Deux types de fichiers sont utilisés: les fichiers spécifiques au SGBD (éléments descriptifs d'une relation, domaine des données, ...) et les fichiers des outils (textes, textes des identificateurs codés, ...).

2.3. Conclusion

Dans ce qui suit, nous concluons sur les besoins d'un AGL, posons les problèmes d'utilisation de SGBD existants, et exposons les tendances principales des solutions. Nos propres propositions seront détaillées en (7.2).

2.3.1 Besoins

2.3.1.1 Modèles de logiciels

Nous avons utilisé le terme de modèle de logiciel pour la description (définition) de l'ensemble des types d'entités qui le composent, de leurs propriétés, et des relations entre elles. Le rôle du modèle est par conséquent très important. Il ne se limite pas à la description des entités dans un formalisme compréhensible par les utilisateurs, mais détermine les capacités d'un atelier et les mécanismes mis en oeuvre pour supporter le développement d'un logiciel. Dans ce qui suit nous résumons le rôle d'un modèle dans un atelier.

- Description (des entités et relations) : dans tous les ateliers étudiés nous constatons que les logiciels sont décrits dans des langages appropriés. Dans CADES, le langage SDL (System Definition Langage) est utilisé aussi bien comme langage de description de système (conception détaillée) que comme langage de programmation, ce qui a pour avantage de faciliter le développement de logiciels.

Dans certains autres ateliers, deux langages sont utilisés pour supporter la conception et la programmation. Dans GALAAD, le langage LGP (Langage de Gestion de Programmes) permet la description des logiciels alors que n'importe quel autre langage de programmation peut être employé. Dans SDEM/SDSS le langage PSL (Problem Statement Langage) permet la conception générale d'un logiciel, alors que le langage MDL (Module Description Langage) est utilisé pour la conception détaillée (description d'attributs d'un module) et différents langages de programmation sont permis. Le langage SPECTRE de SSP permet l'expression de cette description alors que le langage LTR est utilisé pour la programmation. Dans CEDAR, cet aspect se manifeste par le langage SML qui permet la description des différents composants d'une configuration, et leurs connexions.

La compilation des définitions et spécifications, permet d'extraire des informations structurelles concernant les entités et les relations et de les stocker dans la base de données.

- **Propriétés** : nous résumons les propriétés internes à une entité E, et leurs effets sur les entités en relations avec E. Les propriétés internes permettent le maintien de la cohérence de l'entité elle même, alors que leurs effets permettent le maintien de la cohérence des entités entre elles. Ces deux aspects sont présents dans la plupart des ateliers, bien que leurs description formelle dans les modèles soit peu apparente.

Dans CEDAR, les propriétés internes d'une entité sont implicites. Les entités dont les noms se terminent par ".Mesa" peuvent être éditées, compilées, et connectées. Les entités dont les noms se terminent par ".Bcd" ne peuvent être que chargées. Les effets de ces propriétés sont moins apparents ; ils permettent, par exemple, de recompiler les codes objets qu'une modification d'entité a périmés. Il s'agit du maintien de l'intégrité d'une configuration.

Dans GALAAD, la description du type des entités définit leurs propriétés internes : le type de leurs attributs, et par conséquent l'ensemble des opérations qui peuvent leur être appliquées. La cohérence interne d'une entité est assurée par l'atelier en vérifiant la présence de tous les attributs, leurs types, et les opérations permises. Les effets de ces propriétés sont moins formels, ils se manifestent sous forme de règles conservant la cohérence des entités entre elles. Par exemple, pour détruire une description de type d'entités, aucune entité de ce type ne doit exister.

Dans SPRAC, les propriétés sont plus apparentes et plus formelles. Les propriétés internes des entités sont définies par leurs types (fonction, algorithme, ou module). Chacune de ces entités étant décrite dans un langage approprié, ses propriétés internes sont bien définies. L'ensemble des effets de ces propriétés est exprimé sous forme de règles de contrôle de la structure du logiciel et de son évolution qui permettent de rendre conforme le développement de ce logiciel à une certaine méthode.

Le manque d'intégration de la spécification des propriétés et de leurs effets avec la description des logiciels fait que les propriétés (et leurs effets) ne sont pas souvent formellement et clairement définies. Ce manque d'intégration est dû à l'incapacité des langages de définition (description) d'exprimer des sémantiques complexes.

En résumé, chacun des ateliers étudiés gère des logiciels décrits selon un modèle fixe. Les ateliers sont donc fortement caractérisés par leurs modèles de logiciels. Les entités des ateliers sont très complexes. Souvent leurs attributs ne sont pas de simples chaines de caractères ou de chiffres, mais plutôt des objets structurés, ayant une sémantique complexe. Dans SPRAC par exemple, l'entité fonction a un attribut qui est la définition axiomatique de la fonction, exprimée en LF. Le compilateur de ce langage et l'interpréteur sont deux outils habilités à manipuler cette structure et à assurer son intégrité.

2.3.1.2 Méthode de développement

Nous avons également utilisé le terme de méthode de développement de logiciels pour désigner l'ensemble des règles qui dirigent la décomposition, la composition, et l'évolution des logiciels. Dans ce qui suit nous résumons la présence des trois aspects de la méthode de développement, dans les ateliers étudiés.

- Décomposition : la décomposition d'un logiciel en plusieurs composants est très apparente dans les ateliers étudiés. Elle est presque complètement spécifiée dans les langages cités précédemment, dans la description des logiciels dans les ateliers.

Dans CADES, un logiciel est décomposé selon la méthode appelée "data driven" qui consiste en la décomposition de la structure de données en niveaux de détails croissants, et en la spécification des procédures qui réalisent la transformation de ces données de niveau en niveau. Un résumé de cette méthode est donné dans (PRA 81). Un logiciel est donc composé de deux arborescences : arborescence des données et arborescence des procédures (cf. 2.2.1.2).

Dans GALAAD, SDSS, et SSP la décomposition descendante est utilisée. Elle consiste en la décomposition récursive d'un logiciel en sous logiciels, plus simples, jusqu'à ce que l'on ait des programmes jugés faciles à implanter.

- Composition : la méthode de composition d'un logiciel est loin d'être aussi claire que la méthode de décomposition. Elle est perçue de façon imprécise, à travers un ensemble hétérogène de mécanismes, et elle n'a pas fait l'objet d'études spécifiques.

Le système CEDAR est une exception. La composition de logiciels est spécifiée clairement et formellement dans un langage de connexion SML, dérivé de C/MESA (SCH 82). SML permet de construire un logiciel à partir de ses composants tout en assurant son intégrité dans le temps.

- Evolution : comme les modèles et méthodes de développement des logiciels sont figés dans les ateliers, faciliter l'évolution des logiciels est primordial. Nous constatons la présence de mécanismes qui permettent l'évolution structurelle des logiciels. Dans GALAAD, par exemple, le mécanisme de description de types d'entités permet l'évolution structurelle des logiciels, alors que dans les ateliers ayant un modèle décrit par un formalisme relationnel, cette évolution structurelle est réalisée par la modification du modèle lui même.

L'évolution des textes est souvent réalisée par des mécanismes de versions multiples (CADES, CEDAR, GALAAD, et SPRAC). Ces mécanismes ne sont pas clairement définis, bien que la modification des textes, les effets de bord induits, et les mécanismes qui automatisent le traitement de ces effets soient des aspects assez importants dans un atelier de logiciel.

A part CEDAR, aucun autre atelier n'a donné à cet aspect l'intérêt qu'il mérite.

2.3.1.3 Dépendances entre modèles et méthodes

De l'étude précédente, nous constatons que la méthode de développement d'un logiciel n'est pas totalement indépendante du modèle de ce logiciel, et inversement. Les modèles relationnels sont très utilisés parce qu'ils permettent toutes sortes de décompositions (descendante, ascendante, ...).

Cette dépendance est plus apparente encore, quand nous remarquons qu'un même langage est utilisé aussi bien pour la description d'un logiciel (la description des attributs de ses entités, ...) que pour sa décomposition (liaisons entre entités). Ceci prouve que le modèle et la méthode de développement d'un logiciel peuvent facilement aller de pair.

L'expression des propriétés des entités est en fait l'établissement de la méthode par laquelle ces entités peuvent se combiner et évoluer. Pour permettre le développement d'un logiciel selon une méthode donnée, le modèle doit permettre l'expression des ces propriétés.

Bien que ces deux besoins (modèle et méthode de développement) paraissent étroitement liés, leur intégration (fusion) est uniquement réalisée dans CADES.

2.3.2 Problèmes posés par l'utilisation des SGBD actuels

Les problèmes d'utilisation des SGBD actuels se posent essentiellement dans la mise en oeuvre des besoins précédemment cités :

- Modèle de logiciel : certains projets, SPRAC par exemple, ont tenté de construire des SGBD généraux à partir de SGBD existants. Le modèle des logiciels était pris comme paramètre ; certains autres l'ont figé dès le départ. La difficulté de modification des modèles encourage, sinon impose, cette issue.

Les modèles des logiciels sont, dans la plupart des ateliers, décrits dans le formalisme relationnel, alors que les ateliers disposent (ou jugent mieux adaptés) de SGBD non relationnels.

Les attributs des entités gérées par des SGBD sont élémentaires, alors que ceux des entités logicielles sont complexes et ont une sémantique plus forte,

- méthode de développement : nous avons vu (cf. 2.1.2.2) que les Contraintes d'Intégrité Sémantiques constituent essentiellement la méthode de développement. Bien que les entités gérées par les SGBD classiques soient simples, l'expression et le maintien de leurs CIS est difficile. Dans un domaine comme celui du génie logiciel, où les entités et leur sémantique sont complexes, l'expression et le maintien des contraintes (conformant le développement à une certaine méthode) est, inévitablement, une tâche délicate.

Si pratiquement toutes les CIS des SGBD classiques sont sous forme d'assertions logiques, celles nécessaires au domaine du génie logiciel sont pratiquement du type "déclencheur" (mécanisme équivalent au "on condition" du langage de programmation PL/1). Elles s'expriment par de grands algorithmes compliqués. Les contraintes décrites dans le schéma (3.2.2) en sont un exemple concret.

Un but fondamental du génie logiciel est l'automatisation de tous les traitements qui peuvent l'être, pour alléger la charge de l'utilisateur, et réduire ainsi considérablement le taux d'erreurs. Les CIS constituent justement un moyen de réaliser cette automatisation. Le nombre de ces contraintes est donc considérable (cf. 3.2.2), ce qui n'est pas nécessairement le cas dans les applications que les SGBD actuels supportent bien.

Un aspect particulier de ces contraintes est qu'elles peuvent être des attributs d'une entité. Elles ne sont pas mises en oeuvre uniquement lors de la mise à jour d'un attribut, mais aussi à chaque création d'un attribut. Le fait que ces contraintes soient des attributs, fait qu'elles sont souvent soumises à d'autres contraintes (ce qui n'est pas le cas dans les SGBD actuels). La modification de ces attributs risque d'engendrer des modifications récursives d'autres contraintes et le contrôle de ces répercussions devient difficile.

Si on trouve dans (FER 83) que le contrôle de la récursion des "déclencheurs" est un problème ouvert dans les SGBD relationnels, ce contrôle se complique d'avantage en tenant compte des caractéristiques de CIS dans le génie logiciel.

2.3.3 Solutions actuelles

Pour satisfaire les besoins cités précédemment, certains ateliers (CADES, SPRAC, ...) tentent d'utiliser, malgré les problèmes qui s'y opposent, des bases de données existantes. D'autres (GALAAD, SSP, ...) préfèrent développer des SGBD spécialisés dans le domaine du génie logiciel. Dans les deux cas, plusieurs tendances se font remarquer :

- le modèle du logiciel et la méthode de développement sont figés (sauf dans SPRAC). De ce fait, les contraintes d'intégrité sont prédéfinies et leur maintien est assuré par les divers outils habilités à les manipuler. Les outils de gestion de versions multiples, le langage SDL de CADES, l'analyseur de projets de SPS sont des exemples. Pour amortir l'effet de ce gel, les ateliers permettent la modification (l'évolution) de ces modèles par des mécanismes divers : modification des descriptions des types d'entités dans GALAAD, modification des sous schémas des outils dans SPS, versions d'entités dans CADES, ...

- Le modèle de logiciel étant souvent décrit dans le formalisme relationnel, des couches logicielles sont construites autour des SGBD choisis (ceux-ci ne sont pas souvent relationnels) pour implanter cette vue, ainsi que le contrôle de certaines contraintes d'intégrité de leurs logiciels (règles de la méthode de développement, contrôles des versions multiples, ...). Dans CADES par exemple, des couches de logiciels sont construites autour d'IDMS pour implanter la vue relationnelle, gérer les versions multiples, conformer le développement à la méthode qu'il supporte et même de modifier certaines fonctions de base d'IDMS pour l'adapter aux concepts de l'atelier.

- Dans les ateliers étudiés, sauf dans SPRAC, les contenus des attributs structurés (complexes) sont stockés hors de la base où on ne garde que les attributs élémentaires (ceux supportés par les SGBD existants). Ceci est, à notre sens, contraire au concept même de base de données. La dispersion d'informations sur divers supports ne fait qu'augmenter le risque de leur incohérence.

- Certains ateliers utilisent plusieurs bases pour contenir des informations de natures différentes. Dans SDEM/SDSS la base de PSA à été utilisée pour contenir les documents des spécifications générales et la bibliothèque pour contenir les programmes. Dans SPRAC, une base est utilisée pour contenir les connaissances dans le domaines des applications supportées et une autre pour les applications elles-mêmes.

3 MODELE DE DONNEES POUR LA DESCRIPTION DE SYSTEMES MODULAIRES DANS ADELE

Dans ce chapitre nous commençons par l'introduction des notions fondamentales de la décomposition modulaire de systèmes, utilisée dans ADELE. Nous terminons par la définition des types d'objets et de leurs associations nécessaire à cette décomposition.

3.1 Décomposition modulaire d'un système : structures et propriétés

3.1.1 Généralités

Les programmes gérés par ADELE sont construits selon une décomposition modulaire. Les principes de la décomposition modulaire des programmes ont été largement développés dans la littérature (voir par exemple une synthèse et une bibliographie dans (KRA 81)), et ses avantages confirmés par l'expérience (CAS 80, HOR 79, LAU 79).

Pour exploiter pleinement ces avantages, il est néanmoins nécessaire que la notion de module soit partie intégrante aussi bien du langage utilisé que de son environnement. Ce principe est encore loin d'être appliqué dans la pratique, ne serait ce qu'en raison du fait que les langages usuels ne sont pas réellement modulaires. Dans le cas d'ADELE, par exemple, le langage Pascal nécessite des extensions pour supporter une forme quelconque de modularité ; nous pouvons citer en exemple les propositions faites dans (GHO 81), et le Pascal ADELE défini dans (EST 83b).

Nous supposons donc admise l'idée que la modularité est une notion centrale dans l'organisation d'un atelier de logiciel. Cette notion se traduit par :

- La décomposition d'un système en modules, dont chacun est défini par une spécification fonctionnelle qui décrit entièrement son comportement sans faire intervenir les détails de sa réalisation. Dans l'exemple (fig.3.2) le système SystèmeGestionBase est décomposé en modules : ManipStructure, Manipattributs, ModulesService, et StructDonnées, qui a leur tour sont décomposés.

- la possibilité de décrire un système de façon hiérarchique en séparant à divers niveaux de détails sa structure globale (assemblage de parties) de sa structure détaillée (description de chaque partie).

3.1.2 Entités et relations

La fig.3.1 résume les entités et relations qui résultent de ce mode de structuration, en utilisant les notions du modèle entité-association (CHE 76).

La terminologie dans ce domaine n'est pas standard. Dans ADA (ADA 79), par exemple, un module se compose de deux parties : son interface qui spécifie les ressources qui peuvent être utilisées de l'extérieur, et son corps qui implante concrètement ces ressources. Dans MESA (MIC 79) l'interface aussi bien que le corps sont des modules. Dans ADELE, nous definissons un module comme étant une association entre l'une des interfaces qui décrit entièrement les ressources qu'il fournit à ses utilisateurs, et d'un corps (appelé réalisation) qui réalise concrètement ces ressources. Une telle distinction entre interface et corps est maintenant courante : citons entre autres les langages MESA et ADA et les systèmes de développement GANDALF (KAI 82b), et LEGOS (BOU 80).

Interfaces et corps sont textuellement séparés (comme le montre la (fig.3.3)) et peuvent être compilés séparément. Ils utilisent au besoin des ressources fournies par d'autres modules. Dans la pratique ces ressources sont des constantes, des types, des variables, et des procédures comme le montre la (fig.3.3).

Une conséquence importante de la structuration modulaire est qu'un utilisateur (ce terme désigne aussi bien un usager humain qu'un autre module) ne distingue pas un module unique d'un système complexe constitué d'un ensemble de modules, si les deux lui présentent la même interface.

La spécification externe d'un module décrit son comportement pour un utilisateur, en faisant uniquement intervenir les ressources visibles fournies par l'interface (le type tynom, les procédures lire et écrire, ... de la (fig.3.3)). La spécification de la réalisation du corps d'un module décrit sa structure interne ; elle est inaccessible aux utilisateurs du module.

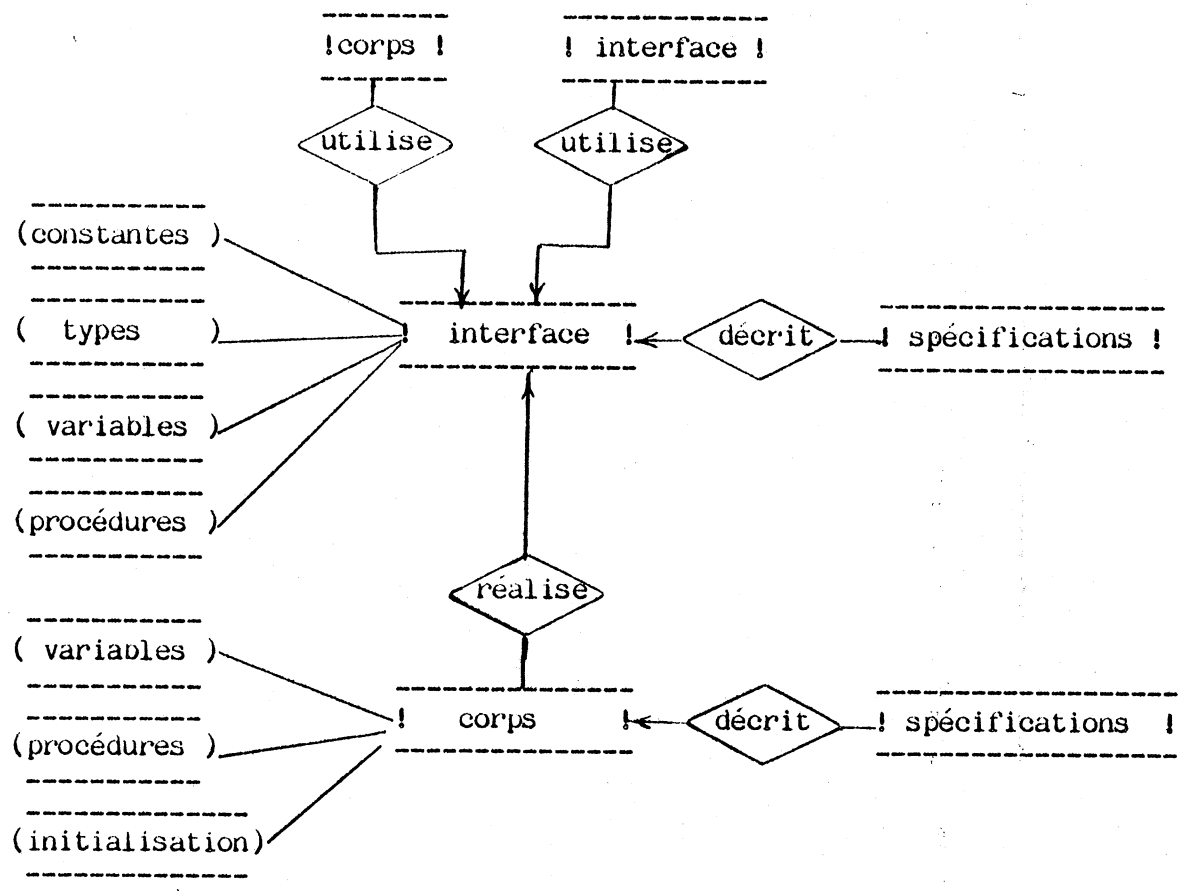


Fig.3.1 Entités et relations induites par une décomposition modulaire.

La fig.3.1 met en évidence trois types de relations entre ces divers types d'entités :

- la relation décrit entre une spécification et l'entité logicielle à laquelle elle s'applique,

- la relation réalise entre l'interface d'un module et le corps qui réalise les ressources qu'elle fournit (cette relation existe entre l'interface SystèmeGestionBase_prog et le corps SystèmeGestionBase_prog_c1 de (fig.3.3)),

- la relation utilise ou relation de dépendance, entre un module A et un module B dont la réalisation met en oeuvre des ressources fournies par A (des exemples sont donnés au (3.1.4).

L'intérêt de ces relations est qu'elles permettent à des contraintes de cohérence d'être maintenues quand une entité est modifiée, remplacée, ou supprimée. Un aspect important de ces relations est donc leur possibilité de vérification par le système de développement. Ce problème n'est actuellement pas résolu pour la relation décrit ; on peut seulement demander au système de détecter si un terme de la relation a été modifié.

La relation réalise est en général exprimée implicitement dans les langages existants, par une simple identité textuelle : la liste des ressources fournies par l'interface est reproduite dans le corps, avec leurs noms et leurs types. Cette relation d'identité peut être assouplie de trois manières :

- modification du nom : une ressource réalisée par le corps, sous un nom donné, est fournie dans l'interface par un autre nom. Ce changement permet d'éviter des conflits d'homonymie en cas de remplacement d'un composant,

- transformation de type : une ressource réalisée par le corps est fournie par l'interface avec un type différent. Cette transformation pose le délicat problème d'équivalence de type ; diverses solutions ont été proposées et qui se prêtent à une vérification automatique (WOR 81). Nous trouvons dans (EST 83b) comment cette identité stricte peut être assouplie dans ADELE,

- restriction d'accès aux ressources : parmi les ressources réalisées par un corps, on n'en rend accessibles qu'un sous-ensemble. On peut ainsi associer à un corps donné une famille d'interfaces qui jouent le rôle de filtres, chacune étant destinée à une classe particulière d'usagers différents par leurs droits d'accès.

L'étude de la relation de dépendance fait l'objet du paragraphe suivant.

3.1.3 Relation de dépendance

On dit qu'un système A dépend d'un système B si le bon fonctionnement de A (compilation, exécution, ...) nécessite la présence de B. Cette relation est celle nommée "use" dans (PAR 76). Dans un système modulaire, l'existence de la relation de dépendance se traduit sous les formes suivantes :

1) si on impose à la relation de dépendance d'être sans circuit, ce que nous supposons, la conception d'un système peut se faire d'une manière hiérarchique ; on peut alors répartir les modules en "niveaux", tels que pour tout i 0, les modules du niveau i dépendent uniquement des modules des niveaux inférieurs, comme le montre l'exemple (fig.3.4). Au niveau le plus élevé, il existe un module unique, dit principal, dont aucun autre module ne dépend. L'interface de ce module est l'interface du système complet. Réciproquement, à tout module on peut associer un système (appelé dans ADELE réalisation composée) dont ce module est le module principal, en construisant à partir de ce module la fermeture transitive de la relation de dépendance. Les termes de "module" et de "système" peuvent donc être utilisés indifféremment lorsqu'on s'intéresse aux propriétés globales, traduites dans l'interface ; on parle plutôt de "module" lorsqu'on veut désigner un composant particulier. Nous verrons nettement cette homogénéité dans ADELE,

2) la relation de dépendance entre deux modules A et B prend les deux formes suivantes :

- importation : lorsqu'un module A utilise des ressources fournies par B (donc décrites par son interface), on dit aussi que A importe ces ressources, et que B les exporte,

- inclusion : le module A inclut dans son texte le module B, ou une copie (éventuellement paramétrée) du module B.

Ces deux formes de relations diffèrent notamment par le mode d'utilisation des modules dont dépendent plusieurs autres modules : un module importé est partagé, en exemplaire unique, par ses importateurs (dans une même configuration). Un module inclus se comporte plutôt comme un modèle de macrogénération, dont sont tirées autant de copies distinctes qu'il y a de modules importateurs. Cette macrogénération peut dépendre de paramètres, mais cet aspect n'est pas considéré ici.

3) La relation de dépendance impose un ordre partiel de compilation. Un module A qui importe un module B ne peut être compilé qu'après que l'interface de B ait elle même été compilée (dans notre exemple (fig.3.3) l'interface SystèmeGestionBase_prog ne peut être compilée qu'après l'interface StructDonnées_i1).

4) La relation de dépendance permet de déterminer certaines conséquences d'une modification portant sur un composant d'un logiciel : la modification d'une interface nécessite la recompilation de tous les composants qui peuvent être potentiellement affectés par cette modification. Ces composants n'incluent pas seulement ceux qui utilisent directement cette interface, mais aussi tous ceux qui le font par transitivité. La modification d'un corps peut rester invisible tant qu'elle respecte les relations "réalise" où ce corps est impliqué.

3.1.4 Exemple

Cet exemple est une partie significative et simplifiée du système de gestion de la base de données ADELE. La fig.3.2 donne une structure partielle de décompostion modulaire, où ne sont explicitées que des relations de dépendance entre modules qui forment une arborescence. Cette structure arborescente permet essentiellement la désignation unique des composants d'un système. La structurre complète où toutes les relations de dépendance sont explicitées forme un graphe quelconque.

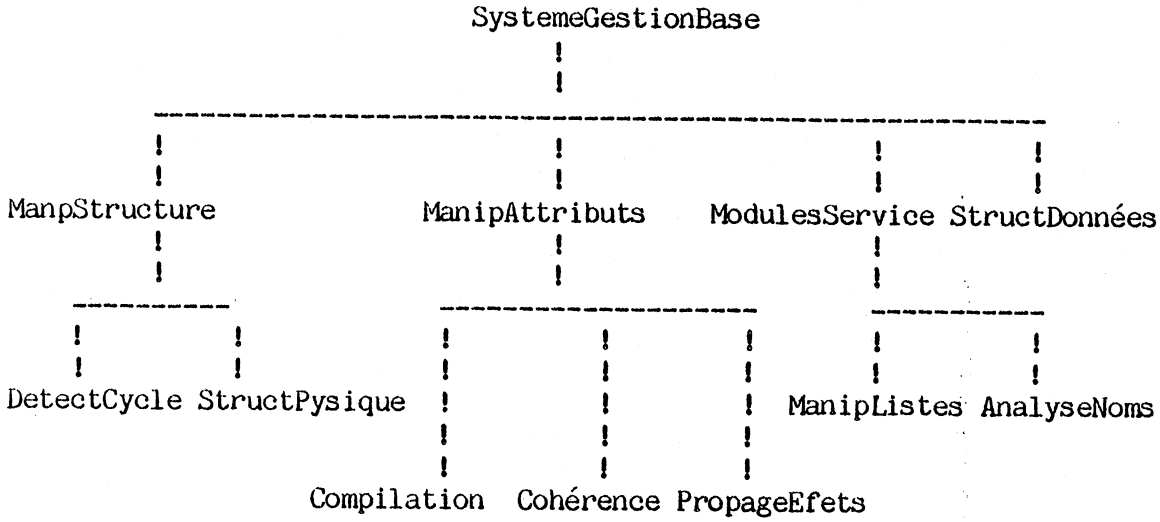


Fig.3.2 Dépendances partielles entre les modules de l'exemple.

La spécification partielle d'un de ces modules, SytèmeGestionBase, en langage Pascal ADELE (extension du pascal standard (EST 83b), est décrite par la (fig. 3.3).

SystemGestionBase_prog et SystemGestionBase_prog_c1 sont respectivement l'interface et le corps de ce module. L'interface inclut (cf. 3.1.3) les ressources fournies dans l'interface StructDonnées_i1, et le corps importe (cf. 3.1.3) les ressources exportées dans les interfaces i1 par les modules ManipStructure et ManipAttributs respectivement.

```
interface SysttèmeGestionBase_prog

    include

        StructDonnées_i1 (* contient la déclaration du type 'tynom' *)

    procedure lire (nombase, fichloc : tynom)
    procedure ecrire (nombase, fdep : tynom; util, cohérent :boolean)

    ...

end.


body SystèmeGestionBase_prog_c1

    import

        ManipStructure_i1, ManipAttributs_i1

    const
        ...

    type
        ...

    var
        ...

    procedure lire (hombase, fichloc : tynom);

        (* déclarations locales *)

    begin ... end;

    procedure ecrire (nombase,fdep : tynom; util,coherent : boolean);

        (* déclarations locales *)

    bégin ... end;


    BEGIN

        (* initialisation *)

    END.
```

Fig.3.3 Spécification partielle d'un module en Pascal ADELE.

Le graphe de la relation de dépendances (décrites au (A1.1.1)) des modules de notre exemple donne la répartition de ces modules en plusieurs niveaux hiérarchiques (fig.3.4). Pour simplifier la figure nous ne représentons pas la relation de dépendance entre chaque module et le module "StructDonnées".

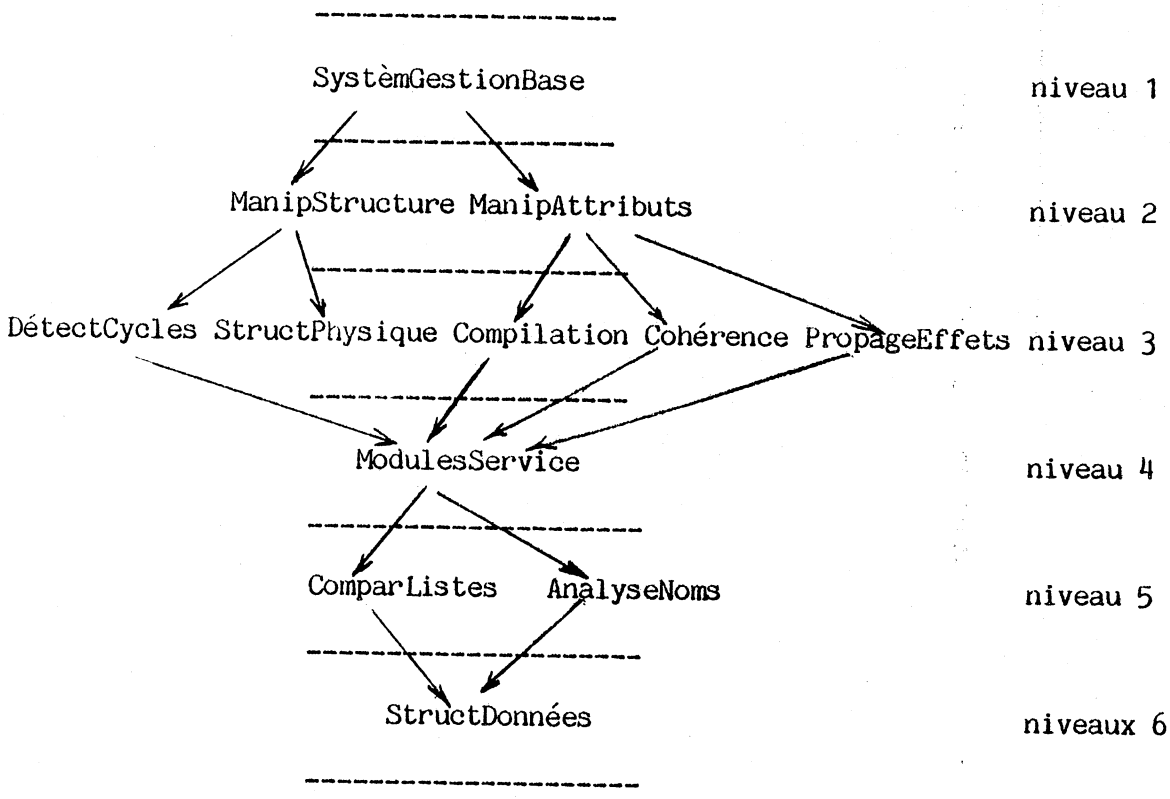


Fig.3.4 Différents niveaux du système induits par la relation de dépendance.

3.1.5 Problèmes inhérents à la décomposition modulaire

L'utilisation de la décomposition modulaire nécessite la résolution de quelques problèmes :

- la vérification inter-modules des types : la vérification des types des ressources (types, variables, procédures) partagées entre plusieurs modules s'exprime ainsi :

  - chaque module doit fournir réellement les ressources de son interface,

  - le type des ressources fournies doit être conforme à sa spécification,

  - aucun module ne peut utiliser d'autres ressources que celles spécifiées,

  - toutes les ressources importées doivent être utilisées conformément à leur type.

Ce problème de vérification est l'un des problèmes fondamentaux à résoudre dans les langages et systèmes modulaires. Cette vérification est aussi importante que celle des types au sein d'un module car plusieurs utilisateurs importent des ressources dont ils peuvent complètement ignorer l'implantation. Cette vérification doit aussi déterminer tous les modules atteints par une modification de types et décider de leur recompilation ou de leur adaptation (modification). Nous trouvons une étude de ce problème dans (TIC 79),

- l'effet de la modification d'une interface dans les grands systèmes : le coût de recompilation de l'ensemble des composants logiciels "potentiellement" affectés par cette modification est considérable, même catastrophique. En pratique cet ensemble contient un faible pourcentage des composants "effectivement" atteints par cette modification, et un gain considérable peut être fait en recompilant seulement ces composants (CAS 81). L'ensemble des composants effectivement atteints par la modification augmente avec la propagation des types via les interfaces,

- la cohérence des configurations : une configuration doit toujours être composée de versions compatibles de ses composants. Il faut pouvoir tenir compte de l'évolution des composants d'une configuration de façon à maintenir son intégrité et sa cohérence. Plusieurs travaux récents sont orientés dans ce sens (CRI 80, KAI 82b, SCH 82).

3.2 Entités et relations dans ADELE

La spécification des entités et relations gérées (pour une application) par un système de gestion de base de données est désignée dans la littérature par le terme de "schéma conceptuel".

Le schéma conceptuel est la partie fondamentale de l'architecture d'une base de données. Il a pour but de décrire en termes abstraits mais fidèles, une organisation et ses processus de gestion nécessitant la mise en oeuvre d'une base de données.

Le passage du monde réel au schéma conceptuel correspond à un processus de modélisation, où les objets du monde réel sont classés en catégories désignées par des noms et reliées par des relations.

Un SGBD fournit un langage de définition de données qui permet de spécifier le schéma conceptuel. La notion de définition de données est liée à celle de modèle de données. Un modèle de donnée (cf. 2.1.2.2) est un outil formel, utilisant un ensemble de concepts mathématiques, pour définir la sémantique des objets (d'une application donnée) conservés dans une base de données. La sémantique de ces objets est déterminée par leurs propriétés structurelles et celles de leur évolution (cf. 2.1.2.2), et en particulier par l'effet des opérations qui leur sont applicables.

Le schéma conceptuel peut donc être vu comme :

- un ensemble de structures (types) de données, de leurs opérateurs, et les mécanismes de contraintes qui maintiennent l'intégrité sémantique inter-structures. Une structure décrit un objet ou une relation.

- un ensemble de variables (entités et relations) appartenant à l'un des types définis ou construits à partir d'eux.

Il existe plusieurs modèles (cf. 2.1.2.2) qui se distinguent par la nature des associations qu'ils permettent de modéliser. Ces modèles peuvent être utilises pour servir de base à un schéma conceptuel. Parmi les modèles relationnels on peut citer : le modèle relationnel (COD 70), le modèle entité-association (CHE 76), le modèle relationnel binaire (ABR 74), et d'autres propositions de modèles plus généraux (par exemple (JOL 82, OLI 83) faits dans le cadre du projet TIGRE (TIG 83)).

Nous spécifions notre schéma conceptuel au paragraphe suivant. Certaines notions sont empruntées aux modèles précédents, essentiellement au formalisme UBIK (JOL 82).

3.2.1 Définition du formalisme

3.2.1 Type

Un type dénote en général :

- un ensemble d'objets et des opérations associées, appelés definition du type,

- une représentation des objets et des algorithmes réalisant les opérations, appelée spécification du type.

Un type abstrait est un ensemble d'objets sur lesquels il est possible d'effectuer une liste déterminée d'opérations, qui seules peuvent accéder à la représentation des objets.

Un type générique est un type paramétré par un ou plusieurs types formels. Un type générique définit donc une famille de types dont chaque membre est caractérisé par les valeurs des paramètres.

Nous utilisons le formalisme de type abstrait générique (étendu par quelques notions nécessaires à l'enrichissement sémantique) pour la description du modèle des logiciels dans ADELE.

3.2.1.2 Propriété

Nous utilisons le terme de propriété pour désigner un ensemble de fonctions dépendant d'un type formel (JAC 78). On peut donc dire que toute propriété, moyennant la donnée d'une représentation et des corps de fonctions, peut être considérée comme un type abstrait générique.

Un type particulier possède une propriété p (ce qui est exprimé dans notre formalisme par le mot clé "satisfy") si sa définition contient les fonctions implantant celles de la propriété p. Une propriété peut en impliquer une autre par le mot clé "implies".

Exemple :

```
property maniptext (?)

    -- "?" est le type formel de la propriété maniptext, il pourra
    être remplacé par tout type pouvant avoir cette propriété. Nous
    trouvons, dans les paragraphes suivants, plusieurs exemples.

    -- Les fonctions de la propriété maniptext dépendent du type
    formel "?". Le type de la partie gauche du symbole "-->" est celui
    du domaine de définition de la fonction, alors que celui de la
    partie droite est le type du domaine de valeurs de la fonction.


    init:       -->? --initialisation d'un objet de type "?" (après
    sa création),

    éditer:  ? -->? --éditer (ou modifier) un objet de type "?"
    (modifier sera utilisé comme synonyme de cette fonction),

    imprimer: ? -->? -impression d'un objet de type "?".

end maniptext
```

Quelques proprietés sont engendrées par des types prédéfinis :

- le type relationship, qui implante les relations bien connues du modèle relationnel, définit la propriété maniprelation suivante:

```
property maniprelation

    insert:     -->? --insertion d'un n-uplet dans une relation
    de type "?",

    remove:   ? -->   --suppression d'un n-uplet
    update:   ? -->? --mise à jour d'un n-uplet

end maniprelation
```

- le type text, qui implante des textes, définit la propriété maniptext (décrite en haut) :

```
type text (structure ... end structure)

    init:       -->?
    éditer:   ? -->?
    imprimer: ? -->?

end text
```

3.2.1.3 Type entity

Le type entity est destiné à la représentation des objets du monde réel, sur lesquels le système ne possède aucun moyen d'action direct (des personnes, des projets, ...) ; c'est le type qui correspond à une abstraction dans les modèles de données classiques.

Le type entity est un type générique ayant un seul paramètre de généricité : la relation qui forme des entités du type modélisé, à partir de leurs attributs. Ce type est prédéfini comme suit :

```
type <id_type_entité> is new entity

    <schéma_de_la_relation>

end.
```

Des exemples sont donnés au (3.2.2).

3.2.1.4 Type relationship

Ce type permet de représenter les liens sémantiques entre différentes entités. C'est un type générique qui possède trois paramètres de généricité correspondant respectivement à la relation décrivant l'association et au deux types d'entités associées. Il symbolise une relation binaire de cardinalité (m, n) entre les entités de type t1 et les entités de type t2. Le lien est matérialisé par les fonctions rol1 et rol2. Ce type est prédéfini comme suit :

```
type <id_type_relation> is new relationship

    between type1 (rol1, m) and type2 (rol2, n)
    <schéma_de_la_relation>

    --le caractère "*" peut être utilisé pour indiquer une cardinalité
      quelconque.

end.
```

Nous trouvons des exemples au (3.2.2).

3.2.1.5 Type document

Le type document est un exemple d'objet sur lequel le système peut agir directement. Ce type est paramétré par une caractéristique (cette caractéristique peut être la structure du document). Il est prédéfini comme suit :

```
type <id_type_document> is new document

    text      is    id_type_texte
    <schéma_de_la_relation>

end.
```

Des exemples sont donnés au (3.2.2.4).

3.2.1.6 Contraintes

Nous introduisons informellement, pour notre besoin d'expression, la notion de contraintes ("constraints") qui est un attribut de type tableau d'actions (procédures). Une action Ai est déclenchée si une opération permise Oi est effectuée sur un objet ayant cet attribut. C'est donc un moyen pour maintenir la cohérence entre les objets de la base (une autre alternative serait d'intégrer ces actions dans le code des opérations qui les déclenchent, ce qui figerait le modèle).

L'attribut "constraints" sera utilisé pour spécifier les contraintes structurelles des objets de différents types (entité, relation, et document). Ces contraintes sont donc statiques dans un modèle. La définition des contraintes statiques et dynamiques, ainsi que leur nature, est donnée au (2.1.2.2). Les actions seront décrites dans un langage informel, pour faciliter la compréhension. Des exemples sont donnés au cours de ce chapitre.

3.2.2 Un schema conceptuel

3.2.2.1 Généralités

Dans ADELE les types des entités et les assciations qui existent entre elles sont prédéfinis. Le SGBD ADELE n'est donc pas général, mais specialisé dans la gestion de ce type d'entités et associations. Sa puissance est donc restreinte (pas de langage de définition des types du schema conceptuel, puisqu'ils sont prédéfinis). Le SGBD ADELE offre seulement le moyen de déclarer des variables de ces types.

Dans ce qui suit, nous décrivons (en utilisant les notions précédemment introduites) la partie principale du schéma conceptuel de la base ADELE : les types prédéfinis du schéma. L'autre partie, création des entités et relations de ces types, sera exposée au (3.2.4).

Les objets gérés par ADELE sont de trois types : entity, relationship, et document.

Du type entity nous avons généré trois types d'entités en fixant le paramètre de généricité. Ces trois types sont :famille, interface, et réalisation qui peuvent être informellement définis comme suit :

- famille : une famille définit un ensemble d'interfaces "voisines" et logiquement reliées entre elles. L'exemple le plus courant de famille est l'ensemble des systèmes dont les interfaces sont obtenues par diverses restrictions de l'ensemble des ressources offertes par une interface de base.

Le couple interface-réalisation (cf. 3.1.2) correspond à la notion habituelle de module. Une famille est alors l'ensemble des versions d'un module donné ; elle peut être vue comme un module au sens large qui peut évoluer, se diversifier, s'adapter, et avoir des codes objets et des documents associés.

Vue comme un module, une famille f peut utiliser deux sortes de familles :

- celles qui sont issues de sa décomposition, appelées familles filles de f,

- celles qui sont issues de la décomposition d'autres familles, appelées familles externes à f.

- interface : la définition précise d'une interface est donnée au (3.1.2). Par rapport aux publications antérieures (voir par exemple (EST 83a, CHV 82), nous avons apporté une précision : une interface peut se reduire à des déclarations de constantes, de types, et de variables. Ce type d'interface (désigné par le terme d'"environ") peut être sans réalisation. Un environ a toutes les propriétés d'une interface sauf qu'on ne peut l'utiliser que par "inclusion" dans une interface ou un autre environ,

- réalisation : la définition précise d'une réalisation est donnée au (3.1.2). Le terme de réalisation est utilisé aussi bien pour désigner un corps qu'une configuration. Ces deux notions sont identiques dans ADELE (cf. 3.1.2). Nous utilisons cependant le terme de "corps" et de "réalisation composée" pour faire la différence quand c'est nécessaire.

Le type relationship contient essentiellement les relations introduites par la décomposition modulaire, et certaines autres relations propres à la méthode de développement de logiciels utilisée dans ADELE.

Les documents peuvent varier selon les caractéristiques fournies en paramètre.

Nous utiliserons :

- la notation suivante : <id_attribut> (m, n) is <id_type_de_domaine_de_valeurs> pour exprimer que <id_attribut> est de cardinalité minimale et maximale m, n respectivement, et qu'il prend ses valeurs dans <id_type_domaine_de_valeurs>,

- l'attribut "man" (manuel) : qui contient un texte (fourni par l'usager) définissant les contraintes d'utilisation de l'entité ayant cet attribut. Ces contraintes sont manipulables dynamiquement par les usagers ayant ce droit,

- l'attribut "text" : qui contient les sources des entités qui ont cet attribut. Si cet attribut est multivalué (cas des réalisations), alors chaque valeur (source) constitue une révision d'une autre (cf. 4.4.1),

- l'attribut "doc" : qui contient le document "décrivant" (au sens de la relation décrit de la décomposition modulaire) l'entité à laquelle il est associé,

- et la propriété :

```
property manip (?)

    -- propriété de création et de destruction des objets pouvant
    avoir cette propriété.

        créer:      -->? --création d'un objet
        détruire: ? -->? --destruction d'un objet
        annuler:  ? -->? --annulation des effets d'une opération

end manip.
```

3.2.2.2 Définition des entités

1) Définition de l'entité famille

```
type famille is new entity

     integer n

     --attributs

          --contraintes d'intégrité dynamiques
     man (1,1)                  is   contraintes_famille
     programmeurs               is   espace_travail
          --fin de contraintes dynamiques

     doc (1,1)                  is   doc_famille

     --relations
     familles_filles (0,n)      is   rel_décomposition.familles_filles
     fille_de (0,1)             is   rel_décomposition.fille_de
     familles_externes (0,n)    is   rel_liens.familles_externes
     externe_a (0,n)            is   rel_liens.externe_a
     interfaces (0,n)           is   rel_interfaces.interfaces

     satisfy manip

     --contraintes d'intégrité statiques
     constraints

          f : famille
     créer :
          début

            init (f.man), init (f.doc),
            init (f.programmeurs),

            si f n'est pas la première famille alors
               f doit être familles_filles d'une autre
            fin
            mise à jour de rel_décomposition
            mise à jour de rel_liens

          fin créer

     détruire :
          début

            si f.familles_filles ≠ ∅ alors annuler (détruire) fin
            détruire (f.man), détruire (f.programmeurs),
            détruire (f.doc),
            pour tout interface i∈ f.interfaces faire détruire(i),
            mise à jour de rel_décomposition
            mise à jour de rel_liens

          fin détruire

     fin constraints

end famille.
```

L'attribut "programmeurs" associe à une famille un document qui décrit ses contraintes d'utilisation : l'ensemble des personnes autorisées à travailler sur cette famille (cf. 4.2.4). Les contraintes d'intégrité dynamiques seront détaillées dans le chapitre 4, et les relations seront définies au (3.2.2.4).

2) Définition de l'entité interface

```
type interface is new entity

    integer n

    --attributs

    -- contraintes d'intégrité dynamiques

    man (1,1)                is   contraintes_interface

    -- fin de contraintes d'intégrité dynamiques

    doc (1,1)                is   doc_interface
    text (1,1)               is   text_interface

    --relations

    interface_de (1,1)       is   rel_interfaces.interface_de
    réalisée_par (0,n)       is   rel_réalise.réalisée_par

    satisfy manip

    --contraintes d'intégrité statiques

    constraints

        i : interface

    créer :

        début

          init (i.man), init (i.text), init (i.doc),
          si i est un environ alors i.réalisée_par doit être vide,
          mise à jour de rel_interfaces

        fin créer

    détruire :

        début

          détruire (i.man), détruire (i.text), détruire (i.doc),
          pour toute réalisation r∈ réalisée_par faire
                  détruire (r),
                  mise à jour de rel_réalise,
          fin

          mise à jour de rel_interfaces

        fin détruire

    fin constraints

end interface.
```

3) Définition de l'entité réalisation

```
type réalisation is new entity

    integer n

    --attributs

    --contraintes d'intégrité dynamiques

    man (1,1)              is   contraintes_réalisation

    --fin de contraintes d'intégrité dynamiques

    text (1,n)             is   <id_type_text_réalisation>
    doc (1,1)              is   doc_réalisation

    --relations

    réalise (1,1)          is   rel_réalise.réalise

    satisfy manip

    --contraintes d'intégrité statiques

    constraints

        r : réalisation

    créer :

        début

          init (r.man), init (r.text), init (r.doc),
          mise à jour de rel_réalise

        fin créer

    détruire :

        début

          détruire (r.man), détruire (r.text), détruire (r.doc)
          mise à jour de rel_réalise,

        fin détruire

    fin constraints

end réalisation
```

Chaque réalisation d'une interface est une version d'une réalisation particulière. L'attribut "text" associe à une version un ou plusieurs documents qui sont ses révisions (cf. 4.4.1). Le type de l'attribut "text" est un paramètre de l'entité réalisation ; on peut donc avoir selon ce type plusieurs types de réalisations. Dans ADELE on a deux types de ce "text" : text_corps pour un corps et text_réalcomp pour une réalisation composée.

5.2.2.3 Définition des documents

Dans ADELE les documents peuvent être répartis en trois catégories (ensemble de types ayant une même sémantique et pouvant avoir des représentations différentes) : la catégorie des contraintes d'intégrité (types de l'attribut "mus"), la catégorie des textes sources des entités (types de l'attribut "text"), et la catégorie des documents au sens classique (contenant des spécifications fonctionnelles des entités auxquelles ils sont attachés ; ils sont du type de l'attribut "doc"). Dans ce qui suit on ne spécifie que les documents du type "doc" et "text" ; ceux des contraintes d'intégrité seront spécifiés au chapitre 4.

Nous introduisons les propriétés suivantes :

property ci (?)

--propriété des contraintes d'intégrité

-- "?" est le type formel de la propriété

cohérence:? --> booléen --vérifie si un texte de contraintes d'intégrité (nouvellement introduit ou modification d'un autre) est cohérent avec l'ensemble des autes textes de contraintes impliqués dans les mêmes relations que lui. Effectue toutes les opérations nécessaires pour remettre la base dans un état cohérent avec ces nouvelles contraintes. Les grandes lignes de cette opération sont données en (A2).

implies maniptext

end ci

property réal (?)

--propriété des réalisations

interpréter: ? --> * --interprète le texte d'une réalisation. "*" est utilisé pour indiquer un type quelconque,

compiler: ? --> text_binaire --génération du code binaire du texte d'un objet ayant cette propriété,

compléter: ? --> ? --un texte d'une réalisation peut être incomplet si, pour des raisons de contraintes, il ne contient pas tous les composants qui devaient le constituer (cf. 5.3.3.4). Cette opération permet quand une réalisation composée est complète de propager les effets de sa complétude,

incomplet: ? --> ? --opération inverse de compléter,

listcomp: ? --> list_comp --construit la liste de composition d'une réalisation composée en manipulant les attributs et les relations nécessaires. Le type list_comp est défini plus loin dans text_réalcomp,

décompiler: ? --> * --transforme le texte en une présentation externe conformément à une syntaxe donnée.

implies maniptext

end réal

```
property int (?)

    --propriété des interfaces

    include: ? --> ?x? --inclusion d'environs
    import:  ? --> ? --importation du texte d'une interface
    décompiler? --> *
    restrict: ? --> ? --restriction d'une interface par une autre (cf.
    4.3.1).

    implies maniptext

end int

property text_bin (?)

    --propriété des codes objets

    exécuter: ? --> * --execution d'un texte binaire
    périmer:  ? --> ? --rendre invalide un code objet et propager les
    effets de cette invalidation,

    implies maniptext

end text_bin
```

1) CATEGORIE DOCUMENTS

Généralités

La manipulation des documents n'est pas encore mise en oeuvre dans ADELE, nous faisons cependant, apparaître quelques aspects dans ce domaine en nous intéressant particulièrement aux documents d'interface et de réalisation. Parmi nos objectifs principaux, nous pouvons citer :

- l'assistance à la construction et à la manipulation des textes des documents. Les parties essentielles d'un document (qui décrivent l'état de l'entité et ses fonctions) peuvent (ne serait ce que partiellement) être construites par l'extraction automatique d'informations, au moment de la manipulation du texte par l'éditeur. Par exemple la déclaration des types, des constantes, des procédures avec leurs commentaires, ... D'autres informations (graphe de dépendance, ...) peuvent être calculées par le système de gestion de la base de données. Ces parties seront donc modifiables uniquement par le système. Les parties qui ne peuvent être calculées automatiquement sont laissées aux soins du concepteur. Le système l'assiste par des moyens de structuration, d'accès aux structures, et de manipulation (éditeur de structure, déplacement d'une structure à une autre, modification d'un texte, ...),

- le maintien de la cohérence entre documents et textes (sources) des entités : cet objectif ne peut être entièrement automatisé, mais les parties des documents qui sont sous la responsabilité du système peuvent l'être. Le problème est donc d'automatiser le plus possible. Il est cependant souhaitable, pour simplifier le maintien de la cohérence entre un document et les textes d'entité, que les informations extraites des textes soient partagées et non dupliquées.

Nous spécifions dans les paragraphes suivants une structure possible pour les deux documents d'interface et de réalisation en utilisant les notations introduites dans (BOG 83).

document de l'interface

Le document de l'interface est défini comme suit :

```
type doc_interface is new document

    --attributs

    text (1,1)          is   text_doc_interface
    identité (1,1)      is   text_système
    historique(1,1)     is   text_historique
    état                is   (cohérent, incohérent)

    satisfy manip

end doc_interface
```

Le type text_doc_interface peut être défini par :

```
type text_doc_interface is new text

    structure

        text libre : éditeur_text

        constantes

            (déclaration_constantes,
            commentaires_constantes)  : éditeur_syntaxique

        type

            (déclaration_types,
            commentaires_types)  : éditeur syntaxique

        variables

            (déclaration_variables,
            commentaires_variables)  : éditeur_syntaxique

        Procédures

            (déclaration_procédures,
            commentaires_procédures)  : éditeur_syntaxique

        réalisations

            (nom_réalisation,
            commentaires_réalisation)  : éditeur_text

    end structure

    satisfy manip, maniptext

end text_doc_interface
```

Le type text_système est défini par :

```
type text_système is new text

     structure

          nom_entité           : char (114),
          nom_auteur           : char (32),
          date_création        : char (6),
          date_dernière_modif  : char (6),

     end structure

     satisfy manip, maniptext

end text_système
```

et text_historique est défini comme suit :

```
type text_historique is new text

     structure

          attribut_modifié     : (man, text, doc),
          auteur_modif         : char (32),
          date_modif           : char (6),
          commentaire_modif    : éditeur_text,

     end structure

     satisfy manip, maniptext

end text_historique.
```

Tous les attributs sont gérés par le système sauf "commentaire_modif" où le système invite (par un passage immédiat, par exemple, sous un éditeur de documents) l'auteur de la modification à la commenter.

Document d'une réalisation

Le document d'une réalisation peut être spécifié comme suit :

```
type doc_réalisation is new document

    --attributs

    text (1,1)          is text_doc_réalisation
    identité (1,1)      is text_système
    historique (1,1)    is text_historique
    état                is (cohérent, incohérent)

    satisfy manip

end doc_réalisation
```

Le text_doc_réalisation peut être défini comme suit :

```
type text_doc_réalisation is new text

    structure

        text libre              : éditeur_text

        interfaces importées    : text_doc_interface

        interfaces exportées    : text_doc_interface

        suivi                   : éditeur_text

    end structure

    satisfy manip, maniptext

end text_doc_réalisation
```

Le text_doc_interface, importé ou exporté, peut être partagé avec les interfaces concernées.

2) CATEGORIE DES TEXTES SOURCES

Seules les interfaces et réalisations possèdent un attribut "text". Dans ce paragraphe seront spécifiés les domaines de valeurs de cet attribut.

text_interface

```
type text_interface is new document

    integer n

    -- attributs

    text (1,1)          is  éditeur_syntaxique
    état                is (cohérent, incohérent)

    --relations

    inclut (0,n)        is rel_inclusion.inclut
    inclus_par (0,n)    is rel_inclusion.inclus_par
    importé_par (0,n)   is rel_import.importé_par

    satisfy int (include : text_interface, import : text_corps)

    --contraintes statiques

    constraints

        t : text_interface

    modifier :

        début

          pour toute réalisation (son text) tr∈ t.importé_par faire
        début
          début
             tr.état <-- incohérent,
             pour tout code objet c∈ tr.co faire périmer (c),
          fin


          pour toute réalisation (son text) tr réalisant l'interface
             de t faire
          début
             tr.état <-- incohérent,
             pour tout code objet c∈ tr.co faire périmer (c),
          fin


        fin modifier

    fin constraints

end text_interface
```

"include : text_interface" exprime le fait que l'opération include sur une interface ne peut être faite que par un text_interface, et "import : text_corps" exprime que l'"import", d'un text_interface, ne peut être fait que par un text_corps.

```
text corps

    type text_corps is new document

        integer n

        --attributs

        text (1,1)              is éditeur_syntaxique
        co (0,n)                is générateur_code
        état                    is (cohérent, incohérent)

        --relations

        list_dependance(0,n)is rel_import.import
        composant_de (0,n)  is rel_composition.englobé_par

        satisfy réal (listcomp : text_réalcomp)

        --contraintes

        constraints

            t : text_corps

        modifier :

            début

                si t.état = cohérent alors compléter (t)
                sinon incomplet (t)
                fin

                pour tout code objet c∈ t.co faire périmer (c)

            fin modifier

        fin constraints

    end text_corps
```

```
text réalcomp

    type text_réalcomp is new document

        integer n

        --attributs

        text (1,1)          is text_contr_réalisation
        co (0,n)            is générateur_code
        état                is (cohérent, incohérent, complet,incomplet)

        --relations

        list_dépendance(0,n)is rel_import.import
        list_comp (0,n)     is rel_composition.englobe
        composant_de (0,n)  is rel_composition.englobé_par

        satisfy ci, réal (listcomp : text_réalcomp)

        --contraintes statiques

        constraints

            t : text_réalcomp

        modifier :

            début

              si non cohérence (t) alors
              début
                 incomplet (t),
                 t.état <-- incohérent
              fin

              sinon
              début
                 si t.état = complet alors compléter (t),
                 mise a jour de rel_composition,
                 cohérence (t.man.text)
              fin

              pour tout code objet c∈ t.co faire périmer (c)

            fin modifier

        fin constraints

    end text_réalcomp
```

Le type "text_contr_réalisation" sera spécifié au chapitre 4.

5.2.2.4 Définition des relations

Dans ce qui suit nous définissons les relations de la même manière que les entités précédemment décrites, sur lesquelles elles portent :

rel décomposition : cette relation permet de décrire la hiérarchie des familles qui correspond à celle de la décomposition hiérarchique des systèmes.

```
type rel_décomposition is new relationship
     between famille (familles_filles, *) and famille (fille_de, 1)

     constraints
          f : famille
     insert :
          début
            si f est fille_de une autre famille F alors
               f ne peut être famille externe_à F
            fin
            si f est une familles_filles d'une autre famille f alors
               f.fille_de = F
            fin
          fin constraints

end rel_composition
```

- rel liens : associe à une famille donnée la liste des familles qui ne sont pas issues de sa décomposition, mais de celle d'autres familles et qu'elle utilise.

```
type rel_liens is new relationship
     between famille (familles_externes, *) and famille (externe_à, *)
     constraints
          f : famille
     insert :
          début
            si f est externe_à une autre famille F alors
               f ne peut être fille_de F
            fin
          fin constraints

end rel_liens.
```

- rel interfaces (interfaces, interface_de) : associe à une famille l'ensemble de ses interfaces, et inversement à une interface sa famille.

- rel réalise (réalisée_par, réalise) : associe à une interface l'ensemble de ses réalisations, et inversement à une réalisation l'interface qu'elle réalise.

- rel inclusion (inclut, inclus_par) : associe à une interface les environs qu'elle inclut ou qui l'incluent.

- rel import (import, importé_par) : associe à une réalisation la liste des interfaces qu'elle importe, et inversement à une interface la liste des réalisations qui l'importent.

- rel composition (englobe, englobé_par) : associe à une réalisation composée la liste des réalisations qui la constituent, et inversement à une réalisation la liste des réalisations composées dont elle est constituante.

3.2.3 Désignation

3.2.3.1 Généralités

La désignation d'un attribut "a" d'une entité "e" peut être notée de plusieurs manières : "a(e)", "e.a", ... Dans ADELE nous avons retenu la notation "e.a".

Si "a" est multivalué, une valeur particulière de "a" sera désignée par "e.a.<valeur_particulière>".

Exemples :

- l'attribut "doc" d'une famille f1 est désigné par : f1.doc,

- la famille f2, valeur particulière de l'attribut familles_filles de f1, est désignée par : f1.familles_filles.f2,

- la famille f3, valeur particulière de l'attribut familles_filles de f2, est désignée par : f1.familles_filles.f2.familles_filles.f3,

- l'interface it1, valeur particulière de l'attribut "interfaces" de f1, est désignée par : f1.interfaces.it1.

Cette désignation étant très lourde, nous avons apporté une simplification considérable en utilisant les notations : ">" pour ".familles_filles" et ".familles_externes", "<" pour ".fille_de", et "_" pour ".interfaces" et ".réalisée_par". Les exemples précédents s'écrivent respectivement comme suit :

f1.doc, f1>f2, f1>f2>f3, f1_it1

Si on veut désigner "it1" à partir de "f1" on écrira : _it1 ; et sans ambiguïté on sait qu'il s'agit d'une interface associée à f1. La désignation de f1 à partir de f2 se fait par <f1.

3.2.3.2 Désignation formelle

- désignation d'un composant : pour désigner un composant, remarquons d'abord que toute famille comporte une interface et un corps principal ; réciproquement on peut associer à tout corps C ou interface I la famille F dont C est le corps principal et I l'interface. On peut donc désigner un composant par un nom qualifié :

<nom_de_composant> ::= <nom_de_famille><nom_local_de_composant>

D'autre part, par analogie avec les systèmes de fichiers, nous convenons que tout nom est interprété dans un contexte ou environnement (appelé dans ADELE espace de travail) courant qui définit un ensemble de noms. Par convention un nom de famille définit un contexte : un composant peut y être désigné sans ambiguïté par son seul nom local, à l'aide de ces règles :

```
<nom_local>                    ::= <nom_local_d'interface>/
                                   <nom_local_de_réalisation>/
                                   <nom_local_d'attribut>

<nom_local_d'interface>        ::= "_"<identificat>

<nom_local_de_réalisation>     ::= <nom_local_d'interface>"_"<identificat>

<nom_local_d'attribut>         ::= (<nom_local_d'interface>)
                                   (<nom_local_de_réalisation>)
                                   "."<identificatatribut>

<identificatattribut>          ::= "man"/"text"/"doc"
```

Pour l'attribut multivalué "text" des corps, une révision est désignée par "text."<entier> ou par "."<entier> (exemple text.02, ou .02).

- désignation de familles : nous définissons tout d'abord un contexte universel, ou racine, permettant de construire un ensemble initial de noms, au moyen d'identificateurs. On dit aussi que ce contexte définit le premier niveau, auquel appartiennent les familles désignées par ces noms. La décomposition de familles (systèmes) en sous-familles (familles_filles) permet de définir des niveaux successifs à partir de ce niveau initial et de construire ainsi les noms qualifiés. Le principe de la désignation locale s'applique aussi pour les familles : un nom local de famille est interprété dans un contexte particulier ; en outre, et par convention, les noms universels (construits à partir de la racine) sont interprétables dans tous les contextes. Plus précisément :

```
<nom_de_famille>               ::= <nom_universel_de_famille>/
                                   <nom_local_de_famille>

<nom_local_de_famille>         ::= <identificat>(">"<identificat>)*

<nom_universel_de_famille>     ::= ">"<nom_local_de_famille>
```

La décomposition en sous-familles (familles_filles et familles_externes) ne définit pas un schéma unique de désignation lorsque des sous-familles sont partagées : elles sont familles_filles d'une famille et externes à d'autres. On pourra donc les désigner en suivant les attributs familles_filles de leurs parents ou en suivant les attributs familles_externes (qui sont des liens) des familles qui les utilisent. Pour éviter cette désignation multiple, nous convenons de désigner une famille en suivant l'attribut familles_filles de ses parents.

- désignation par défaut : pour alléger la désignation nous convenons d'associer à chaque famille, une interface qui sera prise dans l'interprétation des noms si aucun nom d'interface de cette famille n'est précisé (exemple "f_" sera interprété "f_i1", si i1 est l'interface par défaut de f1). De la même manière on associe à une interface une réalisation par défaut. Ce mécanisme est très souvent utilisé dans la désignation.

3.2.4 Rôle du formalisme dans ADELE

Dans ADELE, le rôle du formalisme dans lequel est exprimé le modèle des logiciels n'est pas seulement limité à la représentation claire et complète des objets (entité, relation, et document) gérés par la base, en intégrant la description des données aux traitements qu'elles subissent. Mais il permet aussi de donner à l'usager une interface orientée "objet-opérateur". Cette interface permet à l'usager, de raisonner uniquement au niveau des abstractions définies dans l'atelier, et de sélectionner les opérations définies sur un objet donné.

De plus le rôle de ce formalisme est de refléter, le plus clairement possible, la sémantique des données et les mécanismes mis en oeuvre pour la maintenir. Parmi ces mécanismes celui de la désignation et l'interprétation des noms des objets dans ADELE (déduit du modèle au 3.2.3).

La déclaration d'une entité d'un type donné est réalisée par une fonction de création qui déduit à partir du nom de l'entité à créer (son seul paramètre) le type de cette entité. Le système veille à ce que cette entité ne soit manipulée que par les opérations définies sur son type.

Nous verrons dans les chapitres suivants l'impact de ce formalisme sur les autres mécanismes tels que l'expression et la composition des configurations, l'évolution des objets, ...

4 INTEGRITE ET EVOLUTION D'UN SYSTEME MODULAIRE DANS ADELE

Au chapitre précédent, nous avons exprimé dans le schéma conceptuel les contraintes d'intégrité statiques. Dans ce chapitre nous définissons l'autre type de contraintes : les contraintes d'intégrité dynamiques que l'usager peut imposer et que l'atelier doit vérifier. Ces contraintes définissent essentiellement les règles qui permettent la composition de systèmes à partir des composants logiciels. Nous commencerons donc par rappeler la notion de composition de système, puis nous donnons un formalisme pour l'expression et le maintien de ces contraintes (règles) de composition, et nous terminerons par l'évolution des systèmes dans ADELE.

4.1 Expression de la composition d'un système

La composition est l'opération qui permet de construire un système complexe à partir de ses composants. Nous distinguons deux aspects : l'expression de la composition, et la construction proprement dite. C'est surtout le premier aspect qui nous intéresse ici, l'autre est détaillé au (4.3.2.2) et (A2). Suivant (DER 75), on peut qualifier l'expression de la composition, d'un système, de "programmation globale" ("programming in the large") par opposition à la programmation détaillée ("programming in the small").

L'expression de la composition a reçu relativement peu d'attention jusqu'à une période récente. Son expression classique n'est autre qu'une liste de commandes à l'éditeur de liens, spécifiant les fichiers qui contiennent les composants à relier. Ce niveau d'expression est comparable à un langage d'assemblage. Nous trouvons dans la littérature les motivations, les concepts de base, et des études des langages d'expression de la composition d'un système (voir par exemple (KRA 81)).

Si chaque module n'existait seulement qu'en une seule version, les composants d'un système seraient déterminés en parcourant le graphe obtenu en suivant les deux relations : réalise et utilise (induites par la décomposition modulaire (cf. 3.1.2)) à partir de l'interface du système. Le problème du choix de la version d'un module ne se poserait pas. La difficulté de l'expression de la composition et de sa maintenance provient essentiellement de la présence de versions multiples et de révisions de modules. Ce qui nécessite un moyen pour spécifier :

1. la version (réalisation) particulière de chaque interface,

2. la révision particulière nécessaire à chaque version qui est un composant du système,

3. les contraintes particulières entre les différents composants. Deux types de contraintes paraissent nécessaires (HUF 81) :

   - implication : la présence d'un composant A implique la présence d'un autre composant B (ces composants sont par exemple adaptés à une configuration hardware),

   - incompatibilité : la présence d'un composant A exclut celle d'un composant B (ces composants sont, par exemple, des variantes d'implantation d'un même algorithme, ou peuvent nécessiter des versions incompatibles d'un système d'exploitation).

Le problème est de trouver une expression pour ces relations, et de l'utiliser pour la construction et la modification de composition de systèmes. En particulier le système propagera les effets d'une modification dans la description de composition.

Dans les systèmes récents, on peut distinguer deux formes pour l'expression de la composition :

1) expression globale, ou langage de connexion

La composition d'un système est exprimée dans un langage qui définit des objets tels que module ou interface, et des opérations telles que la liaison. Un aspect essentiel est la vérification de la compatibilité des types des ressources importées et exportées.

Le langage MESA (MIC 79), par exemple, est complété par un langage de connexion appelé C/MESA. Le langage MESA définit des interfaces et des corps ; C/MESA définit des configurations qui assemblent des composants : interfaces, des corps, et des configurations. Un programme en C/MESA constitue une description globale d'un système. L'utilisation de configurations comme composants permet de rendre cette description hiérarchique (notion de sous-système). Le langage de connexion permet de nommer des composants, d'établir la relation "réalise" entre corps et interface, et de vérifier la compatibilité des types des ressources importées et exportées. Plusieurs interfaces peuvent être associées à un même corps ; réciproquement, plusieurs corps peuvent coopérer à la réalisation d'une interface. Une expérience d'utilisation de C/MESA est rapportée dans (NOR 79, LAU 79). Nous trouvons dans la littérature des langages plus ou moins proches (voir par exemple (DER 75, LUC 76, TIC 79, SCH 82)),

2) expression de la relation de dépendance

Une autre façon d'exprimer la composition d'un système est de la considérer comme une description d'un chemin dans un graphe qui commence à partir de l'interface la plus externe (l'interface de la configuration en entier). Ce graphe est sans circuit. Les noeuds sont des interfaces et des réalisations (au sens d'ADELE) et les arcs sont les relations réalise et utilise. A chaque noeud la sélection du noeud suivant se fait comme suit :

- choisir une version (réalisation) pour l'interface représentant le noeud,

- pour cette version choisir une révision (cette version peut dépendre d'autres interfaces qui sont implantées par d'autre réalisations, ...).

La configuration est définie en termes d'attributs spécifiques de ses composants. Cette définition est sous la forme d'expression logique de ces attributs. La configuration est alors construite par une recherche dans le graphe à partir de l'interface la plus externe. A chaque noeud l'expression qui définit la configuration est comparée aux attributs de la version et de la révision du composant, et le choix sera fait ou une impossibilité sera détectée (un exemple de la mise en oeuvre de ce mécanisme est décrit au A1.2).

Cette forme d'expression de la composition est propre à ADELE, et elle sera détaillée dans les paragraphes suivants. Néanmoins elle a été introduite d'une manière différente dans d'autres systèmes.

Par exemple, Make (FEL 79) est un outil de construction de programmes réalisé sur le système UNIX (voir par exemple (MIT 81) pour une description de ce système vu comme un environnement de programmation). Make construit un système composé en utilisant la relation de dépendance entre les composants de ce système. Make utilise deux sources d'information sur les dépendances :

- informations internes : des dépendances telles que objet-source se traduisent par des conventions de dénomination des fichiers correspondants et sont donc automatiquement détectables,

- informations externes : le progarmmeur doit fournir les autres relations de dépendance sous forme d'une liste de construction ("Makefile") qui énumère ces relations, ainsi que les commandes à exécuter pour les mettre en oeuvre lorsqu'elles ne sont pas connues du système. A chaque système est donc associée sa liste de constructions. La commande Make utilise cette liste et ses informations pour construire et explorer le graphe de dépendance. La relation de dépendance sert à déterminer les répercussions de la modification d'un composant. Cette évolution est détectée par le système en estampillant chaque module par la date de sa dernière modification. Les recompilations ou éditions des liens sont ainsi limitées à ce qui est strictement nécessaire.

4.2 Expression des contraintes

Dans ce paragraphe, nous définissons les quatre types de documents, contenant les contraintes que l'usager peut initialiser, ou modifier ; le système de gestion de la base de données maintient la base dans un état cohérent après chacune des opérations.

Les contraintes de composition, associées à une famille, sont implicitement associées à toutes les interfaces et réalisations de cette famille. Les contraintes de composition associées à une interface sont implicitement associées à toutes les réalisations de cette interface.

4.2.1 Contraintes famille

Ce document est le type de l'attribut "man" de l'entité famille, décrite au chapitre précédent. Il est défini comme suit :

```
type contraintes_famille is new document

     --attributs

     text (1,1)     is   text_contr_famille
     état           is   (cohérent, incohérent)

     satisfy manip

     --contraintes statiques

     constraints

          t : contraintes_famille

     modifier  (t.text) :
          début

            si non cohérence (t.text) alors t.état <-- incohérent
            pour toute réalisation composée tr englobant une
                réalisation de la famille f ayant cet attribut,
                ou dépendant de l'une des interfaces de f
             faire cohérence (tr.text)

            pour toute réalisation tr important une des interfaces I,
               de f et tr n'est pas dans la liste "use" de t.text
               associé à I faire
            début
               tr.text.état <-- incohérent
               incomplet (tr)
               pour tout code objet c∈ tr.text.co faire périmer (c)
            fin

          fin modifier

     fin constraints

end contraintes_famille
```

Le type text_contr_famille est défini comme suit :

```
type text_contr_famille is new text

     structure

          def
               text : éditeur_ci
          sel
               text : éditeur_ci
          use
               text : éditeur_ci

     end structure

     satisfy ci

end text_contr_famille
```

4.2.2 Contraintes interface

ce document est le type de l'attribut "man" d'une interface, il est défini comme suit :

```
type contraintes_interface is new document

    --attributs

    text (1,1)      is   text_contr_interface
    état            is   (cohérent, incohérent)

    satisfy manip

    --contraintes statiques

    constraints

        t : contraintes_interface

    modifier (t.text) :

        début

          si non cohérence (t.text) alors t.état <-- incohérent
          pour toute rélisation tr englobant une des réalisations de
               l'interface I ayant cet attribut, ou dépendant de I
               faire cohérence (tr.text)
        fin

        -- traitement des modifications des restrictions d'interfaces

    fin constraints

end contraintes_interface.
```

Le type text_contr_interface est défini comme suit :

```
type text_contr_interface is new text

    structure

        def
             text : éditeur_ci
        sel
             text : éditeur_ci
        incl
             text : éditeur_ci

    end structure

    satisfy ci

end text_contr_interface
```

4.2.3 Contraintes réalisation

Ce document est le type de l'attribut "man" d'une réalisation, il est défini comme suit :

```
type contraintes_réalisation is new document

    --attributs

    text (1,1)      is   text_contr_réalisation
    état            is   (cohérent, incohérent)

    satisfy manip

    --contraintes statiques

    constraints
        t : contraintes_réalisation
        r : réalisation ayant l'attribut t

    modifier (t.text) :

        début

          si non cohérence (t.text) alors t.état <-- incohérent

          pour toute réalisation tr englobant r ou dépendant de
              l'interface de r faire
                  cohérence (tr.text)

        fin modifier

    fin constraints

end contraintes_réalisation
```

Le type text_contr_réalisation est défini comme suit :

```
type text_contr_réalisation is new text

    structure

        def
            text : éditeur_ci
        sel
            text : éditeur_ci
        attr
            text : éditeur_ci
        util
            text : éditeur_ci

    end structure

    satisfy ci

end text_contr_réalisation
```

4.2.4 Espace travail

Ce document est le type de l'attribut "programmeurs" d'une famille, il est défini comme suit :

```
type espace_travail is new document

     --attributs

     text (1,1)          is   text_contr_espacetravail


     constraints

          t : espace_travail
     modifier  : conérence (t)

     fin constraints

end espace_travail
```

Le text_contr_espacetravail est défini comme suit :

```
type text_contr_espacetravail

     structure

          nom_personne    :    char (32)
          id_espace       :    char (16)
          nom_espace      :    char (114)

     end structure

     satisfy ci

end text_cont_espacetravail
```

4.3 Définition des structure introduites

Les structures utilisées, dans l'attribut "text" des documents des contraintes d'intégrité précédentes, permettent d'identifier le type des contraintes qu'elle décrivent. On distingue deux types de contraintes :

- des contraintes de décomposition de systèmes (use, incl, et programmeurs),

- des contraintes de composition de systèmes (attr, sel, def, et util).

Nous les étudions respectivement dans les paragraphes qui suivent.

4.3.1 Contraintes de décomposition

Dans ADELE, les contraintes de décomposition de systèmes permettent essentiellement d'établir la visibilité entre familles, la restriction entre interfaces d'une même famille, et la répartition des programmeurs sur les différentes parties du système. Le système de gestion de la base doit maintenir ces contraintes pour assurer la cohérence et l'intégrité de la base.

- Contraintes de visibilité (use) : pour des raisons de protection, les interfaces et réalisations d'une famille F ne peuvent être utilisées implicitement que par le père de F (si elle en a un). Pour qu'une réalisation d'autres familles, FU puisse utiliser les entités de F, il faut que le concepteur de F donne, dans la structure "use" la liste des interfaces, que que FU peut utiliser, de la manière suivante :

<nom_famille>("("<id_interface>(","<id_interface>)*")")

Exemple :

```
>F1>F2
<F3 (I1 , I2)
<*
```

Si la liste des interfaces est vide, alors toutes les interfaces de la famille sont visibles à la famille désignée. Les contextes déclarés par l'usager (noms des familles utilisatrices) ne sont acceptés que si ceux ci ne risquent pas d'introduire des circuits dans le graphe de dépendance. En effet les extensions des droits d'accès ainsi définis permettent à la relation de dépendance de décrire un graphe orienté, des circuits peuvent donc apparaitre,

- contraintes de restriction (incl) : la restriction est une relation définie entre deux interfaces d'une même famille. Une interface A est une restriction d'une interface B, si A fournit un sous-ensemble des ressources fournies par B.

Un exemple est donné au (A1.1)

- contraintes de répartition des programmeurs (programmeurs) : dans les SGBD, ces contraintes correspondent à des schémas externes, qui permettent de délimiter les entités et les relations qu'un groupe d'usagers peut manipuler. Pour ce groupe d'usagers, la base se réduit donc aux schémas externes qu'il en a d'elle.

Dans ADELE, chaque famille correspond à un schéma externe de la base, appelé espace de travail. Les familles étant hiérarchisées par l'attribut "familles_filles", les espaces de travail le sont aussi (fig.4.1). Un espace de travail E défini par une famille F est formé de tous les espaces de travail définis par les familles obtenues par la fermeture transitive de l'attribut "F.familles_filles". Ces espaces sont appelés sous espaces de travail de E.

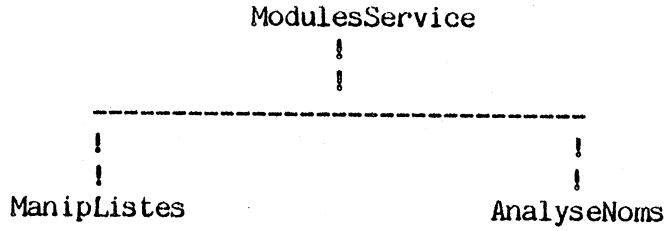


Fig.4.1 Hiérarchie des espaces de travail.

L'attribut programmeurs d'une famille permet d'associer à l'espace de travail qu'elle définit un ensemble d'usagers. Ces usagers peuvent créer, détruire, modifier les entités de cet espace, et consulter une entité dans n'importe quel espace. Un usager peut être associé à un espace de travail E par tout usager de cet espace, ou par lui même si E est un sous espace de l'un de ses espaces.

Tout usager de la base a un espace de travail par défaut, celui qu'il s'est associé comme tel, ou le premier auquel il était associé.

Cette notion de répartition des programmeurs sur les espaces de travail est couramment utilisée dans les ateliers de logiciel (on peut citer par exemple CADES, et CDL2 (BAY 81)).

4.3.2 Contraintes de composition

Les contraintes de composition consistent essentiellement en la designation de versions de réalisations à choisir (cas de l'implication) ou à ne pas choisir (cas de l'incompatibilité) dans une réalisation composée donnée. Le système de gestion de la base ADELE, doit construire les réalisations composées selon ces contraintes, et assurer la cohérence et l'intégrité de ces réalisations en tenant compte de l'évolution de ces contraintes.

Nous commencerons par la définition des structures utilisées pour l'expression de ces contraintes, et nous terminerons par la description de la méthode de construction d'une réalisation composée.

4.3.2.1 Définitions

- attr (attributs) : permet de définir les attributs des réalisations. C'est donc un moyen pour définir les caractéristiques de chaque réalisation, et peut être considéré comme un moyen complémentaire de désignation. La syntaxe de cette structure est la suivante :

<id_attribut> = <valeur_attribut>

l'opérateur "=" affecte la valeur <valeur_attribut> à l'attribut <id_attribut>. Des exemples sont donnés au (A1),

- sel (sélection) : cette structure définit les contraintes de sélection impératives qu'une entité E impose (désignation des réalisations impliquées par, ou incompatibles avec, E). L'opération de sélection est le choix des composants qui réalisent un système particulier. Nous avons restreint le choix à celui des versions de réalisations : les interfaces sont choisies une fois pour toutes dans le texte des corps (pour des raisons de compilation séparée) et les révisions choisies sont les plus récentes.

Les contraintes de sélection sont donc exprimées uniquement par des caractéristiques des versions (les attributs et leurs valeurs). Si l'une de ces contraintes ne peut pas être respectée alors l'atelier considère qu'une erreur est détectée. Les expressions de contraintes les plus importantes peuvent être formalisées par les règles suivantes :

<contr> ::= <sel_libre> / <sel_par_attr> /
<sel_nominative> / <sel_alternée>

<sel_libre> ::= <nom_interface>

<sel_par_attr> ::= { (<nom_famille>) / (<nom_interface>) } (<id_attribut> { : / = } <valeur_attribut>)

<sel_nominative> ::= <nom_réalisation>

<sel_alternée> ::= "or" "(" <contr> <liste_contraintes> ")"

<liste_contr> ::= (","< contr>)*

Les noms sont interprétés dans le contexte de l'entité qui impose ces contraintes, dans son attribut "man". Le symble "*" peut être utilisé pour remplacer n'importe quel nom de famille. Des exemples donnés dans (A1), nous prenons ces contraintes :

```
ManipStructure_i1   --dans SystèmeGestionBase_com_rc2.text
Compilation (trace = faux) --dans ManipAttributs_i1_c1.man
* (version : minimale) --ManipAttributs_i1_c1.man
or (ManipAttributs_i1_c1, ManipAttributs (auteur = Pierre))
                                    -ManipAttributs_i1_R.text
```

qui signifient respectivent :

- la réalisation composée SystèmeGestionBase_com_rc2 impose que les réalisations qui seront choisies avec elle pour constituer une réalisation composée doivent importer l'interface ManipStructure_i1 et choisir n'importe laquelle de ses réalisations (c'est une sélection libre),

- le corps ManipAttributs_i1_c1 impose que toutes les réalisations qui partagent avec lui la famille Compilation doivent choisir une version qui a l'attribut "trace" = "faux",

- le corps ManipAttributs_i1_c1 impose que si l'attribut "version" est présent dans au moins une des réalisations, des familles descendantes, alors la réalisation à choisir doit avoir cet attribut avec la valeur "minimale" ; sinon choisir n'importe quelle réalisation de ces familles,

- la réalisation composée ManipAttributs_i1_R impose dans son texte qu'il faut choisir soit la réalisation ManipAttributs_i1_c1 ou n'importe quelle autre réalisation qui a l'attribut "auteur" = "Pierre",

- def (défaut) : cette structure est un assouplissement de la précédante, car ses contraintes sont préférentielles alors que les autres sont obligatoires. Ses contraintes ne sont donc prises en compte que s'il n'existe pas de contraintes imposées qui portent sur la même réalisation, ou si elles sont compatibles. L'atelier essaie de choisir une version de réalisation qui satisfait en même temps les deux types de contraintes si elles existent. Si le choix ne peut être fait alors l'atelier choisira une version qui satisfait les contraintes imposées (voir l'algorithme de sélection de réalisations en (A2)). La syntaxe de cette structure est la même que celle de "sel". Des exemples sont donnés au (A1),

- util (utilisabilité) : l'usager indique dans la structure "util", si la réalisation est disponible ou non. Une réalisation indisponible ne peut être choisie dans une construction de réalisation composée.

4.3.2.2 Construction d'une réalisation composée

PRINCIPE

Nous allons détailler la méthode décrite au (4.1). Cette méthode est essentiellement basée sur le texte de la réalisation composée à construire, la relation de dépendance "import", et les attributs "man" de chaque entité d'ADELE (famille, interface, et réalisation) décrites précédemment.

C'est essentiellement la description de l'opération cohérence de la propriété ci, décrite dans le schéma conceptuel au (3.2.2.3), appliquée au texte de la réalisation composée en cours de construction et qui satisfait cette propriété. Elle montre donc comment sont utilisées les informations fournies dans le schéma conceptuel et leurs adaptabilité pour construire une réalisation composée. L'algorithme implantant cette méthode est décrit dans l'annexe (A2). La méthode se divise en deux étapes :

1) une phase d'initialisation qui consiste en :

- la prise en compte des contraintes décrites dans le texte de la réalisation composée à construire R, et dans son attribut "man",

- la prise en compte des contraintes de composition, décrites dans l'attribut "man" de la famille de R,

- la prise en compte des contraintes de composition, décrites dans l'attribut "man" de l'interface I réalisée par R,

- selon cet ensemble de contraintes, choisir un corps C qui réalise l'interface I. I sera l'interface de la réalisation composée, et C sera le corps principal de cette réalisation,

- la prise en compte des contraintes de C,

2) une phase de construction récursive qui consiste en :

- pour toute interface I importée par la réalisation choisie, tenir compte des contraintes de I et de celles de sa famille,

- selon les contraintes qui portent sur la version de la réalisation à choisir pour I, choisir une version et tenir compte des contraintes décrites dans son "man", ...

Nous trouverons au (A1.2) un exemple complet de construction de réalisations composées.

Le choix possible des versions est limité, en plus des contraintes de compositions, par deux contraintes de l'atelier ADELE :

- tous les utilisateurs (réalisations) d'une famille partagée (utilisée donc par l'opération "import") doivent utiliser la même interface et la même version de réalisation de cette interface,

- Les contraintes de composition, décrites dans les attributs "man" des composants sélectionnés doivent être respectées

OPTION PAR DEFAUT

Dans ADELE, le mécanisme de désignation est assoupli par des possiblités de désignation par défaut (cf. 3.2.3.2). Il est donc possible, même si le texte de la réalisation composée à construire, ainsi que les attributs "man" des composants, noeuds du graphe de dépendance, sont vides, de construire cette réalisation en choisissant ses composants par le mécanisme de la désignation par défaut.

CONFLITS DE SELECTION

Lors de la sélection d'une version, les contraintes qui la déterminent peuvent entrer en conflit :

- conflits de sélection par défaut (def) : si des contraintes de cette structure entrent en conflit, et que les contraintes imposées ne peuvent pas résoudre ce conflit, alors la contrainte imposée par le module le plus haut sera prise,

- conflits des contraintes imposées (sel) : a priori tout conflit entre sélections imposées aboutit à une erreur. Toutefois un assouplissement existe : si deux interfaces différentes sont importées dans une même famille, il n'y a pas de conflit si l'une de ces interfaces est une restriction de l'autre. L'interface la plus étendue sera choisie.

EFFETS DE LA CONSTRUCTON

1) Etat du texte de la réalisation composée

- Si un conflit ne peut être résolu, le texte de la réalisation composée à construire sera dans un état incohérent,

- si aucun conflit n'est détecté pour le choix d'une réalisation d'une interface donnée, et s'il n'existe pas de réalisation ou si aucune des réalisations existantes ne satisfait les contraintes imposées, la réalisation à construire sera dans un état incomplet. Il en est de même si une des réalisations choisies est elle même incomplète ou incohérente,

- si une réalisation composée est modifiée, le sytème essaiera de compléter toutes les réalisations qui peuvent l'être. L'usager dispose aussi d'une commande de reconstruction qui permet de reconstruire une réalisation en essayant de la compléter si elle ne l'était pas.

2) liste de composition, liste de dépendance

Si une réalisation composée est dans un état cohérent, le système met la liste des composants qui ont été choisis, dans l'attribut "list_comp" de cette réalisation. Cependant le choix de certains composants doit être retardé. Ces composants n'apparaissent donc pas dans la liste de composition "list_comp" de la réalisation, mais ils sont pris comme dépendances de cette réalisation. Comme pour un corps, une réalisation composée R1 dépend d'une famille F, si F est partagée avec une autre réalisation R2 qui n'est pas descendante de R1. L'atelier impose que toutes les réalisations qui se partagent une famille, se partagent également les mêmes interfaces et réalisation pour cette famille. Si R1 choisit une version V1, et R2 choisit une version V2, alors toute réalisation composée R, qui englobera les réalisations R1 et R2 (contenir tous leurs composants) aura V1 et V2 pour un même composant partagé, ce qui rend la contrainte précédente non respectée. Il est donc nécessaire de retarder ce genre de choix.

Toutes les interfaces dont dépend une réalisation composée seront calculées automatiquement par le système et placées dans l'attribut list_dependance de cette réalisation composée. Les contraintes que cette réalisation composée impose pour le choix des réalisations des interfaces dont elle dépend sont également conservées par le système dans l'attribut "list_contr_héritées" (cf. 5.3.3.3) ; elles seront utilisées quand cette réalisation est choisie dans la composition d'une autre (un exemple est donné au (A1.2)).

Le choix d'une version V dont dépend une réalisation composée, est effectué lors de la construction d'une réalisation composée qui englobe toutes les réalisations qui se partagent cette version V, ou par décision de l'usager de créer une version autonome de sa réalisation composée (pour des tests, ...).

4.4 Evolution d'un logiciel

Nous distinguons deux types d'évolution : évolution des textes des entités et évolution de la structure de ces entités.

4.4.1 Evolution des textes des entités

Dans la littérature nous constatons que l'évolution des textes et ses conséquences est un problème actuel. Nous trouvons dans (IVI 77, KAI 82b, SCH 82, TIC 82) des solutions à ce problème sous formes d'outils utilisés dans les ateliers de logiciels. Dans ce qui suit, nous résumons la solution d'ADELE, décrite en grande partie dans le schéma conceptuel spécifié au chapitre précédent. Les mécanismes de versions multiples et de révisions, similaires dans leur spécification à ceux définis dans SVCE (KAI 82b), permet de faire évoluer les textes :

- versions d'interfaces : par le mécanisme de la relation de restriction on peut étendre ou diminuer les ressources accessibles d'un module. Le système maintient la base dans un état cohérent au cours de cette évolution, en respectant les contraintes décrites dans la structure "incl" de l'attribut "man" des interfaces,

- versions multiples de réalisation, révisions : Une version est une réalisation particulière d'une interface. Une interface peut donc avoir plusieurs versions de réalisation. Une version peut différer d'une autre par son implantation des ressources, ou par ses dépendances. Une version de corps peut exister en plusieurs révisions, qui diffèrent les unes des autres par des modifications mineures (correction d'anomalies, options de mises au point, ...). Le système assure la cohérence de cette évolution.

Nous résumons ici les effets des modifications des attributs "text" décrits dans le schéma conceptuel de la base ADELE, sous forme de contraintes statiques.

- modification d'une réalisation : dans la mesure où la nouvelle réalisation respecte les définitions de l'interface à laquelle elle est associée, la modification est logiquement invisible aux utilisateurs. En fait les codes objets associés à ce corps doivent être périmés de même que tous les codes objets incluant directement ces codes objets périmés, et récursivement. La liste de ces codes objets périmés est fournie à l'usager désirant modifier la réalisation, pour confirmation ou infirmation. Les codes objets périmés seront détruits explicitement par leur responsable, ou automatiquement lors du catalogage d'un nouveau code objet de même type,

- modification d'une interface : si une interface est modifiée, les réalisations qui lui sont associées ou qui en dépendent directement deviennent incohérentes. Les réalisations marquées incohérentes sont considérées comme modifiées : leurs codes objets et ceux des réalisations qui les englobent sont marqués périmés. La liste des réalisations atteintes par la modification est fournie à l'usager. Si celui-ci confirme sa décision de modifier l'interface, les actions précédentes sont automatiquement effectuées.

- modification des contraintes d'intégrité : la modification de la structure "sel" ou de la relation "rel_import" d'une réalisation peut induire la modification de la liste de composition de certaines réalisations. Celles-ci sont donc à reconstruire. Cette reconstruction (automatique) peut entrainer d'autres modifications (list_contr_héritées, list_dependance). L'usager est averti des réalisations qui vont être atteintes par la modification des contraintes d'intégrité. Si celui-ci persiste, toutes les réalisations atteintes sont reconstruites. Les anciens codes objets de ces réalisations sont marqués périmés.

La modification de la structure "use" peut rendre invalides des relations d'importation. Les corps atteints sont marqués "incohérents" et il est procédé comme pour la modification d'une réalisation.

Il est au préalable vérifié que tout attribut "man" d'une entité est cohérent. Pour cela on simule la création d'une réalisation composée ayant uniquement cet attribut comme texte. Si cette réalisation simulée ne peut être construite, alors le texte de cet attribut est considéré dans état incohérent.

- modifications concurrentes : Le fait que plusieurs postes de travail fonctionnent en parallèle pose le problème de l'accès concurrent à la base dans les cas suivants :

  - modification concurrente d'un même objet,
  - modification concurrente de deux objets dépendants,

Ce problème est traité au (4.5).

A la création d'une version ou révision, le système garde la base dans un état cohérent, en reconstruisant toutes les réalisations qui sont affectées par cette création ou modification. Dans l'état actuel, on ne dispose pas de mécanismes d'optimisations qui permettraient de ne recompiler que les composants effectivement atteints par cette évolution, car la compilation se fait hors de la base. Nous périmons donc les codes objets potentiellement atteints. De même nous n'avons pas de mécanismes qui déterminent les objets qui doivent obligatoiremnet être modifiés à cause de cette évolution ; nous rendons incohérents tous les objets potentiellemnt atteints.

Le système assure, toujours, après une modification :

- la compatibilité des interfaces entre les modules d'une réalisation : le système rend incomplet toutes les réalisations composées qui englobent des corps qui dépendent ou réalisent les interfaces modifiées. Ces réalisations ne peuvent devenir complètes (donc utilisables) qu'après l'adaptation de ces corps aux nouvelles interfaces,

- la reconstruction automatique des réalisations composées, affectées par ces modications, en les composant des nouvelles versions.

4.4.2 Evolution structurelle

Les types d'entités étant figés, ainsi que les opérations qui assurent la maintenance des contraintes statiques relatives à la structure du système, les systèmes ne peuvent donc évoluer structurellement.

4.5 Modifications concurrentes

La vocation de la base est celle d'un serveur partageable. On doit donc résoudre deux types de problèmes : la protection entre les utilisateurs, et les incohérences pouvant résulter de l'exécution parallèle de requêtes émises par plusieurs usagers.

4.5.1 Protection entre usagers

La protection des usagers entre eux est réalisée par plusieurs mécanismes complémentaires : la protection fournie par les espaces de travail, la limitation de l'utilisation des familles par "use", la disponibilité des entités assurée par "util", et la réservation des objets que nous décrivons ici. La commande "reserve", appliquée à une réalisation ou à une interface, assure à l'auteur de cette demande qu'il sera le seul autorisé à manipuler cet objet. La réservation d'une entité entraine la réservation de toutes les entités que sa modification risque de rendre incohérentes. L'objet est réservé jusqu'à sa libération explicite par l'usager, ou implicite par le système (lors d'une opération de catalogage par exemple).

Ce mécanisme de réservation ressemble à ceux utilisés dans RCS (TIC 82) et dans SVCE (KAI 82b). La réservation dans ADELE est faite au niveau d'une version, alors que dans RCS elle est faite au niveau d'une révision. Le problème de réservation des versions impliquées dans une relation avec la version modifiée n'était pas dans le cadre de RCS et n'est pas pris en compte dans SVCE.

4.5.2 Actions parallèles

A de rares exceptions près, il n'est pas possible de dire, a priori, compte tenu des relations parfois lointaines entre les objets, si l'exécution parallèle de deux requêtes peut conduire à des incohérences. La solution consistant à n'exécuter qu'une requête à la fois ne peut convenir vu le temps d'exécution des commandes interactives. Nous permettrons les actions parallèles en utilisant des mécanismes classiques de transactions et de reprise. Ce mécanisme n'est pas encore défini dans la base, mais il est bien résolu et depuis longtemps (voir par exemple la solution prise dans GALAAD, ou celle prise dans SSP).

5 REALISATION DU SYSTEME DE GESTION DE LA BASE ADELE

Nous commençons par présenter le contexte de la réalisation, l'utilisation, et l'implantation physique du système de gestion de la base ADELE. Nous terminons par sa portabilité. Nous montrons le long de cette présentation comment sont réalisées les entités et relations décrites dans le schéma conceptuel de la base aux (3.2, 4.2, et 4.3).

5.1 Contexte de la réalisation

La configuration matérielle prévue, comporte un réseau local, du type Danube, auquel seront connectés des postes de travail individuels et un serveur central. Ce dernier est un HB 68 fonctionnant sous MULTICS ; il est utilisé pour l'archivage et pour des fonctions nécessitant une grande puissance de calcul. Les fonctions interactives sont réalisées sur des postes de travail qui sont des ordinateurs individuels, ou des terminaux directement connectés au serveur central.

En plus des outils classiques (éditeurs des textes, compilateurs, ...) MULTICS offre un système de gestion de bases de données relationnelle MRDS (MRD 81).

5.2 Utilisation du système

La base n'est pas invisible à l'usager, comme c'est le cas par exemple dans SSP (LAP 81). Elle peut être utilisée aussi bien directement par l'usager (création, destruction d'entités, ...) ou indirectement à travers les outils (éditeur syntaxique, décompilateur, ...).

Les textes des objets n'étant pas analysés par le système de gestion de la base, mais par les outils qui les ont créés ou qui les manipulent, la base est donc indépendante des textes qu'elle gère. Elle peut contenir des programmes écrits en n'importe quel langage, leurs représentations internes, et leurs codes objets et documents.

Le système de gestion de la base peut donc être utilisé en dehors de l'atelier ADELE (indépendamment de ses outils) pour gérer (selon les concepts déjà définis) des logiciels à versions multiples. Notre expérience de ce système, dans cette optique, sera décrite au chapitre 6.

Le système offre une interface commune à l'utilisation directe (usager) et indirecte (à travers les outils). Cette interface consiste en un ensemble de procédures qui permettent toute manipulation des données dans la base, indépendamment de leur structure interne. Ces procédures sont de deux types : manipulation de la structure des systèmes (création, destruction d'entités, ...) qui sont réalisées dans l'exemple (3.1.4) par le module ManipStructure, et la manipulation des attributs des entités (modification des textes sources des entités, modification et vérification des contraintes d'intégrité, ...) qui sont réalisées dans l'exemple (3.1.4) par le module ManipAttributs. Cette interface est donc le seul moyen d'accès à la base. On peut ajouter des outils, ou modifier ceux déjà existant, sans que la base soit affectée, tant que les besoins de ces outils correspondent toujours aux fonctionnalités offertes par cette interface.

Sous ADELE, la base peut être manipulée par une interface usager "navigationnelle", avec une désignation directe des objets sur l'écran. Elle est basée sur la notion "objet-opération", qui pour chaque objet désigné sur l'écran, affiche l'ensemble des opérations qui peuvent lui être appliquées (nous trouvons un résumé de cette interface dans (CHV 83)).

5.3 Implantation

5.3.1 Problèmes d'utilisation de MRDS

Au moment où nous avons commencé à examiner les possibilités d'implantation de la base (GHO 82), une expérience d'utilisation de MRDS dans l'atelier de logiciels SPRAC (2.2.5) avait été faite. Cette expérience nous a été profitable puisqu'elle nous à permis de connaitre les problèmes pratiques qui se posent lors de l'utilisation de MRDS pour la gestion des logiciels (FOI 82a).

En plus de ces problèmes cités au (2.2.5.3), nous exposons un problème important de protection : MRDS crée la base dans l'anneau de protection le plus faible (les détails de la notion d'anneaux sont décrits dans (MUL 79)).

Un usager qui a le droit de créer une relation ou de la détruire peut exécuter ces opérations sans être sous le contrôle de MRDS : se charger (dans l'anneau de la base) sous le "directory" qui contient la relation à créer ou à détruire, et exécuter les commandes MULTICS réalisant ces fonctions. Ce problème aurait pu être évité si MRDS implantait la base dans des anneaux sur lesquels les usagers ne peuvent faire de "login" (les anneaux 1, 2, et 3). Les objets qui sont dans ces anneaux ne peuvent être manipulés qu'à travers des "guichets" ; ensemble de procédures, qui définissent les seules actions possibles sur ces objets. Il sera donc impossible de manipuler la base autrement que par ces procédures, qui constitueront l'interface du système de gestion de la base de données.

Pour ces diverses raisons, nous avons développé notre système de gestion de la base qui sera implanté dans l'anneau 3 de MULTICS.

5.3.2 Différents niveaux du système

Notre système est découpé en trois couches (fig.5.1) :

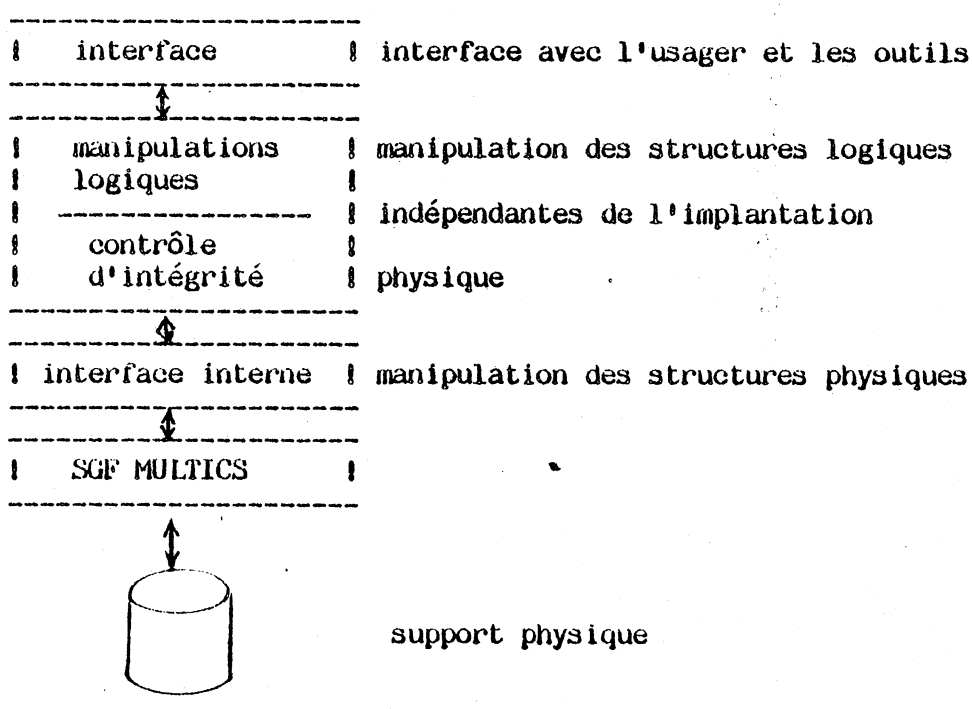


Fig.5.1 Différents niveaux d'abstraction du système de gestion de la base ADELE.

- la couche interface est composée de modules qui réalisent la recupération et l'analyse des paramètres, la détermination des options prises par défaut, le dialogue éventuel avec l'usager, ...

- la couche manipulations logiques, contrôle d'intégrité est composée de modules qui manipulent les entités (telles qu'elles sont décrites dans le schéma conceptuel) indépendamment de leur structures physique, tout en assurant la cohérence de la base. Ces modules sont de trois sortes : les modules qui gèrent la structure des entités (ManipStructure), les modules qui gèrent les attributs des entités (ManipAttributs), et des modules de service, qui sont utilisés par les deux autres types (ModulesService),

- la couche interface interne est réduite à un seul module (StructPhysique) qui utilise les fonctions du SGF MULTICS pour implanter physiquement les entités et quelques relations du schéma conceptuel.

5.3.3 Implantation des entités et relations

5.3.3.1 Structure physique des entités

Les fonctions des différentes couches étant déjà décrites, nous exposons ici les structures physiques utilisées pour implanter les entités et leurs attributs :

- les entités sont représentées physiquement par des annuaires ("directories") de MULTICS,

- les attributs (man, doc, et text) des entités sont représentés par des segments MULTICS attachés aux entités qu'ils décrivent.

Dans ce qui suit, nous décrivons les représentations internes des attributs relations, de l'attribut multivalué "text", de l'attribut "man", et de l'état de chaque entité.

Toutes ces représentations internes des informations relatives à une entité et ses "text" (ses relations, ses contraintes, et son état) sont conservées ensemble dans un segment, appelé "man" (manuel de l'entité). La structure de ce segment se compose de trois sous-structures : représentation interne des relations, représentation interne des contraintes d'intégrité, et représentation interne de l'état de l'entité.

5.3.3.2 Représentation interne des relations

Les attributs de type relation (définis dans le schéma conceptuel de la base) d'une même entité, sont regroupés dans une même structure, représentation interne des relations, définie dans le segment "man" de cette entité. Ces relations sont conservées sous forme de listes de noms, définis par la syntaxe de désignation d'ADELE (cf. 3.2.3). Cette structure est seulement consultable par les usagers, les initialisations et les mises à jour sont automatiquement assurées par le système de gestion de la base qui gère toutes ces relations (cf. 3.2.2.4). Les relations structurelles (familles_filles, familles_externes, interfaces, réalise, et l'appartenance des attributs aux entités) sont, en plus, gérées physiquement par MULTICS, et représentées dans la fig.5.2.

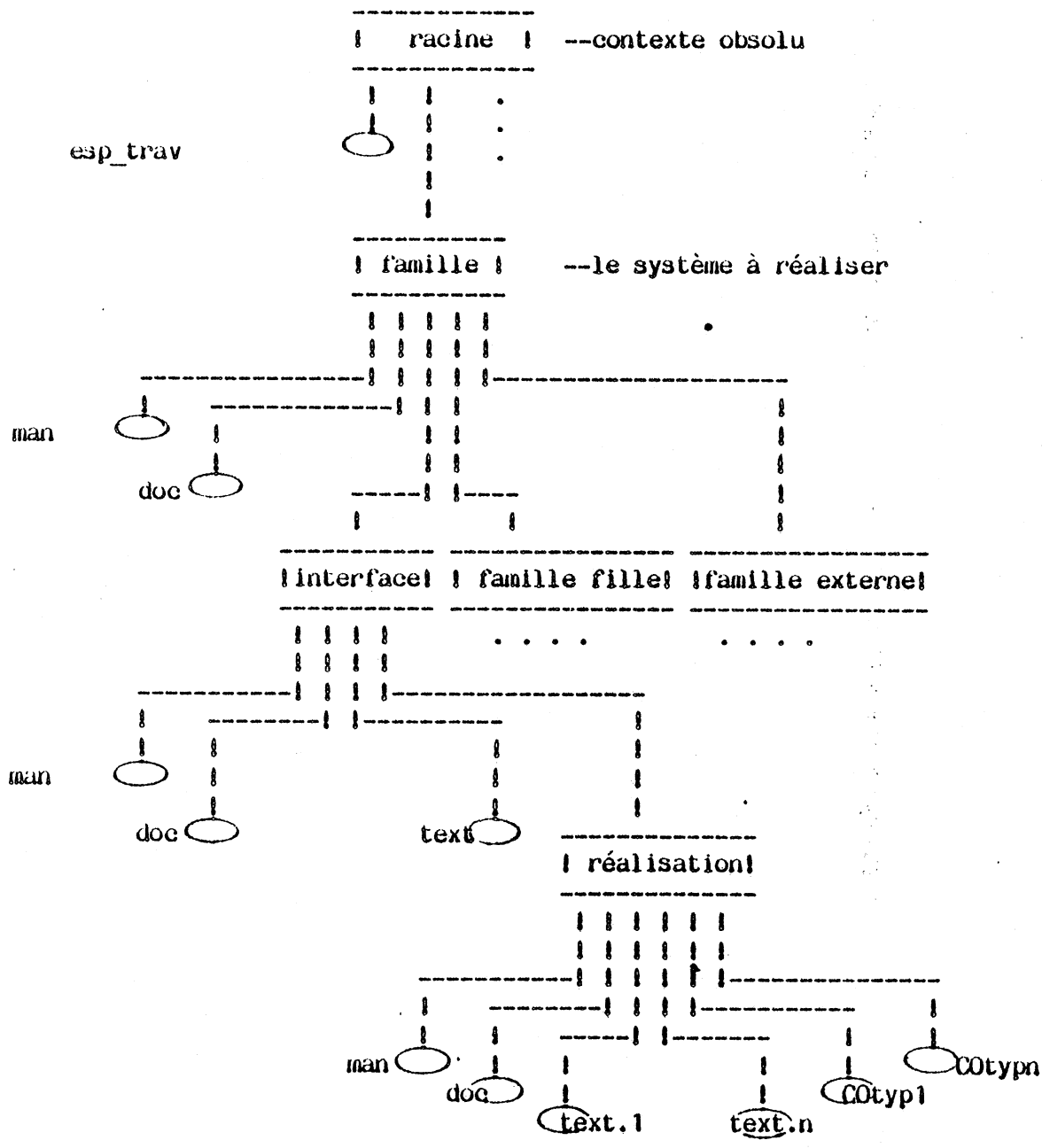


Fig.5.2 Structure interne d'un système sous ADELE.

- Représentation interne des relations d'une famille : la structure représentation interne des relations, du segment "man", associé à une famille F, contient les listes suivantes :

- liste_familles_filles : liste de toutes les familles filles de F. Elle concrétise l'attribut F.familles_filles. La relation inverse (fille_de) est implicite, puisqu'elle peut être obtenue (sans ambiguïté) à partir du nom de F,

- liste_familles_externes : liste de toutes les familles externes à F. Un élément de cette liste est de la forme sivante (nom_local, nom_famille_externe). Le "nom_local" est l'identificateur du lien et le "nom_famille_externe" est le nom ADELE de la famille externe. Cette liste concrétise l'attribut F.familles_externes,

- liste_familles_liées : liste de toutes les familles qui ont fait un lien sur F (pour lesquelles elle est déclarée comme externe). Elle concrétise l'attribut F.externe_à. Cette liste ne contient que les familles qui sont effectivement autorisées à utiliser F,

- liste_interfaces : liste des interfaces de la famille F. Elle concrétise l'attribut F.interfaces.

- Représentation interne des relations d'une interface : la structure représentation interne des relations, du segment "man", associé à une interface I, contient les listes suivantes :

- liste_corps : liste des corps réalisant l'interface. Elle concrétise un sous-ensemble des valeurs de l'attribut I.réalisée_par,

- liste_realcomp : liste des réalisations composées réalisant cette interface. Elle concrétise donc l'autre sous-ensemble de l'attribut I.réalisée_par,

- liste_dépendance_inverse : liste de toutes les réalisations qui dépendent de I (utilisation par la clause "import"). Elle concrétise l'atribut I.text.importé_par,

-liste_environs_inclus : liste des environs que le texte de cette interface inclut. Elle concrétise l'attribut I.text.inclut,

- liste_environs_incluants : liste de tous les environs qui incluent le texte de I (si I est un environ, sinon cette liste est vide). Elle concrétise l'attribut I.text.inclus_par.

l'attribut "I.interface_de" est directement déductible du nom de I, en prenant le nom de sa famille.

- Représentation interne des relations d'une réalisation : la structure représentation interne des relations, du segment "man", associé à une réalisation, R, contient les listes suivantes :

- liste_dépendance : liste de toutes les interfaces dont dépend R. Elle concrétise l'attribut R.text.list_dependance,

- liste_réalisation_englobantes : liste de toutes les réalisations composées qui englobent R. Elle concrétise l'attribut R.text.composant_de,

-liste_composition : liste de toutes les réalisations qui composent (englobées par) R, si R est une réalisation composée, sinon elle est vide. Elle concrétise l'attribut R.text.list_comp.

- Représentation des attributs multivalués : seul, l'attribut "text" d'un corps est multivalué, et nous avons vu que chaque "text" définit une révision du corps. Nous avons considéré cet attribut comme une relation et l'avons donc représenté sous forme de liste (liste_révisions) dans la représentation interne des relations d'une réalisation. Comme chaque "text" peut avoir plusieurs types de codes objets, nous avons défini une autre liste (liste_codes_objets), dans la même struture, qui contient les codes objets de toutes les révisions des réalisations.

5.3.3.3 Représentation interne des contraintes

Les contraintes associées à une entité sont représentées dans la structure représentation interne des contraintes, du segment "man" associé à l'entité. Cette structure est formée des listes (sauf pour la représentation de la contrainte "util") :

- liste_incl : liste simple de noms d'interfaces,

- liste_incl_inverse : liste simple de noms d'interfaces, obtenue par la fonction inverse de la contrainte "incl",

- liste_use : liste ayant la structure (nom de la famille autorisée, liste des interfaces visibles à cette famille),

- liste_attr : liste ayant la structure (nom d'un attribut, opérateur (: ou =), valeur de cet attribut),

- liste_def : liste ayant la structure (nom d'une entité , liste_attr),

- liste_sel : liste ayant la strcuture (liste_def),

- liste_sel_héritées : même structure que liste_sel,

- util : boolean.

Une exception est faite pour les contraintes de répartition des programmeurs dans la base. Toutes les contraintes de ce type (définies sur des familles de la base) sont regroupées dans un segment "esp_trav" (cf. fig.5.2) associé à la racine (contexte obsolu dans ADELE). Ce segment contient des listes de structures définies au (4.2.4).

5.3.3.4 Représentation interne de l'état d'une entité

Chaque entité a un état, qui est en fait plus compliqué que celui décrit dans la schéma conceptuel. La structure de la représentation interne de l'état contient :

- nom : nom ADELE de l'entité,

- niveau : entier, le niveau, tel qu'il est défini au (3.1.3), de l'entité dans le graphe de dépendance. Une famille a le même niveau que ses interfaces et réalisations,

- vide : boolean, l'objet a été créé mais aucun texte n'a été fourni,

- en_cours_de_modification : (nom de l'usager, date) une interface, ou une réalisation est en cours de modification lorsque l'usager mentionné s'est réservé l'entité (par une commande de réservation),

- verrouillé : (liste des entités) une interface est verrouillée si elle est en cours de modification, ou lorsqu'un des corps qui dépend directement d'elle, ou qui la réalise, est en cours de modification. Un corps est verrouillé s'il est en cours de modification, ou lorsque l'une des interfaces dont il dépend, ou qu'il réalise, est en cours de modification. Les entités causes du verrouillage sont mentionnées dans cette liste. Une entité verrouillée ne peut être que consultée,

- incohérent : (liste des entités) une réalisation composée est dans un état incohérent lorsque sa construction conduit à des conflits de sélection non solubles. Un corps est incohérent lorsque les interfaces dont il dépend directement ont été modifiées ou détruites. Le texte de la réalisation incohérente doit être modifié par l'usager. Les interfaces, causes de cette incohérence, sont mentionnées dans cette liste,

- incomplet : (liste d'entités) une réalisation composée est incomplète lorsque certaines des réalisations à inclure n'ont pas pu être choisies (elles sont inexistantes, ou ne répondent pas aux critères imposés), sont incomplètes ou incohérentes,

- périmé : (liste des entités) un code objet d'une réalisation est périmés, si cette réalisation est modifiée ou si un des codes objets utilisés pour le constituer est détruit, périmé, ou modifié.

5.4 Portabilité du système

L'un des objectifs fondamentaux du système de gestion de la base ADELE était sa portabilité. Nous avons découpé le système (fig.5.1) de façon à limiter la dépendance de ce système par rapport à MULTICS, à la couche la plus basse "interface interne". Ceci nous permettra de modifier uniquement les fonctions de cette interface pour changer la structure interne des entités et relations. De plus nous avons pu réduire cette interface à un seul module qui réalise la création et la destruction physique des entités et relations selon le schéma de la fig.5.2.

La structure représentation interne des relations d'un segment "man", nous permettra de gérer logiquement, toutes sortes de relations, quelle que soit la structure physique de la base. Le système étant en grande partie écrit en Pascal, il peut (moyennant quelques adaptations) être portable sur toute machine disposant d'un compilateur Pascal semblable à celui de MULTICS.

6 EXPERIMENTATION ET EVALUATION DU SYSTEME DE GESTION DE LA BASE ADELE

6.1 Contexte de l'expérimentation

La base ADELE est en utilisation depuis quelques mois par des utilisateurs "privilégiés" que sont les membres de l'équipe ADELE. Le principal logiciel supporté par la base est son propre système, qui consiste en 15 000 lignes de code source, divisées en 30 familles (modules). La profondeur de l'arbre de décomposition (niveaux de la structure du système, défini au (3.1.4)) est de 10. Le système existe actuellement en trois versions, et certaines réalisations ont plus de 20 révisions.

Dans la version courante, la gestion des documents, et l'utilisation concurrente de la base n'ont pas encore été implantés.

A ce stade nous manquons beaucoup de données pour faire une évaluation rigoureuse. Cependant, il n'est pas négligeable de donner quelques indications basées sur notre implantation et notre utilisation de ce système.

6.2 Implantation

La base a été écrite en Pascal et en Pl/1, en 1 an par deux personnes. La version actuelle n'est pas complète. Elle sera complétée, étendue, et évaluée au cours de l'année (83-84). Les 15 000 lignes de code source qui la composent sont réparties sur les différents niveaux du système(cf. 5.3.2) comme suit :

interface	:	écrit en Pl/1 (8% du code),
manipulation et contrôle logique	:	écrit en pascal (90% du code),
interface interne (MULTICS)	:	écrit en Pl/1 (2% du code).

Afin de faciliter un portage éventuel, toutes les fonctions offertes par MULTICS n'ont pas été exploitées. La base utilise seulement les fonctions de création, de destruction, et de copie de segments.

Le système de gestion de la base d'ADELE étant écrit en grande partie en Pascal, nous avons rencontré des difficultés qui se posent pour l'implantation de toute grande application en Pascal, et qui sont connues comme étant ses insuffisances majeures :

- le problème des fichiers externes : un module en Pascal est un programme qui représente une unité de compilation. Tout programme (pris séparément) peut manipuler des fichiers externes ; mais lorsque ces programmes sont utilisés dans une application (ensemble de programmes) représentée par un programme principal un problème se pose. Pascal associe une sémantique particulière au mot clé "BEGIN" du programme principal : initialisation de tous les "blocs de contrôle" des fichiers externes déclarés dans le programme principal. Aucun autre fichier externe ne pourra être utilisé dans les programmes de l'application. Ceci oblige à déclarer, externes au programme principal, tous les fichiers externes d'autres programmes de l'application. Le programme principal doit alors exporter ces déclarations, et les programmes les utilisant doivent les importer,

- le problème des paramètres de procédures : Pascal fige le type de paramètres des procédures. Pour chaque type de paramètre il faut une procédure différente (même si ces types diffèrent, par exemple, uniquement par des dimensions des tableaux) ; ceci alourdit considérablement l'écriture des applications. Le compilateur ne vérifie, à la compilation d'un programme, que la correspondance des utilisations des procédures importées avec la déclaration de l'en-tête de celles ci, mais pas leur déclaration effective dans les programmes qui les exportent. Nous avons utilisé ce fait pour alléger cette contrainte.

Toutes les fois que celà nous a été possible (sans risque d'erreurs), nous déclarons une même procédure (avec des types de paramètres différents) autant de fois qu'il y a de types que nous estimons équivalents (des tableaux de même type à dimensions variable, des pointeurs, ...)

Les systèmes que supportera ADELE, devant être écrits en Pascal, ces deux problèmes et leurs solutions nous ont inspiré des mécanismes de récupération automatique, des déclarations des fichiers externes des programmes d'une application, et leur insertion dans le programme principal (voir (EST 83c)), et des mécanismes d'équivalence de types par la notion d'interfaces restreintes (voir (EST 83b)).

6.3 Utilisation de la base pour le développement de son système

Nous sommes arrivés à un stade où il était possible d'utiliser le système de gestion de la base pour continuer son développement. L'utilisation du système pour son propre développement et sa maintenance, a été pour nous un exemple réel des applications à supporter.

En plus des tests significatifs pour la mise au point, que cette utilisation nous a permis, nous avons pu faire une première expérimentation et évaluation des commandes offertes, des mécanismes de contraintes, de la propagation des types via les interfaces, d'évolution du système, et de l'efficacité des algorithmes.

6.3.1 Commande offertes

Nous avons constaté quelques insuffisances, dans les commandes, pour le support de grandes applications, telles que :

- l'espace de travail : étant sur un espace de travail courant EC, il fallait, pour manipuler une des entités d'une autre famille, se mettre dans l'espace défini par cette famille, faire la manipulation, et revenir à l'espace de travail initial EC. Nous avons constaté que cette séquence est très fréquente, coûteuse et désagréable. Nous avons alors permis de manipuler des entités de n'importe quel espace de travail (à condition d'y être autorisé) à partir de n'importe quel autre. Ce mécanisme c'est avéré très efficace,

- certaines séquences d'opérations (création d'une entité, sa réservation, et le transfert des valeurs de ses attributs) sont fréquentes. Nous avons alors fourni des transactions qui réalisent de telles séquences,

6.3.2 Mécanisme des contraintes

Cette évaluation est évidemment loin d'être complète, mais nous donnons une première constatation importante.

- évaluation du coût : l'association des contraintes de composition (def et sel) aux entités famille et interface s'est avérée excessivement coûteuse, car la modification d'une de ces contraintes peut amener à réanalyser toutes les réalisations composées dans la base.

En effet, la description partielle de l'opération "modifier", définie sur le type "contraintes_famille" d'une famille f (cf. 4.2.1), exprime que la modification d'une contrainte d'intégrité implique la reconstruction de :

- toutes les réalisations composées qui ont pour composant un des corps de f,

- toutes les réalisations composées qui dépendent d'une des interfaces de f.

Cet ensemble de réalisations composées Ef, est maximal.

La description de l'opération "modifier", définie sur le type "contraintes_interface" d'une interface I (cf. 4.2.2), exprime que la modification d'une contrainte implique la reconstruction de :

- toutes les réalisations composées qui ont pour composant un des corps de I,

- toutes les réalisations composées qui dépendent de I.

L'ensemble de ces réalisations Ei est donc un sous ensemble de Ef (il est relatif à une seule interface de f).

La description de l'opération "modifier" définie sur le type "contraintes_réalisation" d'une réalisation R (cf 4.2.3), exprime que la modification d'une contrainte implique la reconstruction de :

- toutes les réalisations qui ont pour composant R,
- toutes les réalisations qui dépendent de l'interface de R.

L'ensemble de ces réalisations Er est donc un sous ensemble de Ei (il est relatif à une réalisation de I). C'est un ensemble minimal.

bien que ces contraintes au niveau d'une famille et d'une interface, puissent être utiles (la mise en commun des contraintes pour toutes les interfaces et réalisations d'une famille, et la mise en commun des contraintes pour toutes les réalisations d'une interface), nous avons préféré la réduction considérable du coût à l'intérêt apparemment réduit de la présence de ces contraintes dans les familles et interfaces.

Nous avons donc supprimé les structures "def" et "sel" des "text_contr_famille" et "text_contr_interface". De cette manière on ne réanalyse automatiquement que l'ensemble minimal "Er" de réalisations.

- Souplesse des contraintes : nous avons constaté la valeur de l'assouplissement des contraintes de composition par les deux mécanismes :

  - les sélections préférentielles (par défaut) : nous avons vu au (4.3.2.1) que ce type de contraintes permet de décrire les réalisations à choisir de préférence. Dans notre expérience limitée, nous avons constaté que la sélection préférentielle était souvent utilisée. La construction de réalisations composées par ce type de sélection est toujours réussie. Si dans une famille f, il existe une réalisation qui satisfait ces contraintes alors elle sera choisie, sinon n'importe quelle autre réalisation de f sera choisie (on suppose qu'il existe au moins une réalisation pour f).

  Nous pensons que ce type de contraintes est celui qui sera fréquemment utilisé ; les contraintes obligatoires ne seront utilisées que dans des cas très spéciaux,

  - attributs préférentiels (l'opérateur ":") : l'utilisation de l'opérateur ":" dans l'expression d'une contrainte de sélection permet la recherche d'une réalisation dont on souhaitait qu'elle ait un attribut donné, avec une valeur donnée, sans toutefois l'imposer. Cet attribut ne devient obligatoire que si une réalisation le possède. Ceci peut permettre la sélection de réalisations qui ont le plus d'attributs demandés.

  Ces mécanismes ce sont avérés d'une remarquable utilité.

- libre définition d'un attribut et de sa valeur : nous avons vu que l'expression d'un attribut d'une réalisation consiste en la donnée du nom de l'attribut et de sa valeur. Ces noms et ces valeurs ne sont pas prédéfinis dans le système, mais laissés au gré de l'utilisateur. Certes, ceci permet à chaque utilisateur de se définir lui-même les caractéristiques de ses réalisations, mais ce manque total de contrôle des noms et de leurs valeurs, notion importante dans ADELE, peut conduire à des erreurs difficiles à trouver.

En effet il est très difficile, dans un gros logiciel, quand on désire par exemple choisir une réalisation ayant un attribut "auteur" qui a pour valeur "Pierre", de se rendre compte que l'échec de cette sélection est dû au fait d'avoir écrit "Piere" au lieu de "Pierre".

Ceci nous a permis de constater le manque de mécanismes de contrôle donnant plus de sécurité et assurant le maximum de liberté.

6.3.3.3 Propagation des types via les interfaces

Dans une première version, il était possible d'importer une interface dans une autre. Cette possibilité créait une propagation des types via les interfaces (cf. 3.1.5) et impliquait, en cas de modification d'une interface, de propager les effets de cette modification à toutes les interfaces qui l'importaient directement ou indirectement. Pour éviter cet effet de bord au coût catastrophique, nous avons limité les importations aux seules réalisations.

Au cours de l'expérimentation de la base, nous nous sommes rendus compte que cette restriction était difficilement acceptable pour de gros logiciels. Nous avons alors défini la notion d'environ (cf. 3.2.2.2).

Toutefois la propagation des types via les interfaces est rétablie, et avec elle tous les problèmes qu'elle implique (cf. 3.1.5).

6.3.4 Evolution du système

Nous avons senti les problèmes posés par l'évolution de gros logiciels. Au départ, le système était composé d'un nombre important de modules ; mais au fur et à mesure que nous l'ayons utilisé, nous nous sommes rendus compte de beaucoup de simplifications importantes à faire. Ceci nous a quelquefois conduit à restructurer complètement le système, à supprimer des modules, ou à en modifier d'autres, ... De sorte que la version actuelle contient une dizaine de modules de moins que la version de départ.

L'évolution de notre propre système, nous amène à améliorer les mécanismes du système qui supporteront aisément cette évolution, tels que le déplacement d'un sous arbre de famille d'un niveau à un autre, ...

6.3.5 Efficacité des algorithmes

La quasi totalité des commandes dont en particulier les commandes les plus fréquentes, s'exécute en moins d'une seconde (cpu) y compris la recopie de segments entre la base et l'espace de l'usager.

La construction de configurations (réalisations composées) mettant en oeuvre plusieurs dizaines de familles nécessite de 3 à 10 secondes. Les effets de bord les plus désastreux (heureusement rares) peuvent consommer plusieurs dizaines de secondes.

6.4 Conclusion

Nous notons que la disponibilité de la gestion automatique de configuration a une influence précise sur le processus de la conception et de la construction de logiciels. La possibilité de reconstruire rapidement et efficacement un logiciel, après une modification d'un de ses composants, permet de faire facilement des expériences avec des versions alternées de modules. Ceci peut avoir une influence positive sur le processus de conception. Il encourage les concepteurs à examiner différentes possibilités pour un module. Il est ainsi possible d'isoler les décisions significatives de la conception.

Nous notons également qu'avec un peu d'adaptation (facilités d'expressions des interfaces), ce système peut être utilisé en dehors d'ADELE, et nous pensons (selon notre première expérience) qu'il sera un outil utile à la gestion automatique de programmes à versions multiples.

7 CONCLUSION

Ce chapitre est divisé en deux parties reflétant les centres d'intérêt de cette thèse. Nous commençons par la présentation des apports et perspectives de la base ADELE. Ensuite, à la lumière de l'étude des bases de données dans les Ateliers de Génie Logiciel, et de notre expérience dans ADELE, nous donnons des suggestions dont la prise en compte pourrait aboutir à des SGBD généralisées pouvant supporter des domaines d'applications tels que le génie logiciel. Nous terminons par une une conclusion générale sur la thèse.

7.1 Apports et perspectives de la base ADELE

Dans ce paragraphe qui constitue la conclusion de la base ADELE, nous commencerons par comparer notre travail avec d'autres traveaux effectués dans le même domaine. Nous dégagerons de cette comparaison l'aspect original de cette base, et nous terminerons par ses extensions et ses perspectives.

7.1.1 Apports de la base ADELE

7.1.1.1 Modèle de logiciel

Dans la plupart des ateliers étudiés, le modèle des logiciels est spécifié dans des langages appropriés. Souvent plusieurs types de données sont prédéfinis dans le modèle mais une certaine combinaison de ces types est libre.

Dans ADELE, il n'existe actuellement aucun langage assurant la fonction de constructeur de types. Les entités et relations sont créées par une seule commandes "créer nom_entité ". Les types étant prédéfinis, il est facile au système de les déduire à partir du "nom_entité".

Nous avons cependant utilisé un formalisme clair et concis pour décrire le modèle des logiciels dans ADELE, ce qui manque dans la plupart des ateliers. Ce formalisme a permis d'intégrer la description des données et leurs traitements. Les propriétés des entités sont bien décrites ainsi que les contraintes qui assurent l'intégrité du logiciel en entier. Cet effort peut servir d'exemple pour une étude de la formalisation des modèles logiciels, ce qui est à notre sens vital dans le génie logiciel.

7.1.1.2 Méthode de développement

- Décomposition : comme dans pratiquement tous les ateliers étudiés, la décomposition des logiciels dans ADELE peut être faite selon la méthode descendante. Cependant une extension est en vue pour permettre la combinaison de la décomposition descendante et ascendante (cf. 7.1.2.2). Nous notons que dans ce domaine la base ADELE se distingue nettement par :

  - la visibilité dynamique entre les familles : il est possible (cf. 4.3.1) de modifier la visibilité entre les familles, et la base est remise dans un état cohérent,

  - la restriction dynamique entre les interfaces : le système de gestion de la base permet à des interfaces qui ne sont pas restrictions l'une de l'autre, de le devenir, et inversement il permet à une interface restriction d'une autre, de ne pas l'être (cf. 4.3.1). Dans les deux cas la cohérence de la base est assurée,

  - la répartition dynamique des programmeurs sur les familles : bien que cette notion soit utilisée dans d'autres travaux tels que CADES, elle est plus souple est plus générale dans ADELE (cf. 4.3.1),

- composition : l'apport essentiel et original de la base ADELE est la construction automatique de configurations à partir de composants à versions multiples, et le maintien, à tout instant, de ces configurations dans un état cohérent.

La philosophie des outils construits sur UNIX (FEL 79) a fourni l'inspiration générale pour la reconstruction automatique, et la notion de liste de composition d'une configuration est analogue à celle de "slist" introduite dans (CRI 80). Dans tous les ateliers étudiés, cet aspect est totalement absent, à l'exeption de CEDAR (cf. 2.2.2).

Le point principal et original dans notre approche, consiste en l'expression de la composition d'une configuration, par des contraintes sur les attributs de ses composants plutôt que sur leurs noms. Les attributs sont localement attachés à une réalisation. Le système permet trois possibilités de construction de configuration qui peuvent être combinées :

  - sans qu'aucune contrainte ne soit spécifiée : le système calcule la liste de composition de cette configuration en respectant les contraintes générales de l'atelier. Cette possibilité est utile dans les cas où l'utilisateur n'a pas d'exigence spéciale, ce qui est souvent le cas,

  - en spécifiant des sélections préférentielles : le système calcule la liste de composition de cette configuration en choisissant, de préférence, les réalisations caractérisées par les attributs spécifiés. Cette possibilité est fréquemment utilisée,

- en spécifiant les sélections obligatoires : pour qu'une réalisation puisse être composante de cette configuration, il faut qu'elle ait obligatoirement les attributs spécifiés. Plusieurs conditions font que cette possibilité de construction de configuration se solde par un échec (4.3.2.2), alors que dans les deux premières possibilités le succès de la construction est assuré (en supposant que toute famille utilisée a au moins une réalisation).

Un autre aspect important de notre approche est la cohérence permanente d'une configuration. Les effets d'une modification d'un composant (de ses contraintes d'utilisation ou de son texte) sont automatiquement et directement propagés sur toutes les configurations qui le contiennent. Des tentatives de reconstruction automatique de ces configurations sont effectuées. Ceci permet d'avoir, à tout instant, les configurations dans le meilleur état possible. Ce point n'a pas été abordé dans les travaux connus dans ce domaine (FEL 79, CRI 80, SCH 82, ...) qui ne remettent une configuration dans un état cohérent qu'explicitement en utilisant le mécanisme des estampilles. Ce mécanisme a l'inconvénient de ne pas refléter les effets d'une modifications au moment où elle est faite. De plus il est impossible de savoir l'état exact d'une configuration.

7.1.2 Perspectives

7.1.2.1 Modèle de logiciel

Nous avons vu au (7.1.1.1) que les types des entités sont prédéfinis et qu'aucun mécanisme ne permette de supporter la construction de nouveaux types.

Il est évident que des organisations différentes peuvent avoir des modèles de logiciels différents, ce que nous avons réellement constaté dans les ateliers étudiés.

Il serait donc important, de pouvoir supporter des logiciels aux modèles différents. Un langage de définition de données est un élément essentiel de la généralisation de la base ADELE. L'impact de cette généralisation est très difficile à juger, mais nous pensons qu'il est justifié.

7.1.2.2 Méthode de développement

- Décomposition : actuellement ADELE supporte la décomposition descendante des logiciels. Dans notre expérience (cf. 6.3.4) nous avons parfois pressenti le besoin d'une décomposition ascendante. La combinaison des deux méthodes est intéressante. On pourra alors déplacer, détruire, et insérer un noeud dans l'arbre de la décomposition d'un logiciel, en assurant l'intégrité totale de la base,

- composition : la notion d'attribut est essentielle dans la base ADELE. Un formalisme pour son expression et son contrôle est donc nécessaire (cf. 6.3.2). Nous pensons que les attributs d'une réalisation appartiennent à deux catégories :

  - les attributs pouvant être calculés par le système, tels l'état de la réalisation, le nom de son créateur, la date de sa création, ... Ces informations ne peuvent être manipulées que par le système de gestion de la base,

  - les attributs fixés par l'utilisateur : ces attributs permettent d'attacher des caractéristiques, particulières et définies par l'utilisateur, à une réalisation. L'étude de l'expression formelle de ces attributs, avec des mécanismes de détection d'erreurs, ne pourra qu'être profitable,

- évolution des logiciels : actuellement, comme effet d'une modification d'un source (d'une interface ou d'un corps), toutes les entités potentiellement atteintes par cette modification sont marquées "inconérentes" et leurs codes objets sont marqués "périmés". Cependant (cf. 3.1.5) le pourcentage des entités effectivement atteintes par cette modification est faible. Il est intéressant d'examiner les possibilités de ne rendre "incohérent" et "périmé" que ce qui devrait l'être.

7.1.3 Problèmes ouverts et prolongements

La première expérience nous permet de prendre un peu de recul et d'examiner la généralité de l'expression des contraintes, tout en préservant notre conception principale (expression de la sélection des révisions, des codes objets ayant certaines options, ...).

Nous espérons que cette expérience marquera quelques progrès dans la recherche de moyens convenables pour l'expression de la "programmation globale". En effet au cours de cette thèse, nous avons montré que l'expression de la "programmation globale" est un problème qui intéresse actuellement beaucoup de chercheurs, et que des expériences sont tentées dans plusieurs directions.

L'expression des contraintes sur les composants, introduites dans ADELE, est une de ces directions. Les possibilités qu'elle offre, les moyens nécessaires à son adaptabilité, son efficacité, et son future prologement sont loins d'être connus. Des travaux de recherche et d'évaluation dans ce domaine soulèveront, sans doute, beaucoup d'intérêt.

7.2 SGBD pour génie logiciel : perspectives et propositions

7.2.1 Modèle de données

Des paragraphes précédents, il apparaît clairement que, pour tenir compte des besoins vitaux des domaines d'applications tels que le génie logiciel, une généralisation des SGBD et donc de modèles de données actuels s'impose. Les types d'attributs des entités ne doivent pas se limiter aux types élémentaires, mais doivent s'étendre aux types construits (documents, dessins graphiques, ...) riches sémantiquement. Le projet TIGRE (TIG 83), en cours de développement à l'IMAG, s'oriente dans cette direction.

Des chapitres précédents, nous pouvons déduire une tendance des ateliers logiciels à utiliser le formalisme relationnel (cf. 2.3). Ceci étant, nous constatons aussi qu'il n'est pas très adapté. Les logiciels ont très souvent des structures arborescentes de plusieurs niveaux de profondeur (un exemple simple est celui donné au (3.1.4), qui n'est pas encore totalement réel), le formalisme relationnel n'offre pas cette vue réelle dont aura souvent besoin l'utilisateur.

Les mécanismes d'accès aux informations des SGBD relationnels sont moins adaptés à cette structure. La plupart des ateliers, pour éviter cet inconvénient gardent la vue externe sous forme du modèle relationnel et l'implantent sur des systèmes de type CODASYL (CADES, et ISDOS).

Nous pensons qu'il serait intéressant de tenir compte de ces problèmes dans un SGBD généralisées, et d'examiner dans quelle mesure l'offre de la vue relationnelle, ou arborescente, ou les deux à la fois est justifiée.

Nous constatons aussi la tendance à séparer les textes des entités de la description de ces derniere (cf. 2.3.3). Il est important d'examiner l'apport réel de cette séparation, par rapport à celui d'une intégration. Les points sensibles de cette étude sont, naturellement, le degré de complexité de la gestion des textes, séparés de la description des entités, la cohérence entre les textes et leurs descriptions, et la protection de ces textes contre toute opération autre que celles fournies par le SGBD (cette séparation doit être transparente à l'usager).

Une attention particulière doit être accordée à l'existence de versions multiples d'un attribut et à leur gestion. Nous trouvons dans la littérature des solutions diverses à ce problème (voir par exemple (IVI 77, TIC 82, KAT 83)), mais il faudra examiner dans quelle mesure ces solutions peuvent être utilisées dans le domaine des bases de données.

7.2.2 Intégrité des données

La fréquence, l'importance, et la complexité des contraintes d'intégrité dans le génie logiciel nécessite des études sérieuses pour la définition des moyens, de leur expression et de l'optimisation de leur mise en oeuvre. Nous pouvons citer, à titre d'exemple :

- l'étude des possibilités d'expression des contraintes d'intégrité sémantiques, pour que les SGBD puissent évaluer la porté des effets d'une opération, avant que celle-ci ne soit validée. Ce mécanisme permettra, par exemple, à l'utilisateur de revenir sur sa décision si les dégats sont très importants ou le coût catastrophique,

- l'étude des possibilités d'optimisation : il est important de définir les cas où une opération ne nécessite pas le lancement de tous les "déclencheurs" conçus pour rendre conforme les effets de cette opération aux contraintes imposées. L'effet d'un sous-ensemble de ces "déclencheurs" peut être déjà connu, lors des opérations précédentes (soit, par exemple, que l'opération en cours ne modifie pas cet effet, soit que ce dernier est facile à mettre à jour) et donc être conservé par le SGBD. Il suffira d'exécuter les "déclencheurs" dont les effets sont encore inconnus ou sont modifiés.

7.3 Conclusion

La notion de base de données est très importante. La réalisation de SGBD généralisées sera un grand apport pour l'informatique.

Des travaux de recherche portent sur les divers domaines d'applications que doivent supporter ces SGBD (HUF 81, HAS 82, TIG 83).

Notre étude contribue à cette recherche en examinant un domaine d'application particulier, le génie logiciel, où le manque de tels SGBD est particulièrement sensible.

A1 EXEMPLE COMPLET

Dans cet annexe, nous commençons par donner des exemples du contenu du segment "man" associé au différentes entités de l'exemple (3.1.4), et nous terminerons par des exemples de construction de réalisations composées dans des cas les plus représentatifs.

A1.1 Contenu des segments "man"

A1.1.1 Famille SystèmeGestionBase

```
manuel SystèmeGestionBase

     use
          --informations internes consultables uniquement
          -- état

     nom                          : SystèmeGestionBase.man
     niveau                       : 0

          -- relations

     liste_familles_filles        : ManipStructure ManipAttributs
                                    ModulesService StructDonnées

     liste_familles_externes      :

     liste_familles_liées         :

     liste_interfaces             : prog com

end

manuel SystèmeGestionBase_prog

     incl
          com

          --informations internes consultables uniquement
          --état

     nom                          : SystèmeGestionBase_prog.man
     niveau                       : 0
     vide                         : faux
     modifdate                    :
     modifusager                  :
     environ                      : faux
     incohérent                   :
     verrouillé                   :

          --relations

     liste_incl_inverse           :
     liste_environs_inclus        : StructDonnées
     liste_environs_incluants     :
     liste_dépendance_inverse     :
     liste_corps                  : c1
     liste_réalcomp               : rc1

end
```

```
manuel SystèmeGestionBase_com

    incl

        --informations internes consultables uniquement
        --état

    nom                          : SystèmeGestionBase_com.man
    niveau                       : 0
    vide                         : faux
    modifdate                    :
    modifusager                  :
    environ                      : faux
    incohérent                   :
    verrouillé                   :

        --relations

    liste_incl_inverse           : prog
    liste_environs_inclus        : StructDonnées
    liste_environs_incluants     :
    liste_dépendance_enverse     :
    liste_corps                  :
    liste_réalcomp               : rc2

end

manuel SystèmeGestionBase_prog_c1

    déf

    sel
        ManipAttributs Cohérence_i1 (version = minimale)

    attr
        auteur    =    Pirre
        version   =    minimale
        trace     =    vrai

    util
        vrai

        --informations internes consultables uniquement
        --état

    nom                          : SystèmeGestionBase_prog_c1.man
    niveau                       : 0
    vide                         : faux
    modifdate                    :
    modifusager                  :
    incohérent                   :
    verrouillé                   :

        --relations

    liste_dependance             : ManipStructure_i1 ManipAttributs_i1
    liste_réal_englobantes       : rc2
    liste_versions               : 01 02
    liste_codes_objets           : Multics

end
```

```
manuel SystèmeGestionBase_prog_rc1

     déf
          (trace : vrai)

     sel
          SystèmeGestionBase (version = sipmlifiée)

     attr
          auteur     =     Pirre
          version    =     minimale
          trace      =     vrai

     util
          vrai

          --informations internes consultables uniquement
          --état

     nom                         : SystèmeGestionBase_prog_rc1.man
     niveau                      : 0
     vide                        : faux
     modifdate                   :
     modifusager                 :
     incohérent                  :
     verrouillé                  :
     incomplet                   : SystèmeGestionBase_prog

          --relations

     liste_dépendance            :
     liste_réal_englobantes      :
     liste_versions              :
     liste_codes_objets          :

end
```

```
manuel SystèmeGestionBase_com_rc2

    déf
        ManipAttributs_11_R

    sel

    attr
        auteur      =     Jaques

    util
        vrai

        --informations internes consultables uniquement
        --état

    nom                         : SystèmeGestionBase_prog_rc2.man
    niveau                      : 0
    vide                        : faux
    modifdaté                   :
    modifusager                 :
    incohérent                  :
    incomplet                   :
    vérrouillé                  :

        --relations

    liste_dépendance            :
    liste_réal_englobantes      :
    liste_versions              :
    liste_codes_objets          :

end
```

```
text réalcomp  SystèmeGestionBase_prog_rc1

     déf

     sel

          --informations internes consultables uniquement

     list_comp :

end

text réalcomp  SystèmeGestionBase_com_rc2

     déf

     sel
            (auteur : Pierre )
          ManipStructure_i1

          --informations internes consultables uniquement

     list_comp : prog_c1 Manipstructure_i1_c1 ManipAttributs_i1_R
                 DétectCycle_i1_c1 StructPhysique_i1_c1
                 ModulesService_i1_RMS

end
```

A1.1.2 Famille MaipStructure

```
manuel SystèmeGestionBase>ManipStructure

     use
          --informations internes consultables uniquement
          -- état

     nom                        : SystèmeGestionBase>ManipStructure.man
     niveau                     : 1

          -- relations

     listes_familles_filles     : DétectCycle StructPhysique

     liste_familles_externes    : DetectCycle StructDonnées

     liste_familles_liées       :

     liste_interfaces           : i1

end
```

--Nous ne décrivons pas les manuels des entités suivantes : ManipStructure_i1, ManipStruture_i1_c1, DétectCycle, DétectCycle_i1, DétectCycle_i1_c1, StructPhysique, StructPhysique_i1, et StructPhsique_i1_c1, parcequ'elles n'ont rien de particulier.

A1.1.3 Famille ManipAttributs

```
manuel SystèmeGestionBase>ManipAttributs

     use
          --informations internes consultables uniquement
          -- état

     nom                          : SystèmeGestionBase>ManipAttributs.man
     niveau                       : 1

          -- relations

     listes_familles_filles       : Compilation Cohérence
                                    PropageEffets

     liste_familles_externes      : StructDonnées

     liste_familles_liées         :

     liste_interfaces             : i1

end

manuel SystèmeGestionBase>ManipAttributs_i1

     incl

          --informations internes consultables uniquement
          --état

     nom                          :
               SystèmeGestionBase>ManipAttributS_i1.man
     niveau                       : 1
     vide                         : faux
     modifdate                    :
     modifusager                  :
     environ                      : faux
     incohérent                        :
     verrouillé                   :

          --relations

     liste_incl_inverse           :
     liste_environs_inclus        : StructDonnées
     liste_environs_incluants     :
     liste_dépendance_inverse     : SystèmeGestionBase_prog_c1
     liste_corps                  : c1
     liste_réalcomp               : R

end
```

```
manuel SystemeGestionBase>ManipAttributs_i1_c1

     déf
          (version : minimale)

     sel
          SystèmeGestionBase>ModulesService (trace = vrai)
          Compilation (trace = faux)
     attr
          version    =    minimale
          trace      =    faux

     util
          vrai

          --informations internes consultables uniquement
          --état

     nom                         :
               SystèmeGestionBase>ManipuleAttributs_i1_c1.man
     niveau                      : 1
     vide                        : faux
     modifdate                   :
     modifusager                 :
     inconérent                  :
     verrouillé                  :

          --relations

     liste_dépendance            : Compilation_i1 Cohérence_i1
                                   PropageEffets_i1
     liste_réal_englobantes      : SystèmeGestionBase_prog_rc2
                                   SystèmeGestionBase>ManipAttributs_i1_R

     liste_versions              : 01 02
     liste_codes_objets          : Multics

end
```

```
manuel SystèmeGestionBase>ManipAttributs_i1_R

     déf

     sel

     attr
          version = minimale

     util
          vrai

          --informations internes consultables uniquement
          --état

     nom                         :
                SystèmeGestionBase>ManipAttributs_i1_R.man
     niveau                      : 1
     vide                        : faux
     modifdate                   :
     modifusager                 :
     incohérent                  :
     incomplet                   :
     vérruillé                   :

          --relations

     liste_dépendance            : ModulesService_i1
     list_contr_héritées         : ModulesService_i1 (trace = vrai)
     liste_réal_englobantes      : rc2
     liste_versions              : 01
     liste_codes_objets          :

end
```

```
text realcomp  SystemeGestionBase>ManipAttributs_i1_R

     déf

     sel
          OR (ManipAttributs_i1_c1 , ManipAttributs (auteur = Pierre)

          --informations internes consultables uniquement

     list_comp : ManipAttributs_i1_c1 Compil_i1_c1 Cohérence_i1_c1
                 PropageEffets_i1_c1

end
```

A1.1.4 Famille ModulesService

```
manuel SystèmeGestionBase>ModulesService

     use


          --informations internes consultables uniquement
          -- état

     nom                          : SystèmeGestionBase>ModulesService.man
     niveau                       : 3

          -- relations

     listes_familles_filles       : ComparListes AnalyseNoms
     liste_familles_externes      : StructDonnées
     liste_familles_liées         : Cohérence
     liste_interfaces             : i1

end

manuel SystèmeGestionBase>ModulesService_i1

     incl


          --informations internes consultables uniquement
          --état

     nom                          :
               SystèmeGestionBase>ModulesService_i1.man
     niveau                       : 3
     vide                         : faux
     modifdate                    :
     modifusager                  :
     environ                      : faux
     incohérent                        :
     verrouillé                   :

          --relations

     liste_incl_inverse           :
     liste_environs_inclus        : StructDonnées
     liste_environs_incluants     :
     liste_dépendance_inverse     : SystèmeGestionBase_prog_c1
                                    PropageEffets_i1_c1
     liste_corps                  :
     liste_réalcomp               : RMS

end
```

```
manuel SystèmeGestionBase>ModuleService_i1_RMS

     déf

     sel

     attr
          version = minimale
          trace = vrai

     util
          vrai

          --informations internes consultables uniquement
          --état

     nom                        :
               SystèmeGestionBase>ModulesService_i1_RMS.man
     niveau                     : 3
     vide                       : faux
     modifdate                  :
     modifusager                :
     incohérent                 :
     incomplet                  :
     verrouillé                 :

          --relations

     liste_dépendance           :
     liste_réal_englobantes     : rc2
     liste_versions             : 01
     liste_codes_objets         :

end

text réalcomp  SystèmeGestionBase>ModulesService_i1_RMS

     déf

     sel

          --informations internes consultables uniquement

     list_comp : Comparlistes_i1_c1 AnalyseNoms_i1_c1

end
```

A1.2 Explication de mécanismes

A1.2.1 Construction de la réalisation composée ManipAttributs_il_R

En suivant la méthode décrite au (4.3.2.2) nous aurons :

1) le choix du corps principal de R : deux contraintes alternantes existent pour le choix de ce corps, celles imposées dans le texte de R, le choix de ManipAttributs_il_cl, ou d'un corps ayant l'attribut "auteur" = "Pierre". La première contrainte est satisfaite par le corps cl,

2) choix d'autres composants :

- deuxième niveau : une contrainte commune est "version : minimale", imposée par cl, nous oblige à choisir de préférence une réalisation ayant cet attribut avec cette valeur chaque fois que c'est possible.

* Compilation_il : en plus de la contrainte précédente, une autre contrainte (obligatoire) est imposée (trace = faux) pour le choix d'une réalisation pour compilation_il. Nous supposons que compilation_il a un corps cl qui vérifie seulement la dernière contrainte. Ce corps sera choisi parceque la contrainte commune est préférentielle,

* Cohérence_il : aucune autre contrainte n'est imposée sur le choix d'une réalisation pour Cohérence_il. Si on suppose que Cohérence_il a un corps cl, que celuici vérifie la contrainte commune ou pas il sera choisi,

* PropageEffets_il : comme pour Cohérence_il, on suppose que le corps cl est choisi,

- troisième niveau : nous supposons que le corps Cohérence_il_cl dépend de ModulesService_il. ManipAttributs_il_cl impose que la réalisation à choisir doit vérifier (trace = vrai). Or on constate que ModulesService_il est partagée entre Cohérence_il_cl et SystèmeGestionBase_il_proq_cl (voir les dépendances inverses de cette interface), et que SystèmeGestionBase n'est pas descendant de la famille Cohérence. Le choix d'une réalisation pour ModulesService_il doit donc être attardé. R dépend alors de cette interface et hérite le contraite (trace = vrai) (voir le manuel de R). Nous trouvons la liste de composition de R dans text_réalcomp de R.

A1.2.2 Construction de la réalisation SystèmeGestionBase proq rcl

De la même manière que R, le résultat de la construction de rcl est porté dans son "man" et "text". Nous constatons qu'elle est incomplète à cause de son interface, qui n'a pas de réalisation ayant l'attribut (version = simplifiée).

A1.2.3 Construction de la réalisation SystèmeGestionBase com rc2

Le résultat de la construction de rc2 est porté dans son "man" et "text". Nous constatons que cette réalisation a choisi R qui dépendait de ModulesService_il. Dans rc2 le choix d'une réalisation peut se faire pour cette interface et la réalisation RMS a été choisie.

A2 ALGORITHMES DE SELECTION ET DE COHERENCE

Dans ce qui suit nous ne donnons que les grandes lignes des algorithmes importants qui réalisent l'approache ADELE pour la construction de configurations et le maintien de leur cohérence.

A2.1.2 desription de la procédure de vérification

Cet algorithme est utilisé aussi bien pour la vérification de la cohérence des contraintes décrites dans les manuels des entités que pour la construction de réalisations composées. En effet la création d'une réalisation composée n'est autre que la vérification de la cohérence de l'ensemble des contraintes d'utilisation de ses composants.

```
procedure cohérence (texte)

début

    (* texte est un document qui contient la spécification des    *)
    (* contraintes de l'entité à partir de laquelle l'analyse     *)
    (* commencera. Pour plus de clarté nous supposons que         *)
    (* cette entité est une réalisation composée RC               *)

    (* initialisation de la racine du graphe de dépendance de RC  *)

    {listinterfaceimportée} := interface de RC

    (* boucle principale de l'algorithme                          *)

    tant que ({listinterfaceimportée} ≠ ∅ ) faire

    début

        (* parmi les interfaces de la listeinterfaceimportée on   *)
        (* choisira celle qui a le niveau de profondeur minimal,  *)
        (* donc la plus haute dans le graphe                      *)

        soit une interface I∈{listeinterfaceimportée/ le niveau min}

        (* récupération de toutes les contraintes faites sur la   *)
        (* famille "F" de I                                       *)

        CONTRAINTES (F)      (* CONTRAINTES est une procédure     *)

        (* on teste si les attributs des contraintes imposées     *)
        (* n'entrent pas en conflits. Si c'est le cas utiliser le*)
        (* mécanisme des contraintes alternantes pour le          *)
        (* résoudre si c'est possible                             *)

        si CONFLIT (attributs) alors
        début

            avertir l'utilisateur que RC est "incohérente",
            propager les effets de cette incohérence,

            (* les réalisations qui l'englobent deviennent       *)
            (* incohérentes, et récursivement toute réalisation  *)
            (* englobant une réalisation incohérente devient     *)
            (* incomplète                                        *)

        fin
```

```
            (* tester s'il existe un conflit sur le choix des          *)
            (* interfaces. Si oui essayer de le résoudre par le        *)
            (* mécanisme des interfaces restreintes. Si pas de         *)
            (* conflit choisir une réalisation pour l'interface I      *)

            si CONFLIT (interface) alors
            début

                 avertir l'usager du conflit,
                 propager les effets de cette incohérence,

            fin

            (* pas de conflit, une réalisation R est choisie           *)

            si RC dépend de I (* cf. 4.3.2.2 *) alors
                 RC hérite les contraintes imposées sur F
            sinon
            début

                 {listeinterfaceimportée} :=
                      {listeinterfaceimportée} U {liste de dépendance de R},
                 récupération des contraintes d'utilisation de R,
                 la liste de composition de RC contiendra R,
            fin

       fin (* boucle principale *)

  fin (* cohérence *)
```

A2.2 Sélection de réalisation pour une interface

On donne ici les grandes lignes de l'algorithme qui sélectionne, selon les attributs exigés, une réalisation pour une interface donnée.

```
procedure selection (I)

début

    (* I est le nom d'une interface pour laquelle on cherche une  *)
    (* réalisation.                                               *)

    si (existe une sélection nominative d'une réalisation R) alors
    début

        si (R est non utilisable ou ne satisfait pas les attributs
              exigés) alors "erreur"

    fin

    sinon

    début


        (* récuperer les contraintes de selection par défaut      *)
        (* portant sur cette famille.                             *)

        si (parmi ces contraintes par défaut existe une sélection
              nominative d'une réalisation R) alors
           R doit satisfaire les attributs exigés pour être choisie,
        sinon

        début

            ajouter les attributs par défaut à ceux exigés

            (* on cherche une réalisation qui satisfait           *)
            (* l'ensemble des attributs.                          *)

            si (existe une réalisation qui satisfait l'ensemble des
                  attributs) alors la choisir
            sinon chercher une réalisation qui satisfait seulement
                  les attributs exigés
            fin

        fin

        (* on a terminé avec cette interface. si on a pas pu      *)
        (* choisir de réalisation pour elle, on cherche une pour  *)
        (* l'une des interfaces avec lesquelles elle en relation  *)
        (* de restriction.                                        *)

        si (le choix non effectué) alors
             SELECTION (une interface restreinte de I)
        fin

fin (* selection *)
```

A3 QUELQUES COMMANDES D'UTILISATION DE LA BASE

Nous donons ici, la description simplifiée de quelques commandes de manipulation de la base.

lire (nombase) (fichloc)

Fonction : lecture d'un texte (source d'un programme, contraintes d'utilisation d'une entité, document, ...). Ce texte sera transférer de la base, à partir de l'attribut désigné par "nombase", dans un segment "fichloc" de l'espace de travail de l'utilisateur. L'usager se trouvera sous l'éditeur de texte "ted" de MULTICS.

Paramètres : les deux paramètres sont en entrée. "nombase" désigne l'attribut à lire, "fichloc" désigne le fichier local, dans l'espace de travail courant MULTICS de l'usager, qui recevra le résultat de la lecture. Si "nombase" est omis le système prendra le dernier attribut réservé. Si "fichloc" est omis le système créera un fichier ayant le même nom que l'attribut lu.

reserve nombase

Fonction : reservation de l'entité désignée par "nombase" ainsi que toutes les entités qui peuvent devenir incohérentes par la modification de l'entité réservée. Par certaines option cette commande permet de faire la réservation de l'entité désignée, et la lecture de son text.

catal (nombase) (fichloc) (fdep) (cohérent)

Fonction : catalogue un texte contenu dans un fichier local "fichloc", dans un attribut, d'une entité de la base, désigné par "nombase". Par certaines options on peut conserver la révision précédente, détruire le "fichloc" après le catalogage, ...

Paramètres : "fdep" est un fichier contenant la liste des dépendances du texte catalogué si ce dernier est celeui d'un corps. "cohérent" indique si ce texte est correcte.

créer nombase ("réalcomp")

Fonction : créer l'entité désignée par "nombase". le type de l'entité a créer est déduit par le sytème sans aucune ambiguïté, sauf pour les réalisations composées où on doit terminer le commande par la chaine "réalcomp". Les attributs de l'entité créée sont également crées et initialisés.

détruire nombase

Fonction : destruction de l'entité désignée par "nombase".

visible (nomfamille) (nomlien)

Fonction : création d'un lien "nomlien" sur une famille externe "nomfamille".

Paramètres : "nomfamille" est le nom de la famille sur laquelle sera fait le le lien. Cette famille sera une famille externe à la famille définissant l'espace de travail courant de l'utilisateur activant cette cammande. Par défaut c'est le nom de la dernière famille créée.
"nomlien" est le nom du lien par lequelle cette famille externe sera connue dans cet espace de travail. Par défaut c'est le nom de la famille sur laquelle est effectué le lien.

delesp (nom_local_famille)(nomespace)(idperson)

Fonction : affectation de l'usager "idperson" à l'espace de travail défini par la famille "nom_local_famille". Cet espace sera connu à l'usager par un identificateur "nomespace".

Paramètres : nom_local_famille : ce nom ne doit pas contenir des liens. s'il est omis c'est le nom de l'espace courant de l'utilisateur. "idperson" est le nom de l'usager affecté à cet espace, par défaut c'est l'appelant.

cesp (nomespace)

Fonction : chargement de l'espace de travail "nomesp" de l'utilisateur. Si "nomespace" est omis l'espace par défaut de l'appelant sera chargé.

BIBLIOGRAPHIE

ABR 74 J.R Abrial
Data semantics,
IFIP Conference on Data Base Management,
France, Cargèse (Corse), North-Hollande, 1974

ADA 79 Preliminary ADA reference manual,
SIGPLAN notice, vol. 4, no. 6, June 1979

AZZ 80 M. Azzouz, et al.
Le projet SSP : Système support de programmation,
BIGRE, no 21-22, Rennes 1980

BAY 81 M. Bayer, et al.
Software development in the CDL2 LABORATORY,
in HUN 81

BOG 83 G. Bogo, H. Richy, I. Vatton
Un modèle de représentation de documents généralisés,
Communication aux Journées Francophones de Rennes, Mai 1983

BOU 80 JL. Bouchenez, et al.
Le système LEGOS : environnement de programmation sur MITRA 125,
BIGRE, no 21-22, Rennes, Dec. 1980

BRI 81 J. Briat, et al.
ADELE : un Atelier de DEveloppement de LogiciEl,
Congrès AFCET, Paris, Nov 1981

CAS 81 P.M Cashin, M.L. Joliat, R.F. Kamel, D.M. Lasker
Experience with a modular typed langage : PROTEL,
Proc. 5th International Conf. on Soft. Eng.
pp 136-143, San Diego, March 1981

CHT 81 T.E Cheatham, JR
Comparing programming support environments
in HUN 81

CHE 76 P.S. Chen
The entity-relationship model, toward a unified view of data,
ACM Trans. on Database Systems,
vol 1, no 1, pp 9-36, March 1976

CHV 80 J.L. Cheval, J. Mossière
Evaluation d'un système d'aide à la production de logiciels modulaires : SESAME,
BIGRE no 21-22, Rennes, Dec. 1980

CHV 82 J.L Cheval, J. Estublier, S. Ghoul, S. Krakowiak
Modularité et composition de programmes dans l'atelier de logiciel ADELE,
AFCET, 1er colloque génie logiciel, Paris, Juin 82

CHV 83 J.L. Cheval, J. Estublier, S. Ghoul
Un système automatique de gestion de versions : la base de programmes ADELE,
BIGRE no 36, Cap d'Agde, Oct. 83

COD 70 E.F. Codd
A relational model of data for large shared data banks,
CACM, vol 13, no 6, pp 377-387, 1970

CRI 80 E. Cristofor, T.A Wendt, B.C Wonsiewicz,
Source Control + Tools = Stable Systems,
Proc. COMPSAC 80 (IEEE Computer Soc. Press) Oct. 80

DEL 82 J.C. Delobel, M. Adiba
Bases de données et systèmes relationnels,
Dunod-Informatique 1982

DER 75 F. De Remer, H. Kron
Programming in the large versus programming in the small,
International conference on reliable software,
SIGPLAN, vol 10, no 6, June 1975

EST 83a J. Estublier, S. Krakowiak, J. Mossière, Y. Rouzaud
Design principles of the ADELE programming environment,
International computer symposium (ACM), Nuremberg, March 1983

EST 83b J. Estublier
Pascal ADELE,
Note interne, IMAG, 1983

EST 83c J. Estublier, S. Krakowiak
Gestion de composants logiciels et composition de systèmes,
annexe au rapport final du contrat ADI 80/225,
Avril 1983

FEL 79 S.I. Feldman
Make : a program for maintaining software,
Software practice and experience, vol 9, pp 255-265, 1979

FER 83 L. Ferrat
Expression et contrôle de l'intégrité sémantique dans les bases de données relationnelles,
thèse 3e cycle, U.S.M. de Grenoble, Mai 1983

FOI 80a J. Foisseau, et al.
Recherches sur la conception de programmes assistée par ordinateur,
CERT, rapport no 3/3155/DERI, CPAO/81/3, 1980

FOI 80b J. Foisseau, et al.
Program development with or without coding,
Proc. of IFIP, Oct. 1980

FOI 82a J. Foisseau, et al
Recherches sur la conception de programmes assistée par ordinateur,
CERT, rapport final no 3607/DERI, CPAO/82/1, 1982

FOI 82b J. Foisseau
Assistance à la spécification des fonctions et des types de données dans SPRAC,
BIGRE no 28-29, Grenoble, Jan. 1982

GIC 79 Y. Gicquel, et al.
Problèmes de gestion de logiciel dans le domaine des télécommunications. Proposition d'un système de gestion de logiciel,
Note technique, journées génie logiciel, Pont-a-Mousson, Fev. 1979

GIC 81 Y. Gicquel, et al.
Langage ATOME, langage d'accès aux tableaux et objets en mémoire étendue,
Note technique NT/LAA/SLC/52, CNET, Lanion, Juin 1981

GHO 81 S. Ghoul
Modularité et Pascal,
Rapport de DEA, IMAG, Juin 1981

GHO 82 S. Ghoul
Possibilités d'implantation de la base ADELE,
Note interne, IMAG, 1982

HAB 82 AN. Habermann, D.S. Notkin
The GANDALF software development environment,
The second compendium of GANDALF documentation, CMU 1982

HAS 82 R.L Haskin, R.A Lorie
On extending the functions of a relational database system,
ACM SIGMOD Conference, Orlando, FL, Jun 1980

HAU 81a HL. Hausen, M. Müllerburg, WE. Riddle
Software engineering environments, a bibliography
in HUN 81

HAU 81b HL. Hausen, M. Müllerburg
Conspectus of software engineering environments,
Proc. 5th International Conference on Soft. Eng.
San Diego, California, March 1981

HER 82 M. Herrmann,
Le poste de travail du projet ADELE,
RR, no 299, IMAG, Avril 1982

HOR 79 TR. Horsley, WC. Lynch
PILOT : a software engineering case study,
Proc. 4th International Conf. on Soft. Eng.
Munich, Sep. 1979

HUF 81 KE. Huff
A database model for effective configuration management in the programming environment,
Proc. 5th International Conf. on Soft. Eng.,
San Diego, California, March 1981

HUN 81 H. Hünke (editor)
Software engineering environments,
Proc. of a symposium (S2E2), Lahnstein, (RFA),
North-Holland, 1981

HOW 82 WE. Howden
Contemporary software development environments,
CACM, vol 25, no 5, May 1982

IVI 77 EL. Ivie
The programmer's workbench-a machine for software development,
CACM, vol 20, no 10, Oct. 1977

JAC 78 P. Jacquet,
Les types génériques, proposition pour un mécanisme d'abstraction dans les langages de programmation,
thèse 3e cycle, U.S.M. de Grenoble, 1978

JOL 82 V. Joloboff
UBIK : Un formalisme de modélisation,
Séminaire Base de Données, organisé par l'ADI,
Toulouse, Nov. 1982

KAI 82a GE. Kaiser, et al.
GANDALF environment user's manual and tutorial,
The second compendium of GANDALF documentation,
CMU 1982

KAI 82b GE. Kaiser, AN. Habermann
A description of the current version control supported by the GANDALF environment,
The second compendium of GANDALF documentation,
CMU 1982

KRA 76 S. Krakowiak, M. Lucas, J. Montuelle, J. Mossière
A modular approach to the structured design of operating systems,
Proc. MRI Symposium on Computer Soft. Eng.
Polytechnic Institute, New-Work, 1976

KRA 81 S. Krakowiak
Structural issues in the design of large systems,
INFOTECH State of the Art Report on System Design,
INFOTECH, ltd, 1981

KRA 82 S. Krakowiak
Systèmes intégrés de production de logiciels : concepts et réalisations,
TSI, vol 1, no 3, pp 187-200, 1982

LAM 83 AE. Lamsweerde
Proposal for an ESPRIT pilot project, II,
Rapport technique, institut d'informatique, Namur, Jan. 1983

LAP 81 C. Lapoujade
SSP : la base de données,
Thèse 3e cycle, Université Paul-Sabatier, Toulouse, Nov. 1981

LAU 79 HC. Lauer, EH. Satterthwaite
The impact of MESA on system design,
Proc. 4th International Conf. on Soft. Eng., Munich, Sep. 1979

LOP 83 M. Lopez, J. Palazzo Oliveira, F. Valez
The TIGRE data model,
RR. TIGRE no 2, IMAG, Nov. 1983

LUC 77 M. Lucas
Conception modulaire des systèmes d'exploitation, présentation du projet SESAME,
thèse 3e cycle, USM de Grenoble, Juin 1977

McG 79 RW. Mc Guffin, AE. Ellitson, BR. Tranter, DN. Westmacott
CADES : software engineering in practice,
Proc. 4th International Conf. on Soft. Eng., Munich, Sep. 1979

MIC 79 JG. Mitchell, W. Maybury, R. Sweet
MESA langage manual,
Technical report CSL-79-3, XEROX PARC, April 1979

MIT 81 RW. Mitze
The UNIX system as a software engineering environment,
in HUN 81

MOI 81 S. Moisan
SSP : l'analyseur de projets,
Thèse docteur ingénieur, Université Paul-Sabatier, Toulouse, Nov. 1981

MOS 82 J. Mossière, et al
Représentation interne et manipulation de programmes dans l'atelier de logiciel ADELE,
RR no 299, IMAG, Avril 1982

MRD 81 MRDS : Multics Relational Data Store,
CII HB, 68 A2 AW53, rev 3, 1981

MUL 79 MULTICS programmers' manual reference guide
CII HB, 68 A2 AG91, Nov. 1979

MUR 81 N. Murakami, I. Miyanari, K. Yabuta
SDEM/SDSS : overall approach to improvement of the software development environment,
in HUN 81

NAK 78 Y. Nakamura, R. Miyahara, H. Takeuchi
Complementary approach to the effective software development environment,
IEEE COMPSAC, pp 235-240, Chicago, III, 1978

OST 81 L. Osterweil
Software environment research --the next five years,
in SIG 81

PAR 76 DL. Parnas
On the design and development of program families,
IEEE Trans. on Soft. Eng. vol SE-2, no 1, march 1976

PRA 81 G.D Pratten
Coherence considered as the prime requirement in systems design,
INFOTECH, State of the Art, Report on System Design,
INFOTECH, ltd, 1981

ROU 77 B. Rougeot
Système de gestion de programmes, spécification de définitions,
Note technique no NT/RC1/SIC/15.ED.2, CNET, Lanion 1977

ROU 78 B. Rougeot
Langage de gestion de programmes,
Note technique no NT/RC1/EGN//7, CNET, Lanion 1978

SAL 80 P. Salembier
Comparaison des approches prises dans le système de gestion de logiciel GALAAD et l'environnement de programmation APSE,
BIGRE no 21-22, Rennes, Dec. 1980

SAN 83 M. Santana
Un système de production automatique de générateurs de code,
Thèse 3e cycle, U.S.M de Grenoble, Déc. 1983

SCH 82 EE. Schmidt
Controlling large software development in a distributed environment,
Thesis PHD, University of California, Dec. 1982, Berkly

SIG 81 SIGSOFT, NBS Workshop Repport on Programming Environments,
ACM Soft. Eng. Notes, vol 6, no 4, August 1981

SNO 81 RA. Snowdon
CADES and software system developpement,
in HUN 81

STN 80 J. Buxton (editor)
Stoneman : requirements for ADA programming environment,
US departement of defense 1980

TEI 77 D. Teichroew, EA. Hershey
PSL/PSA : a computer-aided technique for structured documentation and analysis of informations processing systems,
IEEE Trans. on Soft. Eng. vol SE-3, no 1, pp 41-48, January 1977

THA 81 RH. Thayer, AB. Pyster, RC. Wood
Major issues in software engineering project management,
IEEE Trans. on Soft. Eng. vol SE-7, no 4, July 1981

TIC 79 WF. Tichy
Software development control based on module interconnection,
Proc. 4th International Conf. on Soft. Eng. Munich Sep. 1979

TIC 82 WF. Tichy
Design, implementation, and evaluation of a revision control system,
Proc. 6th International Conf. on Soft. Eng. Japan 1982

TIG 83 Présentation générale du projet TIGRE,
RR. TIGRE no 1, IMAG, Jan. 1983

WAS 81 A.I Wasserman
towards integrated software development environments,
INFOTECH, State of the Art, Report on System Design,
INFOTECH, ltd, 1981

WIL 81 RR. Willis
AIDES : computer aided design of software systems-II
in HUN 81

WOR 81 DB. Wortman, JR. Cordy
Early experiences with EUCLID,
Proc. 5th International Conf. on Soft. Eng. pp 27-32, San Diego, March 1981

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Messieurs

. J. MOSSIERE, Professeur

. R. JACQUART, Ingénieur

Monsieur GHOUL Saïd

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 5 décembre 1983

Le Président de l'INP-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,