Les formulaires complexes dans les bases de données multimédia

Christine Collet

THESE

présentée par

Christine COLLET

pour obtenir le titre de **DOCTEUR**

de **L'UNIVERSITE SCIENTIFIQUE TECHNOLOGIQUE ET MEDICALE DE GRENOBLE**

(arrêté ministériel du 5 juillet 1984)

Spécialité : *INFORMATIQUE*

LES FORMULAIRES COMPLEXES

DANS LES BASES DE DONNEES MULTIMEDIA

These soutenue le : *23 Novembre 1987*

Composition du jury :

President	*P.C. Scholl*
Examinateurs	*S. Abiteboul*
	M. Adiba
	F. Bodart
	C. Delobel
	M. Lopez

Thèse préparée au sein du Laboratoire de Génie Informatique
à l'Université Scientifique, Technologique et Médicale de Grenoble

UNIVERSITE SCIENTIFIQUE TECHNOLOGIQUE ET MEDICALE DE GRENOBLE

Président de l'Université :
M. TANCHE

Année Universitaire 1986 - 1987

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ARNAUD Paul	Chimie Organique
ARVIEU ROBERT	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique Expérimentale CNRS
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Boichimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BELORISKY Elie	Physique C.E.N.G- D.R.F.
BENZAKEN Claude	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CAUQUIS Georges	Chimie Organique
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DELOBEL Claude	Mathématiques Appliquées
DEPORTES Charles	Chimie Minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Rolland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
KUPKA Yvon	Mathématiques Pures
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
DE LEIRIS Joel	Biologie

LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
MACHE Régis	Physiologie Végétale
MAYNARD Roger	Physique du Solide
MICHEL Robert	Minéralogie et Pétrographie (Géologie)
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RASSAT André	Chimie Systématique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAKAROVITCH Michel	Mathématiques Appliquées
SAXOD Raimard	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BLANCHI J.Pierre	STAPS
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHARDON Michel	Géographie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des Fluides
DERRIEN Jacques	Physique
DUFREYNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques Appliquées

HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LEBRETON Alain	Mathématiques Appliquées
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
LUNA Domingo	Mathématiques Pures
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RIEDIMANN Christine	Mathématiques Pures
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SOUTIF Jeanne	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ere Classe

BUISSON Roger	Physique IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1

PROFESSEURS de 2 ème classe

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
NOUGARET Marcel	Automatique IUT 1
PEFFEN René	Métallurgie IUT 1
PERARD Jacques	EEA. IUT 1
PERRAUD Robert	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie IUT 1

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
BUTEL Jean	Chirurgie Générale et Digestive	C.H.R.G.
CHAMPETIER Jean	Orthopédie-Traumatologie	C.H.R.G.
CHARACHON Robert	Anatomie-Topographique et Appliquée	C.H.R.G.
COUDERC Pierre	O.R.L.	C.H.R.G.
DELORMAS Pierre	Anatomie-Pathologique	C.H.R.G.
DENIS Bernard	Pneumophtisiologique	C.H.R.G.
GAVEND Michel	Cardiologie	C.H.R.G.
HOLLARD Daniel	Pharmacologie	Faculté La Merci
LATREILLE René	Hématologie	C.H.R.G.
LE NOC Pierre	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
MALINAS Yves	Bactériologie-Virologie	C.H.R.G.
MALLION Jean-Michel	Gynécologie et Qbstétrique	C.H.R.G.
MICOUD Max	Médecine du Travail	C.H.R.G.
MOURIQUAND Claude	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
PARAMELLE Bernard	Histologie	Faculté La Merci
PERRET Jean	Pneumologie	C.H.R.G.
RACHAIL Michel	Neurologie	C.H.R.G.
DE ROUGEMONT Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
SARRAZIN Roger	Neurochirurgie	C.H.R.G.
STIEGLITZ Paul	Clinique Chirurgicale	C.H.R.G.
TANCHE Maurice	Anestésiologie	C.H.R.G.
VERAIN André	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biophysique	Faculté La Merci
	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital Sud
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci

DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie	C.H.R.G.
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

MEMBRES DU CORPS ENSEIGNANT PHARMACIE

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

Je tiens à remercier

Monsieur Pierre-Claude Scholl, Professeur à l'Université Scientifique, Technologique et Médicale de Grenoble, d'avoir accepté de présider le jury de cette thèse. Je lui suis reconnaissante de m'avoir donné l'opportunité de découvrir les joies de l'enseignement.

Monsieur Claude Delobel, Professeur à l'université Paris Sud qui a été le premier à me parler des Bases de Données. Je lui suis reconnaissante aujourd'hui de me faire l'honneur de participer à ce jury.

Monsieur François Bodart, Professeur aux Facultés Universitaires Notre-Dame de la Paix de Namur (Belgique) d'avoir bien voulu juger mon travail et participer à ce jury.

Monsieur Serge Abiteboul, Directeur de Recherche à L'INRIA Paris d'avoir accepter de participer à ce jury. L'interêt qu'il a porté à mon travail, ses conseils et ses questions ont fait que cette thèse existe.

Monsieur Michel Adiba, Professeur à l'Université Scientifique, Technologique et Médicale de Grenoble, qui est à l'origine et Directeur de cette thèse, pour m'avoir accueilli au sein de son équipe de recherche. Sa totale disponibilité à mon égard, sa grande expérience, ses nombreux conseils et encouragements représentent sans nul doute un facteur décisif dans l'aboutissement de ce travail.

Monsieur Mauricio Lopez, Ingénieur du Centre de Recherche BULL, pour m'avoir acceptée en DEA et avoir guider mes premiers pas dans la recherche.

Messieurs Michel Adiba et Mauricio Lopez sont à l'origine de cette thèse. Ils ont su me faire profiter de leurs expériences respectives et me transmettre un peu de leur savoir-faire. Je les en remercie vivement.

Les membres du service reprographie pour le soin et l'efficacité qu'ils ont apporté au tirage de cette thèse.

J'aurais beaucoup à dire également sur mes collègues du projet TIGRE et de l'équipe Bases de Données pour l'amitié qu'il m'ont toujours témoigné : Espéranza pour son appui moral, Fernando pour sa vivacité, Ngoc et Francisca pour l'interêt qu'ils ont porté à ce travail et les nombreuses discussions qui en ont découlé, Toan et Dominique pour leurs encouragements, Marie-Christine pour sa totale disponibilité et sa bonne humeur.

Enfin, je ne saurais oublier tous mes amis qui ont écouté mes complaintes, mes joies, mes peurs depuis le début de cette galère. Que Fabienne, Renée, Pierrette, Simone, Gillou, Jef, Michel, Pierre, Philippe, Renato ... me pardonnent mes excès.

Je tiens tout particulièrement à remercier Jean, pour son soutien moral, sa patience et ses chaleureux encouragements dans l'accouchement de ces quelques pages.

Il me faut également remercier le Centre de recherche Bull de m'avoir donner les conditions matérielles à la bonne réalisation de cette thèse.

A Paul et Virginie
au tenon et à la mortaise
à la chèvre et au chou
à la paille et à la poutre
au-dessus et au-dessous du panier
à Saint-Pierre et à Miquelon
à la une et à la deux
à la mygale et à la fourmi
au zist et au zest
à votre santé et à la mienne
au bien et au mal
à Dieu et au Diable
à Laurel et Hardy.

Prévert (Paroles Ed 1978, p 198)

RESUME

Pour les applications des bases de données multimédia (bureautiques, médicales, CAO), on a besoin de : (1) décrire, stocker et traiter interactivement des objets considérés comme complexes parce qu'ils sont structurés et contiennent des informations de nature alphanumérique ou de nature multimédia : texte, image, voix, etc (2) gérer les aspects dynamiques des objets et (3) établir des liens entre les objets de la base et les objets apparaissant seulement au niveau du schéma externe de la base pour l'application.

Traditionnellement, le formulaire est soit un document servant à décrire, structurer et améliorer la circulation des informations au sein d'une organisation, soit un objet des interfaces entre l'Homme et la machine. Pour les bases de données multimédia, nous intégrons ces deux tendances en proposant un modèle de formulaires complexes et ses opérations associées. Notre modèle se rattache à la classe des modèles de données relationnelles "non sous première forme normale". Il offre un cadre formel pour décrire et traiter la structure, la dynamique et la présentation des objets d'une application comme des formulaires. Cette approche permet de mieux gérer l'intégrité sémantique d'une application en offrant une vision et un traitement homogène des objets d'une base de données au sens large (données classiques ou multimédia, formulaires, documents, etc).

Mots-clés : Formulaires, Bases de données multimédia, Modèle de données, Objets complexes, Algèbre, Dynamique des données, Schéma externe, Interfaces utilisateur.

TABLE DES MATIERES

Toutes choses étant causées et causantes, aidées et aidantes, médiates et immédiates et toutes s'entretenant par un lien naturel et insensible qui lie les plus éloignées et les plus différentes, je tiens impossible de connaître les parties sans connaître le tout, non plus de connaître les parties sans le tout.

Pascal (Ed. Brunschvicg, II, 72)

L'information, le plus vicieux des caméléons conceptuels

H. von Foerster

INTRODUCTION

Il est courament accepté aujourd'hui qu'un Système de Gestion de Bases de Données (SGBD) fournit des mécanismes pour gérer des objets élémentaires : entiers, réels, booléens, chaînes de caractères voir même des objets plus particuliers tel que les valeurs nulles et la date [ORA 84]. Aujourd'hui, nous voulons des SGBD offrant des outils pour décrire, stocker, retrouver et manipuler de nouveaux objets : du texte, du graphique, un programme, un document, une cellule (représentation particulière d'un circuit VLSI), une image satellite, une carte géographique, une radiographie, un électrocardiogramme, etc. Ces objets se rencontrent dans les nouveaux champs d'application des SGBD que sont la CAO, la bureautique, la cartographie, le dossier médical, etc. Ils sont regroupés sous le terme générique d'**objet complexe** multimédia. Vouloir offrir un SGBD capable de supporter des objets complexes provenant d'environnements différents est un sérieux pari. C'est pourquoi la plupart des travaux de recherche dans ce sens se sont limités à un domaine (bureautique, CAO, médical) de manière à mieux cerner les fonctionnalités nécessaires à ses applications.

Pour le domaine bureautique, nous nous sommes intéressés aux objets **formulaires**. Dans cette thèse, nous ne considérons plus le formulaire comme un objet des interfaces pour les SGBD ou un objet conceptuel pour les systèmes bureautiques mais comme un nouvel objet pour les bases de données multimédia. Bon nombre des applications bureautiques peuvent être vues comme des applications formulaires. Une application sera donc définie par un ensemble de formulaires qui constituent non seulement son interface avec l'utilisateur mais également le support des données ou objets qu'elle utilise et des opérations qu'elle réalise.

Comme nous l'avons déjà dit, les objets que l'on rencontre en bureautique sont complexes. Le modèle que nous proposons et ses opérations associées définissent un cadre homogène pour décrire et traiter ces objets comme des formulaires (complexes). Avec cette approche, la modélisation d'une application devient plus "naturelle" puisqu'on interprète le monde réel en termes de formulaires, objets privilégiés par les organisateurs pour la description et la circulation de l'information. Les idées soutenues ici sont concrétisées au moyen d'une mise en oeuvre hypothétique d'un Gestionnaire Intéractif de Formulaires (GIF). Ce système utilise le modèle proposé et fournit donc un ensemble d'opérations pour la définition et la manipulation de formulaires complexes.

Dans la suite de cette introduction, nous présentons brièvement les tendances actuelles dans le domaine des modèles de données pour objets complexes. Nous présentons ensuite notre approche. Finalement, nous donnons le plan de cette thèse.

Objet Complexe

Un objet est complexe parce qu'il contient plusieurs éléments combinés entre eux d'une manière qui n'est pas immédiatement claire pour l'esprit. Les éléments que l'on peut rencontrer dans un objet complexe sont de nature alphanumérique (entier, chaîne de caractères) ou multimédia : texte, image, graphique, voix, programme, etc. Cette nature multimédia fait qu'un élément peut lui-même être considéré comme complexe : on ne manipule pas aussi facilement du texte qu'un entier. Il est également volumineux : une image satellite est représentée par plusieurs mégabits. La combinaison ou composition des éléments entre eux définit ce qu'on appelle une structure pour l'objet. Cette structure est dans la plupart des cas hiérarchique : un rapport de recherche est composé d'un titre, d'auteurs, d'un résumé, d'une introduction, de différents chapitres, chacun composé d'un titre, de sections (liste de paragraphes), etc.

Pour décrire ces objets, des modèles de données ont donc été proposés; Ils sont considérés comme plus puissants parce qu'ils offrent des structures plus riches et des algèbres plus élaborées pour manipuler ces structures.

Un des premier pas vers un modèle d'objets complexes a été d'abandonner la condition de "première forme normale" imposée par le modèle relationnel. Des modèles de relations non en première forme normale (N1NF), qui introduisent la structure de relation (ensemble) dans un attribut de relation, sont apparus : le modèle NF2 [SCH 82, SCH 84], le modèle NF2 étendu [PIS 86], le modèle V-relationnel [BAN 82, ABI 84], le modèle de Fischer et Thomas [FIS 83], le modèle de Roth [ROT 85a]. On peut citer les systèmes de gestion de formulaires SPECDOC [KIT 84] et OPAS [LUM 82, SHU 82] qui utilisent cette classe de modèles.

D'autres approches ont consisté à offrir des modèles sémantiques [HUL 86]: modèle E-R [CHE 76], modèle fonctionnel tel que DAPLEX [SHI 81], FQL [BUN 79], OFE [MAI 85], IFO [ADI 86b], modèle RM/T [COD 79], modèle dynamique [BRO 81]. Les modèles dédiés qui sont des adaptations ou spécialisations de modèles sémantiques pour un domaine particulier offrent sans nul doute une meilleure adéquation des concepts et des opérations aux besoins de l'application

considérée. On peut citer le modèle TIGRE [VEL 84] et le modèle BIG [CHR 84, CHR 85] pour les applications bureautiques, le modèle et le système CADB [RIE 85, FAU 86] pour les applications CAO.

Néanmoins les modèles proposés ne considèrent pas un objet dans sa totalité (il est éclaté en relations, entités, ensembles, types ...) et ils restent pauvres en concepts pour la représentation de la dynamique des objets.

Nous pensons qu'un objet complexe doit pouvoir également être explicité autrement qu'en termes de structure, décrivant la démarche choisie pour sa décomposition. *"Nous devons abandonner l'approche "réductionniste" de l'objet qui consiste à constituer un objet en isolation de tout environnement et de tout observateur, à l'expliquer en vertu des lois générales auxquelles il obeit et des éléments les plus simples qui le constituent"* [MOR 77]. C'est en référence à l'organisation dans lequel l'objet circule, l'environnement dans lequel il se trouve (utilisateurs), et les interactions entre ses éléments ou avec d'autres objets que l'on va pouvoir l'expliciter. Un objet doit donc être expliqué également en termes de **dynamique**. Sous ce terme nous regroupons les différents niveaux de comportement de l'objet : la spécification - des interactions auxquelles participent l'objet (les opérations que l'utilisateur va vouloir faire sur l'objet, l'évolution de l'objet au cours du temps), - des interactions qui tissent l'organisation de l'objet (contraintes d'intégrité , actions spontanées, liens avec d'autres objets).

Certains modèles sémantiques [BRO 81, GIBS 84, RIE 85], offrent des concepts pour la modélisation de la dynamique des applications : opérations spécifiques à une application donnée, enchaînement contrôlé et cohérent des opérations pour former des transactions, expression de contraintes. Dans le modèle de Gibbs [GIB 84], on donne la possibilité de définir de la sémantique par des *triggers*. Cette sémantique n'est pas considérée comme une composante de l'objet. Des recherches plus récentes visent à introduire la notion de temps [ABI 87, BUR 87], la notion d'événement et d'opérations [LIN 87] dans les modèles comme support pour la gestion de la dynamique des applications. On cherche également à réduire le fossé entre langage de programmation (support des programmes de l'application) et le langage de manipulation de la base.

La tendance bases de données *orientées-objets* est intéressante pour la modélisation d'objets complexes. Les caractéritiques d'une représentation objet sont que : - chaque objet a une existence réelle (pas d'objets dépendants et d'objets indépendants), - qu'il possède une composante statique et une composante dynami-

que (encapsulation), - que l'on ait la notion d'héritage (des deux composantes) : les objets sont alors liés entre eux sous la forme d'une arborescence ou d'un treillis. Cette représentation a comme origine des outils comme les langages orientés objets tels que SMALLTALK [GOL 83] et les FRAMES [MIN 75, VIG 86]. Ces outils ont inspiré beaucoup de méthodes de représentation de la connaissance dans le domaine de l'intelligence artificielle [FIK 85].

Pour les bases de données *orientées-objets*, on cherche donc à introduire les caractéritiques de cette représentation dans les modèles de données [NEU 87, TSI 87]. Certains modèles sémantiques preuvent être considérés comme orientés objet. Dans [GIB 84], on autorise la définition de nouveaux types, les contraintes et les opérations associées par un mécanisme de types abstraits. Actuellement, on cherche à offrir des modèles permettant de décrire et traiter un objet dans sa totalité : c'est à dire fournir un formalisme pour la représentation d'une entité du monde réel sans avoir à l'éclater en différents objets. Les modèles d'objets complexes ou structurés se situent dans cette tendance [ABI 86a, ABI 87]. Trop souvent dans ces modèles le comportement de l'objet n'est pas considéré. D'autres recherches plus orienté objet (au sans langage objet) vise à intégrer la dynamique comme composante de l'objet [RIC 87]. Le modèle KDM [POT 86] est un modèle "hyper-sémantique" dans le sens où il intègre concepts de l'IA et concepts des modèles sémantiques (classification, aggrégation et généralisation). On ajoute de la connaissance aux objets par le biais d'heuristiques et de contraintes incertaines. Chaque objet a de la connaissance et des propriétés structurelles.

Dans le courant bases de données *orientées-objets*, peuvent être incluses les recherches actuelles pour les outils autour des SGBD : les L4G (Langage 4ième Génération) [LEP 87] et les interfaces bases de données [PLA 87a].

Notre approche

Une façon de voir les bureaux est de considérer une grande base de données que les utilisateurs accèdent pour accomplir leur tâches [GIB 84]. Pour cette approche, le formulaire est l'objet idéal tant au niveau modèle de données pour le bureau [LUM 81, LUM 82, SHU 82, TSI 82, BAR 84], qu'au niveau conception d'outils ou méthodes pour - le traitement des contraintes sur les données [FER 82, GEH 83], - l'automatisation des procédures bureautiques [ZLO 77, ZLO 82, DEJ 80, TSI 82].

Cette façon de voir les bureaux est aussi une façon de voir bon nombre d'applications informatiques. De ce fait, le concept de formulaire est utilisé par de nombreux outils construits autour des SGBD (L4G). L'idée de formulaire électronique apparaît immédiatement quand les utilisateurs veulent informatiser leurs applications. Mais il est alors souvent associé à un objet "interface" (sur écran) par lequel l'utilisateur décrit, manipule ou extrait des données de la base, leurs donne un certain format (rapport, diagramme). Les outils proposés sont orientés : soit 1) utilisateur final, on peut citer les outils QBF, VFE, RBF, GBF du système INGRES [LEP 87], soit 2) utilisateur programmeur : le formulaire est l'objet permettant de développer des applications bases de données avec une meilleure productivité. On trouve les outils : IAF du système ORACLE [ORA 84], ABF pour INGRES [LEP 87] commercialisation de l'outil FADS [ROW 82, ROW 85].

L'utilisation qui a été faite du formulaire montre bien qu'il s'agit d'un objet :
- facile à appréhender : permettant de comprendre et d'interpréter de manière naturelle le monde des bureaux.
- "ergonomique" : permettant une organisation rationnelle des données, des actions et des contraintes sur ces données.
- interactif : permettant d'améliorer la circulation de l'information dans les organisations et également entre l'Homme et la machine.

La plupart des travaux réalisés sur les formulaires privilégient un des aspects ci-dessus. Pour les bases de données multimédia, nous avons choisi d'avoir une vision du formulaire plus globale en proposant un modèle qui intègre les trois aspects et fournit ainsi un cadre formel pour la conception des nouvelles applications.

- On ne s'intéresse pas seulement à l'informatisation des applications manipulant des formulaires (gestion des bons de missions et des remboursements) mais également aux applications pour lesquelles une conception de leur(s) schéma(s) externe(s) en termes de formulaires est envisageable (devis descriptif d'un batiment en CAO [AUT 86], lettres de demandes, données administratives et compte rendu d'un examen pour le dossier médical [MUN 87], aide à la description d'un objet en CAO). Pour ces nouvelles applications, il faut décrire et traiter des données multimédia complexes, spécifier leurs comportements et leurs présentations (externes).

- Le formulaire que nous considérons est un nouvel objet pour les bases de données. Il possède une structure hiérarchique et peut contenir des données multimédia. Chacun des éléments le composant a un rôle et subit des actions. Nous considérons les aspects dynamiques du formulaire par le biais de son aspect interactif. Le formulaire possède de la connaissance qui va lui permettre de s'assurer au plus vite que l'information qu'il véhicule est structurée et non ambiguë. Il dispose également d'une ou de plusieurs présentations lui permettant de communiquer avec l'environnement dans lequel il se trouve : 1) affichage avec le menu des opérations que l'utilisateur peut réaliser, 2) impression.

Le formulaire tel que nous le voyons est complexe. Pour définir notre modèle de formulaires, nous nous sommes trouvés confronter aux diverses orientations de recherche prises pour les modèles d'objets complexes. Nous avons opté pour une représentation du formulaire en termes de **Formulaire Abstrait (FA)** et de **Présentation**. Un FA caractérise une classe de formulaires possédant les mêmes propriétés structurelles et sémantiques. Il est décrit par une composante statique (schéma) et une composante dynamique (règles). La méthode fournie pour la définition de cette composante repose sur l'utilisation du langage de manipulation des FA (interrogation et mise à jour) auquel peut être incorporées des fonctions particulières sur les données et des fonctions de programmation traditionnelles. La notion de schéma que nous proposons et l'algèbre associée s'intègrent dans le courant "modèles d'objets complexes ou structurés" du domaine bases de données. Par Présentation, nous entendons : - un format (style) de présentation du formulaire, et - une liste d'opérations spécifiques donnant les moyens d'actions des utilisateurs sur le FA présenté avec le format. Un formulaire est le résultat de l'association d'un FA avec une présentation.

Le modèle de formulaires que nous avons défini permet donc de décrire un formulaire par sa représentation interne, orienté objet : schéma + règles, ses représentations externes et leurs menus associés : Présentations. La réalisation du Gestionnaire Intéractif de Formulaires (GIF) sera faite en utilisant ce modèle. L'objectif de cet outil est de fournir un ensemble de mécanismes pour la définition et la manipulation de formulaires complexes. Cet outil se trouve sur un poste de travail communiquant via un réseau à un serveur de données. Devant son poste, l'utilisateur manipulera les formulaires de son application en activant des opérations spécifiques. Le programmeur définira une application en utilisant des formulaires particuliers. Cette approche s'apparente à celle proposée dans [ROW 85] et apparaît

pertinente si l'on considère les possibilités actuelles des postes de travail (écran bit-map, souris, éditeurs graphique, mémoire, etc).

GIF a une vision de la base de données en termes de FA. Tout objet de la base (FA, relations, entités, catalogues) est vu comme un FA. Cette vision uniforme des objets de la base offre au programmeur un seul outil d'appréhension de la réalité (le FA). Il n'a pas à connaître le modèle utilisé par le serveur et de ce fait ne connait qu'un seul langage : celui utilisé pour la manipulation des FA.

Plan de la thèse

Cette thèse présente le noyau de l'outil GIF : les mécanismes offerts pour la modélisation et la manipulation des formulaires complexes (des FA et des Présentations). Il est clair que disposant de ces mécanismes, on pourra alors construire l'environnement de définition de GIF : les formulaires pour la définition d'une application. La mise en place des mécanismes pour la manipulation de ces formulaires particuliers offrira l'environnement nécessaire à l'utilisation d'une application.

Au chapitre I, nous introduisons les diverses approches en matière de gestion de formulaires en nous appuyant sur quelques travaux et outils significatifs. Puis nous donnons au chapitre II, les exigences auxquelles doit répondre un système pour la gestion de formulaires complexes du point de vue de leur définition (section II.1) et de leur manipulation (section II.2). En conclusion de chaque section, nous discutons des solutions offertes par les travaux de recherche et de développement pour répondre aux exigences présentées. La section II.3 présente le cadre de travail et les objectifs de GIF. Nous introduisons les fonctionnalités de GIF par rapport aux exigences données dans les sections précédentes.

Le chapitre III présente la commande de définition d'un FA sans aborder ses aspects dynamiques : on s'intéresse au schéma du FA.

Au chapitre IV, on s'intéresse aux opérations pour la manipulation des données (de FA). Sont présentées :

- Les FA_opérations : (1) les opérations de l'algèbre de FA pour l'expression de requêtes sur les FA (section IV.1). On donne à ce niveau une définition formelle d'un FA. (2) les opérations pour la mise à jour d'un FA (section IV.2).

- Les FA_expressions qui sont une extension des FA_opérations (section IV.3). On propose des opérations réservées aux différents types de données rencontrés dans un formulaire et des fonctions de programmation. En intégrant ces opérations et ces fonctions aux FA_opérations, on augmente leur pouvoir d'expression.

Le chapitre V décrit la méthodologie choisie pour la description de la composante dynamique d'un FA. Elle est basée sur la notion de **règles**. Une règle peut être vue comme une extension de la notion de "trigger". Elle permet de modéliser la notion d'événement indispensable dans le domaine bureautique (condition d'activation de la règle), les actions devant être réalisées avant une opération (pré-actions) et les effets de celle-ci (post-actions). Elle fait intervenir des FA_expressions. Nous donnons (section V.2) des exemples de règles pour une qualification de celles-ci en règles de contrainte, de statut, de valeur (locale ou globale), de report, d'exception. Puis nous décrivons en section V.3 les problèmes liés à la gestion de la cohérence des règles de FA.

D'autres commandes de définition sont présentées au chapitre VI. Elles permettent de prendre en compte le caractère évolutif de la conception d'un FA : à tout moment, on peut vouloir détruire ou modifier une définition de FA (modifier son schéma ou une de ses règles, ajouter ou supprimer une règle).

Dans la chapitre VII, nous présentons les mécanismes de définition d'une **Présentation** qui admet deux composantes : le format et les opérations spécifiques. Nous décrivons ce qu'est un format et montrons comment peut être construit un format par "défaut" pour un FA (section VII.1). Pour la seconde composante (les opérations spécifiques), nous proposons une approche basée sur la notion de schéma de comportement (section VII.2). Nous introduisons dans la section VII.3, les opérations pour la manipulation des Présentations. Nous montrons également comment peut évoluer la définition d'une Présentation. Un formulaire sur écran est le résultat de l'association d'un FA avec une Présentation. En section VII.4, nous donnons quelques exemples de formulaires particuliers pour l'environnement de définition de GIF.

Finalement, le chapitre VIII contient les conclusions de la thèse : ce qui est fait, ce qui n'a pas été fait, ce qui reste à faire, ce qui devrait être fait ...

CHAPITRE I

FORMULAIRES et BASES de DONNEES

Nous commençons à comprendre le jeu,
mais nous ignorons encore tout du joueur.
P. Veudryes

CHAPITRE I

FORMULAIRES et BASES de DONNEES

Selon le Larousse, le **FORMULAIRE** est un recueil de formules. C'est à dire la réunion de divers modèles contenant les termes formels dans lesquels un acte doit être conçu. Pour communiquer de l'information d'une manière structurée, intelligible et non ambiguë, il faut se donner un modèle d'échange de l'information. Ceci explique que dans notre société et principalement dans les bureaux, on rencontre couramment des formulaires. Un formulaire papier est composé de texte préimprimé et de plages pouvant reçevoir des informations. Il autorise la communication d'informations selon des conventions établies depuis longtemps et parfaitement connues de tous. Ces conventions (pas de modification de l'information fixe, valeurs dans les champs qui peuvent être simplement des croix,...) n'ont pas été mises en place par hasard : elles permettent d'assurer la construction de formulaires dont le contenu (valeurs des champs) est non ambigu et compréhensible par les divers traitements (bureautiques ou de données). Le formulaire offre un support (structure et sémantique) pour l'information manipulée dans les bureaux et plus généralement pour l'information stockée dans une base de données.

Dans ce chapitre nous donnons quelques exemples de modèles et d'outils construits autour du formulaire. Nous essayons de dégager les possibilités de définition et de manipulation qu'ils fournissent. Nous mettons l'accent sur les approches adoptées pour la description de la structure, de la dynamique et de la présentation (externe) du formulaire.

I.1 Utilisation du formulaire en bureautique

Dans le domaine de la bureautique, on peut voir le bureau comme une base de données à laquelle les utilisateurs accèdent pour accomplir leur tâches. Avec cette approche, on doit alors fournir des mécanismes pour décrire des données particulières aux bureaux (texte, image, documents, messages, etc), des contraintes sur ces données. Il faut également fournir des langages autorisant les utilisateurs à décrire leurs activités. Dans la suite de cette section, nous présentons quelques systèmes qui ont adopté cette approche et qui bien sûr traitent des formulaires. Nous verrons que le formulaire prend des significations diverses selon que l'on

s'intéresse à la définition d'objets pour la réalisation de tâches locales (OBE [ZLO 82]), à la conception d'applications bases de données destinées aux bureaux (OPAS [SHU 80]), à la définition d'un modèle de bureau (OFS [TSI 82]). Dans [GEH 84], on s'intéresse plus particulièrement à la sémantique d'un formulaire.

I.1.1 OBE : Office By Example [ZLO 82]

Le formulaire est ici la généralisation de la représentation graphique d'une relation (table) dans QBE [ZLO 77]). C'est donc un objet sur l'écran. Contrairement à la représentation "table", associée automatiquement à une relation, le formulaire lui doit être spécifié explicitement par l'utilisateur d'OBE. Il est défini graphiquement, puis chaque champ est décrit plus précisément de la même manière que les attributs d'une table. La Figure 1.1 montre la définition du formulaire *INVOICE* (facture). Après avoir construit graphiquement le formulaire (a), l'usager met des *variables exemples* dans les champs à définir. OBE affiche alors une table (b) de nom *INVOICE* avec des colonnes ayant comme noms les variables exemples. Dans cette table, on trouve également sur les lignes des attributs : TYPE, FW, etc pour la description des colonnes. L'utilisateur donne alors des valeurs à ces attributs pour chaque colonne.

(a)

1. INVOICE

CUSTOMER: _X

ITEM	QUANTITY	UNIT PRICE	EXTENDED
_I	_Q	_UP	_EX
		TOTAL	

(b)

TABLE	INVOICE	CUSTOMER	ITEM	QUANTITY	UNIT PRICE	EXTENDED
		_X	_I	_Q	_UP	_EX
TYPE	I.	CHAR	CHAR	FIXED	FIXED	FIXED
FW	I.	15	10	8	10	12
FLINE	I.	YES	NO	NO	NO	NO
PB	I.	YES	NO	NO	NO	NO
SV	I.	NO	NO	NO	NO	NO
OP	I.	YES	YES	NO	NO	NO

Figure 1.1 : exemple de définition d'un formulaire dans OBE

Une fois le formulaire défini, il est possible de copier des données de relations dans cette structure. La Figure 1.2 donne le programme à écrire pour construire une facture pour chaque client.

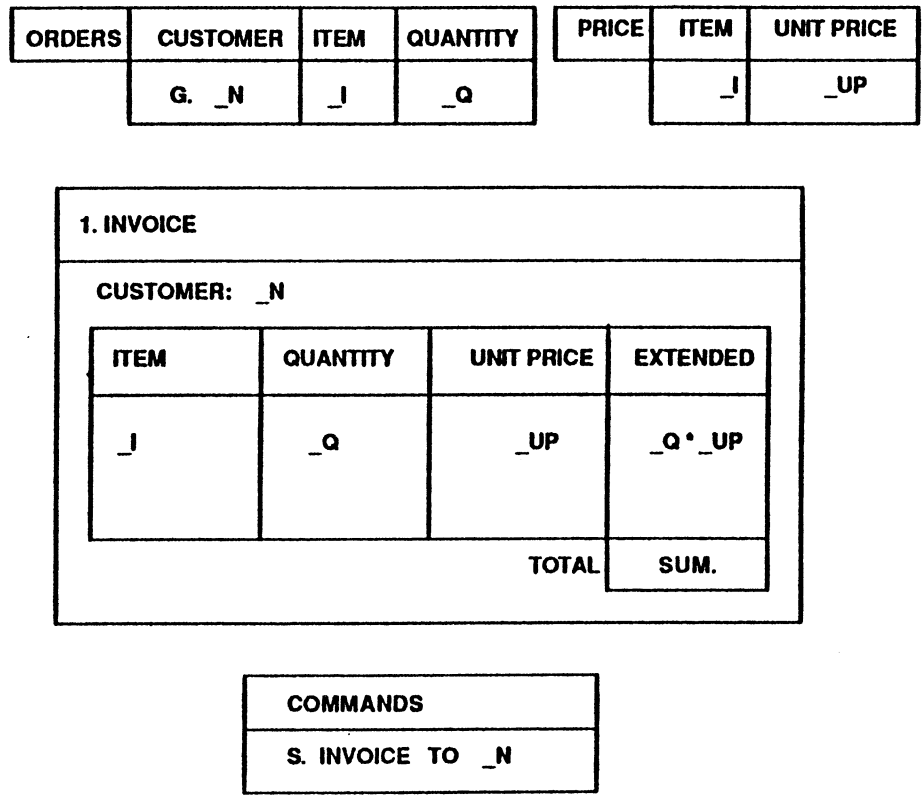


Figure 1.2 : exemple de programme pour produire une facture

Cet exemple montre que les champs d'un formulaire sont associés avec des champs (colonnes) de "tables". De ce fait, on utilise le langage OBE pour exprimer des interrogations ou des mises à jour de données sur les champs d'un formulaire.

Dans OBE, le formulaire est utilisé pour donner une nouvelle structure aux données des relations de la base et pour décrire la réalisation d'une tâche particulière : production d'une facture pour l'exemple. Du point de vue du modèle relationnel, le formulaire recouvre le concept de vue. Aucune indication n'est donnée dans [ZLO 82] pour la mise à jour de données (de relations) à partir d'un formulaire. Comme il est possible de définir pour un formulaire (au même titre qu'une table), des contraintes d'intégrité et des *triggers*, on peut supposer que le report des mises à jour est spécifié en utilisant ces mécanismes.

I.1.2 OPAS : An Office Procedure Automation System [LUM 82, SHU 82]

Dans [SHU 80], on propose un outil formel pour la conception d'applications bases de données. Son but est de faciliter la spécification des besoins d'une application en offrant un outil de dialogue pouvant être compris aussi bien de l'analyste que du demandeur. Le système OPAS [LUM 81, LUM 82, SHU 82, SHU 83] qui a été développé, repose sur le concept de formulaire. Il intègre un ensemble d'outils pour la description des données d'une application, des contraintes et des traitements (procédures bureautiques) devant être réalisés sur ces données.

Le formulaire possède ici une structure hiérarchique. Dans [SHU 85], on situe le modèle de formulaires comme similaire au modèle NT (Nested Tables) du système SPECDOC [KIT 84]. Ce modèle appartient donc à la catégorie des modèles N1NF. Dans [KIT 84], on présente les opérations algébriques pour la manipulation des NT (structure logique de formulaire). Par contre ceci n'apparaît pas clairement pour OPAS. A la spécification du formulaire, on donne certaines propriétés sémantiques (champ clé, unique, contraintes d'inclusion, d'exclusion, ...). Mais les types pour les champs restent standards. La Figure 1.3 donne un exemple de définition de *form*. Le formulaire *PROJECT* représente les projets des différents départements d'une entreprise. Il s'agit bien d'une relation non normalisée : la colonne PROJ est elle-même une relation et pour un département donné, on peut avoir un ensemble de projets.

DEFINE PROJECT

	(PROJECT)					
	DNO	MGR	(PROJ)			
			PJNO	(EQUIP)		COST
				NAME	USAGE	
DATA TYPE	CH(2)	CH(6)	CH(4)	CHV(8)	NUM(2)	NUM(8)
KEY	Y		Y	Y		

END

Figure 1.3 : Exemple de définition de *form*

Une procédure [LUM 81, SHU 82] est vue comme un traitement de formulaires (ou une série de traitements). Un traitement de formulaires consiste à utiliser un ou plusieurs formulaires (en entrée) et à produire un formulaire en sortie. A partir des différences de structures entre le(s) formulaire(s) en entrée et le formulaire en sortie, le compilateur du langage de programmation FORMAL génère le code pour réaliser la restructuration de données. Il permet de construire des procédures en utilisant la structure du formulaire : on donne l'opération (insert, update, ...) qui produit ou modifie les instances du formulaire, et une qualification de l'environnement pour l'opération et la procédure : SOURCE des données, ORDRE des instances, CONDITION sur les valeurs en entrée ou en sortie, etc. La Figure 1.4 donne un exemple de programme FORMAL qui construit le formulaire *DINFO* à partir des formulaires *PROJECT* et PERSON. Le formulaire DINFO donne pour chaque département, certaines informations sur ses projets et ses employés.

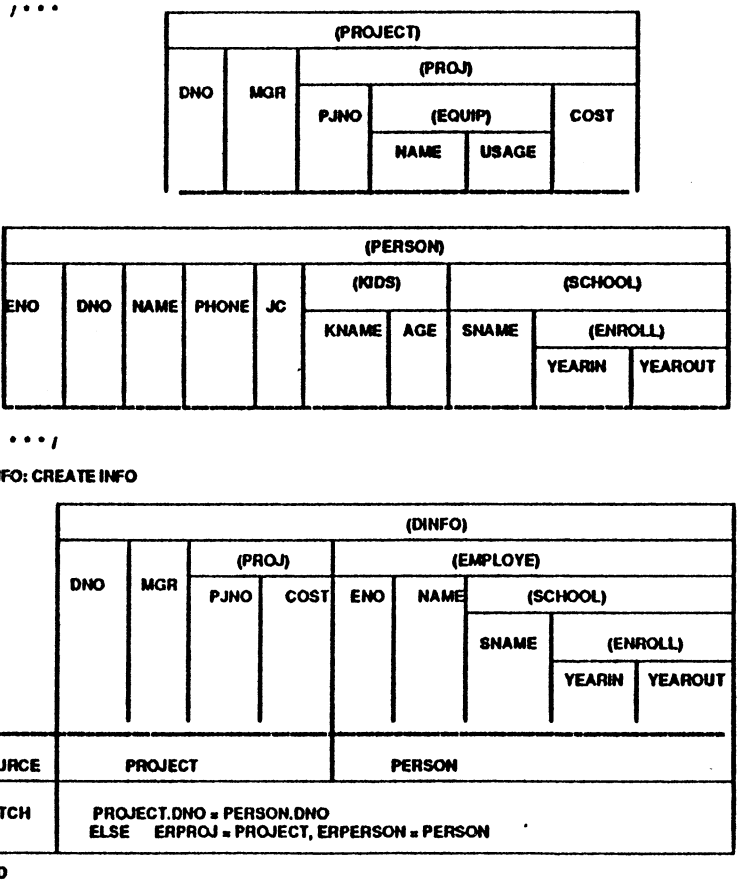


Figure 1.4 : Exemple de procédure OPAS

Le système OPAS et son langage FORMAL, offre un outil pour la conception d'une application. Le formulaire permet de définir de manière élégante l'information cruciale à une application : données, traitements. Il intervient comme outil d'analyse, de restructuration de données de la base, et de traitement de ces données (expressions de leurs sémantiques). Dans [SHU 85], on indique que les recherches s'orientent vers - un modèle de formulaires autorisant la communication de données appartenant à des bases différentes (modèle et système) et - une interface plus conviviale pour le langage FORMAL.

I.1.3 OFS : Office Form System [TSI 82]

Le système OFS [KOR 80, TSI 82] intègre divers outils utiles dans les bureaux (communication vocale, courrier électronique, langages d'interrogation, ...). Il repose sur les concepts de : *form type, form instance and form template.*

Un *form type* est (1) un ensemble d'attributs (voir l'exemple ci-dessous) : le formulaire au niveau logique est vu avec une structure plate, comme celle d'une relation du système MRS [KOR 79], avec lequel OFS communique. (2) un ensemble de procédures. Une procédure est associée à une opération standard sur le formulaire (saisie, modification, suppression de valeurs, stockage et envoi de *form instances* (occurrences) , etc). Elle code un ensemble de contrôles, d'actions à exécuter lorsque l'opération se produit : en fait à la place de l'opération standard, on réalise la procédure, l'opération devient alors spécifique.

Un exemple de *form type* :

> (Invoice#, Customer name, Customer adress, Date, Manufacturer, Item name, Unit weight, Unit cost, Discount rate, #items, Cost, Total, Taxes, Total Weight, Shipping charges, Amount payable, Status)

Le *form type* Invoice rassemble l'ensemble des données nécessaires pour le traitement d'une commande de marchandises. On peut avec ces données gérer les bons de Commandes, les bons de Livraisons, les ordres de Livraisons, les Factures et Relances aux clients.

Un langage de spécification de procédures bureautiques (en termes de formulaires) est proposé. La spécification d'une procédure est comparable à la définition d'un programme dans OBE. On indique par des scénarios sur des formulaires ce qui doit être fait : préconditions à vérifier, actions à exécuter et postconditions à vérifier. La Figure 1.5 montre la définition d'une procédure.

Dans le scénario de précondition (a), on indique le formulaire pour lequel s'applique la procédure : une Commande (*ORDER FORM*) de chapeaux "borsalino". Le scénario de précondition (b) lie une Commande avec le formulaire Inventaire (*INVENTORY RECORD*) décrivant la marchandise commandée. Le scénario action (c) décrit les actions modifiant le formulaire Commande obtenu par la précondition (a) : affectation de valeur au champ *Price* et *Total*. La valeur donnée au champ *Price* est extraite du formulaire Inventaire obtenu par la précondition (b) et celle donnée au champ *Total* est calculée par le programme *mult*.

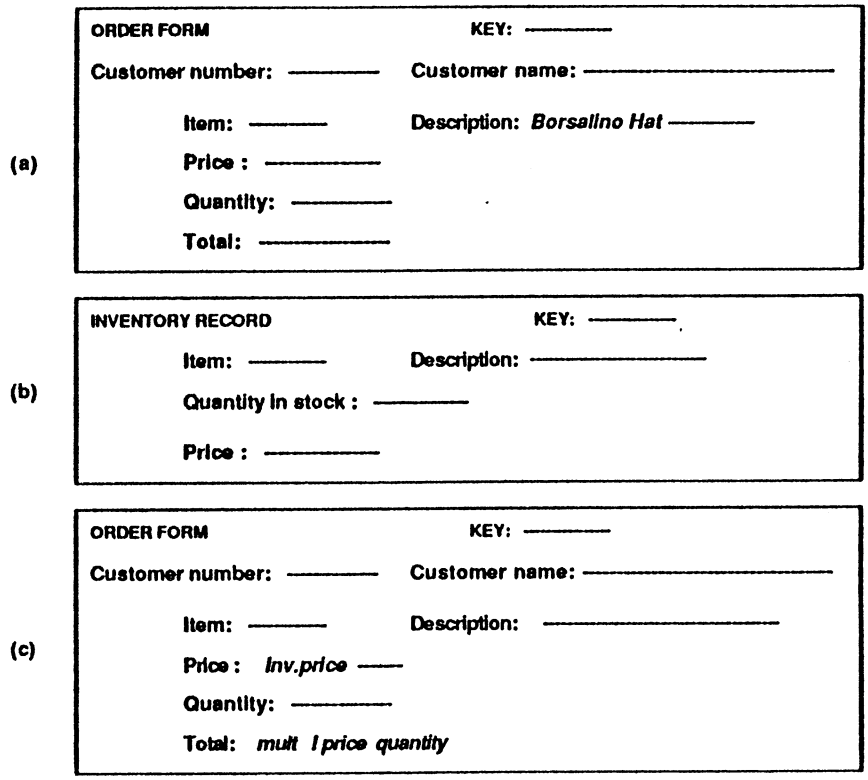


Figure 1.5 : Exemple de définition d'une procédure

Avec le concept de *form template* on peut décrire les diverses représentations externes d'un formulaire : formulaire papier, affiché, vocal, texte, ... Dans la Figure 1.6, on trouve les descriptions de trois *template* : - pour l'affichage d'un bon de Commande (a), - pour la sortie vocale d'une demande de Livraison des marchandises d'un bon (b), - pour l'impression de la Facture correspondant à un bon (c).

(a)

From Customer name **Reference** Invoice#
Date Date
Order the following items produced by Manufactuer
Item name **in quantity of** #items
Please bill and ship merchandise to his adress

(b)

Load #item **of** Item name **by** Manufacturer **for** Customer name
reference Invoice#

(c)

Customer name **XYZ Distributing**
Customer adress

Dear Mr* Customer name

Following our order Invoice# **placed on** Date **we have sent you under separate cover** #items **pieces of** Item name. **We hope you will find the merchandise satisfactory as** Manufacturer **has an excellent reputation. The item cost for** Item name **is** Unit cost. **However we are very happy to offer you a discount rate of** Discount rate. **This will reduce your cost to** Total. **On this we have to add taxes of** Taxe **and Shipping and handling of** Shipping charges. **Your total amount payable to us is** Amount payable. **Please pay this amount at your earliest convenience wihin the next month. If there is any problem please call us referencing to your order** Invoice#.
Thank you for your cooperation.

Figure 1.6 : Exemples de *templates*

Un contrôle du flux des informations est également proposé, consistant à déterminer les chemins pris par les formulaires circulant dans l'organisation (ensemble de stations de travail reliées par un réseau).

L'approche choisie ici montre bien les spécificités que doit avoir un système bureautique pour la description de ses applications : - description des données, opérations génériques sur les données dont l'applicabilité varie selon la sémantique qui leur est associée (procédures), - description de présentations pour les données, - description de la circulation des données.

Dans [SHU 85, TSI 82], on a beaucoup étudié les fonctions à mettre en oeuvre pendant les tâches bureautiques. Ce qui est appelé formulaire (*form* dans [SHU 85] et *form type* dans [TSI 82]) n'est rien d'autre qu'un cadre offrant un moyen d'organiser les données. Toute la sémantique du formulaire est représentée par du code dispersé dans divers modules. On retrouve la traditionnelle séparation entre propriétés structurelles et propriétés de comportement (sémantique) d'un

objet. Ceci ne renforce pas une attitude méthodique lors de la spécification de l'objet, rend dangereux tout travail de modification de l'objet et ne facilite pas son échange entre différents systèmes (portabilité du code). De nombreuses propriétés sémantiques (contraintes sur les valeurs des champs, droits d'accès) peuvent être décrites au moment de la définition de la structure de données.

Partant de cette constatation N.H Gehani [GEH 83] propose un langage et F. Barbic [BAR 84] un environnement pour une description des différents aspects d'un formulaire. Nous présentons ici le travail de N.H Gehani parce qu'il donne une approche très structurée du problème de la conception d'un formulaire, mettant en évidence ses caractéristiques principales.

I.1.4 High Level Form Definition in Office Information System [GEH 83]

Dans cette approche, on propose un mécanisme de haut niveau pour la description d'un formulaire. Un formulaire est défini par diverses parties : *imports, display, fields, operations, access rights, local data and routines.* Considérons le formulaire de la Figure 1.7. Il est utilisé par les employés pour une demande d'avance d'argent liquide et/ou pour le paiement d'un billet d'avion.

Cash Advance and Ticket Requisition Form :

Last Name, Initials : Date :
Company-Id : Room Number : Extension :
Organization : Case :

Employee only *Treasury Only*

Cash :
Tickets :
Total :

Funds to be used for :
Will be required until :
Total amount (in words) :
Signatures - Employee
Supervisor : Date :
Treasury : Date :

Figure 1.7: Le formulaire *Advance* à définir

Le mécanisme proposé permet de définir chaque composante du formulaire :

- *Imports* : contient les noms de modules utilisés dans le formulaire et qui sont définis par ailleurs. C'est par cet artifice que l'on s'interface à une base de données en décrivant, par exemple, un module fournissant les fonctions pour l'accès aux attributs d'une relation.

  Pour le formulaire de l'exemple, on utilise le module EMPLOYEE, qui fournit en particulier les fonctions pour l'accès aux informations d'un(e) employé(e) dans la base et une fonction pour valider une signature.

```
package EMPLOYEE is
    function EMP_NAME(ID:INTEGER) return STRING;
    function EMP_EXT(ID:INTEGER) return STRING;
    function EMP_ROOM(ID:INTEGER) return STRING;
    function EMP_ORG(ID:INTEGER) return STRING;
    function VALID_SIGNATURE(S:SIGNATURE)
        return BOOLEAN;
end EMPLOYEE;
```

- *Display* : dans cette partie, on décrit les positions et les valeurs des informations préimprimées du formulaire (texte). Y sont également décrites les positions et noms des champs. On ne présise pas comment cette partie est définie : par outil graphique, outil "questions-réponses". Il n'est pas possible visiblement d'introduire des informations autres que du texte (logo par exemple). La Figure 1.8 donne le contenu de cette section pour la définition du formulaire *Advance*.

```
                    Cash Advance and Ticket Requisition        Form : $no

Last Name, Initials : $Name                                    Date : $Date_Emp
Company-Id :  $Id          Room Number :$Room  Extension : $Ext
Organization :  $Org       Case : $Case

                        Employee only          Treasury Only

          Cash :  $Cash                        :  $Treas_Cash
       Tickets :  $Ticket_Price                :  $Treas_Ticket_Price
         Total :  $Total                       :  $Treas_Total

Funds to be used for :  $Purpose
Will be required until :  $Until
Total amount (in words) :  $Amount
Signatures -  Employee    $Emp_Sig
              Supervisor :  $Sup_Sig     Date :  $Date_Sup
              Treasury :  $Treas_Sig     Date :  $Date_Treas
```

Figure 1.8 : La composante *Display* pour le formulaire *Advance*

- *Fields* : contient les définitions des champs. On ne s'intéresse pas à la structure hiérarchique du formulaire ou plutôt on met à plat cette structure. Par exemple un tableau d'une colonne et n lignes d'un formulaire est décrit par n champs. Chaque champ est défini avec la syntaxe :

  pre {<condition> ⇒ <procédures>}
  <nom_champ> <type> <attributs>
  post {<condition> ⇒ <procédures>}

  <type> : caractère, chaîne de caractères, entier, réel, flottant, booléen, signature et date. Donc pas de type multimédia proposé.

  Les attributs d'un champ sont :
  - *required* : la valeur du champ est obligatoire
  - *virtual* : la valeur du champ est calculée. L'expression de calcul est donnée en termes de fonctions des modules de la partie Import, ou bien d'expressions arithmétiques.
  - *unchangeable* : la valeur du champ une fois donnée n'est plus modifiable.
  - *tag and variant* : la valeur du champ (tag) une fois donnée permet de déterminer la liste des champs (variant) pouvant être remplis. Du fait de la mise à plat de la structure du formulaire, on peut noter que les champs "variants", qui par définition devraient se rendre visibles à l'utilisateur lorsque le champ "tag" prend une certaine valeur, sont tous présents sur le "Display" donc toujours visibles à l'écran.
  - *ordered* : permet de préciser si le champ doit être rempli avant (before) ou après (after) un certain champ dont on donne le nom.
  - *lock* : pour spécifier que le champ doit être verrouillé (en écriture) après sa valuation. On ne propose pas de fonction pour deverrouiller un champ.

  "**pre**" introduit une pré-condition et "**post**" une post-condition. Selon le résultat de l'évaluation de la <condition>, on réalise les appels de procédures correspondant. Si la pré-condition n'est pas satisfaite, l'utilisateur n'est pas autorisé à remplir le champ. Dans la post-condition, on trouve la condition à évaluer après le remplissage du champ ainsi que les actions qui seront exécutées ensuite. On ne donne pas de précision sur le type des actions pouvant être réalisées.

La Figure 1.9 donne la composante *fields* de la définition du formulaire *Advance*

```
$No: INTEGER virtual NEXT();
$Name: STRING(1...30) required;
$Date_Emp: $Date required;
$Id: INTEGER range 0...99999 required
      post {$Name = EMP_NAME($Id)};
$Room: STRING(1...5) virtual EMP_ROOM($Id);
$Ext: STRING(1...4) virtual EMP_EXT($Id);
$Org: STRING(1...4) virtual EMP_ORG($Id);
$Case: STRING(1...10) required;
$Cash: FLOAT required post {$Cash >= 0.0};
$Treas_Cash: FLOAT required post {$Treas_Cash >= 0.0};
$Ticket_Price: FLOAT required post {$Ticket_Price >= 0.0};
$Treas_Ticket_Price: FLOAT required
      post {$Treas_Ticket_Price >= 0.0};
$Total: FLOAT virtual $Cash + $Ticket_Price,
      post {$Total <= 1500.0};
$Treas_Total: FLOAT virtual $Treas_Cash + $Treas_Ticket_Price,
      post {$Treas_Total = $Total};
$Purpose: STRING(1...100) required;
$Until: DATE required;
$Amount: STRING(1...100) required;
$Emp_Sig: SIGNATURE(1...6) required, lock all above,
      post {VALID_SIGNATURE($Emp_Sig)};
$Sup_Sig: SIGNATURE(1...6) required, after $Emp_Sig,
      post {VALID_SIGNATURE($Sup_Sig)};
$Date_Sup: DATE after $Sup_Sig;
$Treas_Sig: SIGNATURE(1...6) required,
      post {VALID_SIGNATURE($Treas_Sig)};
$Date_Treas: DATE after $Treas_Sig;
```

Figure 1.9 : Composante *fields* pour le formulaire *Advance*

On constate que les conditions s'expriment en utilisant les opérateurs de comparaison définis sur les types, les fonctions booléennes du module de la composante "Imports". On n'a pas la possibilité de dire qu'un champ doit avoir une valeur qui existe déjà dans un autre formulaire, parce que tout simplement il n'y a pas d'algèbre pour le calcul sur la partie *fields*. Ce qui sous-entend qu'il n'y a pas d'échange de valeurs entre formulaires.

- *Operations* : liste les opérations pouvant être réalisées sur le formulaire. Ceci fait voir le formulaire comme un type abstrait. Si aucune opération n'est spécifiée, le système utilisera les opérations génériques pour les formulaires (créer, éditer, envoyer, etc).

- *Access rights* : dans cette partie, on décrit pour chaque catégorie utilisateur de l'organisation auquel est destiné le formulaire ses droits (1) de modification sur les champs (2) d'exécution des opérations.
- *Routines* : décrit les données spécifiques et sous-programmes spécifiques au formulaire.

On regrette de ne pas avoir de renseignements concernant le stockage du formulaire (dans une base de données ?) et sa manipulation. On note deux choses : - la possibilité de verrouiller les champs mais non de les deverrouiller, - l'expression de la sémantique pour les champs ne permettant pas d'exprimer des contraintes entre formulaires ou des règles de calcul faisant intervenir des valeurs d'autres objets.

I.2 Utilisation du formulaire en bases de données

Le formulaire a également suscité l'intérêt des chercheurs dans le domaine strictement bases de données. Son utilisation pour la conception d'applications est reconnue. On offre aux programmeurs des outils pour décrire une application en termes de formulaires [ROW 82, ROW 85] ou de blocs [ORA 84]. Nous allons détailler un peu plus ces deux outils. Nous présentons également le logiciel 4ème Dimension [MAC 86], parce qu'il donne des indications sur la richesse et la complexité de ce que doit fournir un environnement pour la définition des différents aspects d'une application.

I.2.1 "Fill-in-the Form" Programming [ROW 85]

Les travaux de Laurence A. ROWE [ROW 82, ROW85] sont significatifs dans le domaine des générateurs d'applications bases de données relationnelles. Le prototype FADS [ROW 82] a été commercialisé pour le système INGRES sous le nom ABF (Applications By Forms). Dans [ROW 85], on utilise le terme ADE (Application Development Environment) pour nommer l'outil avec lequel le programmeur peut très facilement et très rapidement développer une application sur un SGBD relationnel. L'interface d'ADE est en termes de formulaires au même titre que les applications développées. Une application est décrite par une collection de *frames*. La Figure 1.10 donne un exemple de *frame*.

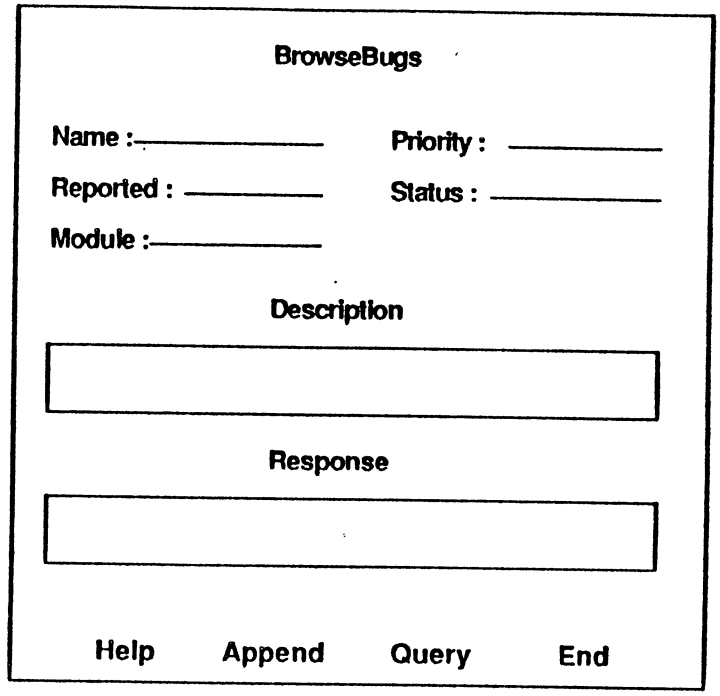


Figure 1.10 : Le frame *BrowseBugs*

Un *frame* se présente comme une boîte avec des boutons pour les opérations permises sur son contenu (*form*). Le frame est d'un certain type : *menu*, *query/update* (pour interrogation et mise à jour de la base : opérations par défaut), *report* pour la génération d'un rapport affiché ou imprimé, *user-defined* pour la définition d'un frame où les opérations pouvant être réalisées par les utilisateurs sont définies en utilisant le langage QUEL étendu (voir FADS [ROW 82]). Le système ADE peut lui même être vu comme une application car il offre au programmeur des *frames* de base : *"Edit Application"*, *"Edit Qyery/Update Frame"*, *"Edit Form"*, *"Edit Menu"*, etc. En progressant dans les différents niveaux de définition (*frames*) de ADE, le programmeur définit son application, effectue des tests pas à pas.

On utilise donc le frame *Edit Query/Update Frame* pour définir un *frame* de type *Query/Update*. La Figure 1.11 montre comment le *frame* de la Figure 1.10 a été défini.

Pour compléter cette définition, le programmeur doit définir le formulaire (*form*) au travers duquel seront présentées ou saisies les données et l'association entre ce formulaire et les relations de la base.

Un formulaire est toujours dans un *frame* de type *query/update* ou *user-defined*. La définition d'un *form* se fait avec un éditeur qualifié de "What-you-see-is-what-you-get". Le programmeur décrit la position des champs, ses attributs : texte d'aide, contrôle pour l'édition, statut.

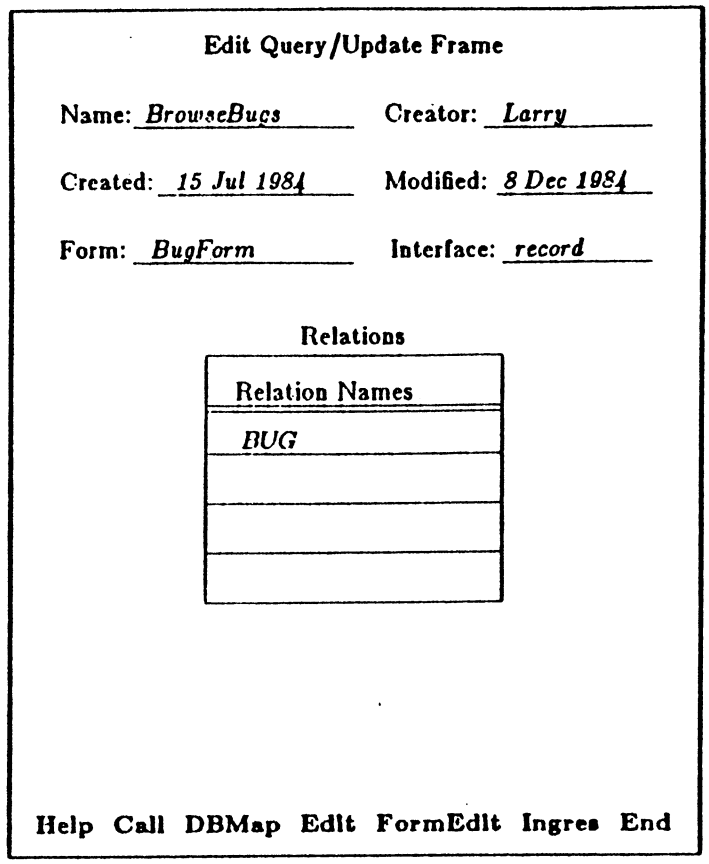


Figure 1.11 : Définition du frame *BrowseBugs*

Le programmeur peut ne pas définir de formulaire associé à un frame *query/update*. Il donne seulement le nom d'une ou de plusieurs relations et le type d'interface : *record* (affichage d'un n-uplet d'une relation), *table* (affichage de plusieurs n-uplets d'une relation) ou *master/detail* (affichage d'un n-uplet d'une relation et de plusieurs n-uplets d'une seconde relation liée ou non à la première). ADE construit alors le formulaire de manière automatique. Cette façon de faire est largement utilisée pour la construction de formulaires utiles à l'interrogation et la mise à jour des relations de la base.

Le frame *BrowseBugs* contient le formulaire *BugForm* (Figure 1.11). Ce formulaire est défini de manière automatique à partir de la relation :

BUG (name, priority, reported, status, module, description, response)

L'association entre relation(s) et *form* est décrite explicitement : pour chaque champ, on donne la désignation de l'attribut de relation correspondant. Dans le cas où le formulaire peut contenir des données de deux relations, on peut décrire le *JOIN* mais il faut alors préciser comment le report d'une mise à jour sur le formulaire doit être fait sur les relations. L'approche choisie est d'imposer des contraintes sur les associations possibles et de forcer le programmeur à choisir une manière de faire le report. Ceci rejoint les propositions faites dans [COL 83] où le formulaire est déjà plus complexe (vue au sens large sur une base de données relationnelles).

L'association entre le formulaire *BugForm* et la base est très simple puisque les données sont extraites d'une seule relation (voir Figure 1.12).

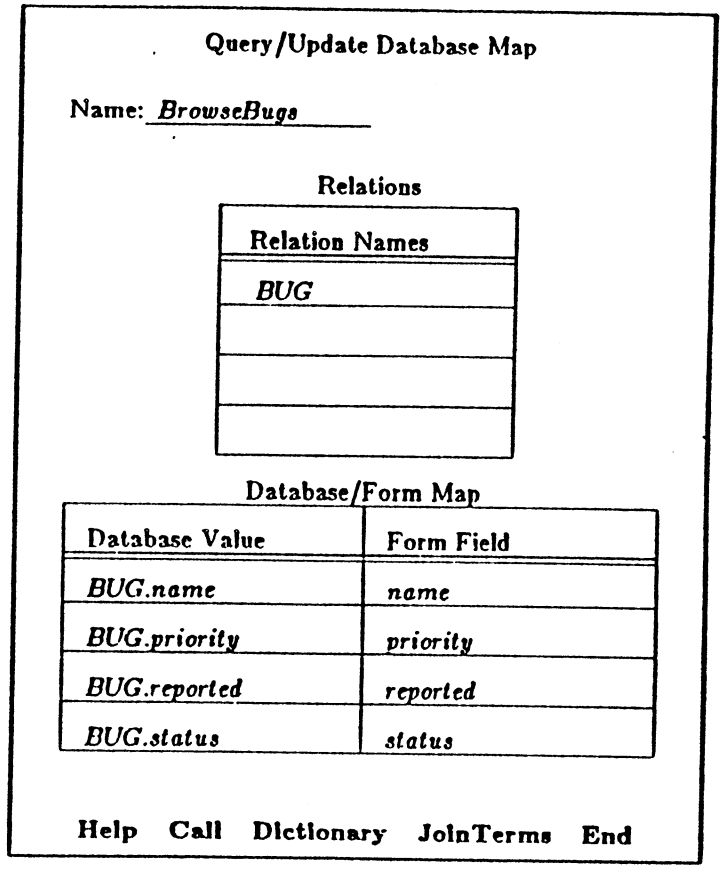


Figure 1.12 : Définition de l'association *BugForm* - BD

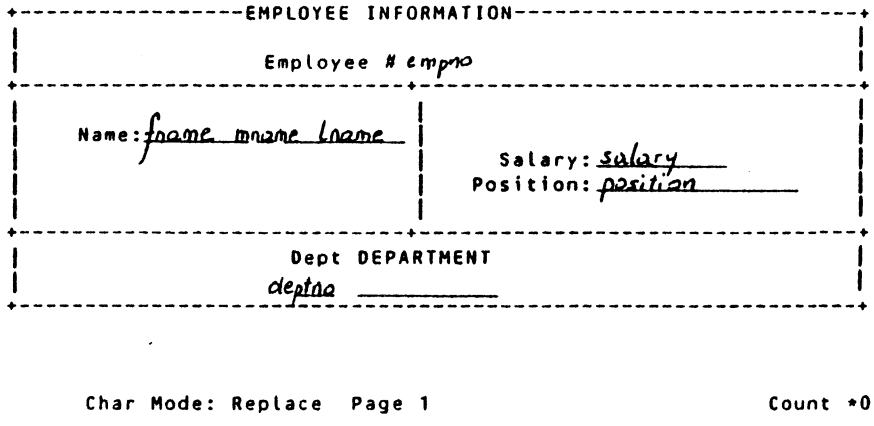
ADE présente de nombreux avantages : - facilité de développement d'une application, - nombreux objets prédéfinis : le domaine, la relation, le formulaire, le frame *query/update*, *menu* ou *report*, - un interface standard intégrant différents outils des LAG.

Il présente des défauts liés au matériel et au modèle sous-jacent. Une application INGRES est définie par des relations, des contraintes, des triggers. Les formulaires construits pour l'application décrivent son interface utilisateur. La sémantique d'un formulaire est celle de la ou des deux relations qu'il représente. L'exploitation de cette sémantique (vérification de contraintes, déclenchement de triggers) se fait lorsque l'application transmet à INGRES, les manipulations réalisées (opérations du *frame*). On n'a donc pas de contrôle immédiat des actions usagers ce qui va à l'encontre du concept d'interaction.

La recherche s'oriente vers : - l'utilisation d'un écran/clavier "bit-map" et d'une souris et - la prise en compte de nouveau objets prédéfinis utiles aux applications bureautiques (texte, image, document, ect ...).

I.2.2 IAF (Interactive Application Facility) d'ORACLE [ORA 84]

Une application pour IAF est une collection de formulaires (masques d'écran) dont disposera l'utilisateur final. L'interaction avec l'application se fait en utilisant les touches fonctions du terminal (classique). A un formulaire correspond une relation ou une vue. Un formulaire "vue" peut être mis à jour sous certaines conditions : - pas d'attribut de la vue, résultat d'une expression de calcul, - pas de *join* ou *group by* dans l'expression de la vue, - tous les attributs "non null" des relations sources doivent appartenir à la vue.



Le formulaire *EMPLOYEE INFORMATION* ci-dessus permet de manipuler les données : EMPNO, FNAME, MNAME, LNAME, SALARY, POSITION, DEPTNO de la relation EMPLOYEE dans les champs : *empno*, *fname*, *mname*, *lname*, *salary*, *position*, *deptno*. Avec ce formulaire, on pourra par exemple changer le salaire d'un employé, l'affecter à un nouveau département en utilisant les

champs : *deptno* et *dname*. Ces champs correspondent aux données DEPTNO (code) et DNAME (nom) de la relation DEPT(département). La donnée DEPTNO est présente également dans EMPLOYE.

IAF est composé de deux utilitaires :

- IAP (Interactive Application Processor) qui exécute une application. Cet outil est vu comme un programme d'application du SGBD ORACLE.
- IAG (Interactive Application Generator) donne au programmeur les moyens de définir une collection de formulaires en répondant à toutes une série de questions. Le résultat d'une définition est un fichier (questions + réponses) géré par le système sur lequel est installé ORACLE. La description (source) d'une application n'appartient pas à la base. Elle devient "ressource" pour le programme IAP au moment de l'activation de l'application.

Nous renvoyons le lecteur à [ORA 84] pour une description complète d'IAF. Nous nous intéressons ici essentiellement aux moyens offerts au programmeur pour décrire la sémantique de son application.

1- Expressions SQL associées à un champ C1 :

question : execute a SQL statement ?
SQL>

On donne alors trois composants :

(1) une requête (SQL étendu) à exécuter dès que l'on value le champ,

(2) un message à afficher si la requête n'a pas retourné de n-uplets,

(3) la signification de ce non aboutissement de la requête : avertissement ou demande de nouvelle valeur pour le champ.

L'exemple ci-dessous donne l'expression SQL associé au champ *dptno* du formulaire *EMPLOYEE INFORMATION*.

execute a SQL statement ?
SQL> *SELECT DNAME*
SQL> *INTO dname*
SQL> *FROM DEPT*
SQL> *WHERE DEPTNO = :deptno* (1)
SQL>
Message if value not found : *No such department number ..... try again;* (2)
Must value exist Y/N : *Y* (3)

Lorsque le champ *deptno* (code d'un département) de *EMPLOYEE INFORMATION* reçoit une valeur, on recherche dans la relation DEPT s'il existe un nom de département (DNAME) correspondant au code. Si on retrouve un nom, on

affecte cette valeur au champ *dname*. Si par contre on ne retrouve pas de département, l'utilisateur doit donner une nouvelle valeur au champ *deptno*.

Dans cet exemple, on constate que :
- Un champ est référencé par son nom préfixé par ':' (clause WHERE).
- Le résultat d'une requête peut être donné à un champ C2 (clause INTO après SELECT). L'effet cascade est limité : le champ C2 ne doit pas avoir déjà une valeur, pas d'exécution de la requête associée à C2 si celui-ci précède C1 (la définition d'une application détermine un ordre sur les formulaires et les champs).
- Une même donnée est désignée par trois noms : 1) un nom de champ, pour l'interface, 2) un nom de variable champ = ':' le nom du champ, pour l'expression de la sémantique, 3) un nom d'attribut pour la base. Ceci ne facilite pas la modification des spécifications d'une application.

A partir de ce mécanisme de base, ont été élaborées de nombreuses extensions : écriture de plusieurs requêtes, utilisation d'une requête pour comparer des valeurs de champs, pour effectuer des calculs, pour donner une valeur par défaut, pour tester si une valeur de champ appartient à une liste, etc. Mais cela nécessite de passer par des artifices : utilisation du catalogue DTAB, de la relation "bidon" DUMMY ne contenant qu'un seul n-uplet.

2- Des triggers

Ils s'activent pour une opération sur la base, ou une opération d'affectation de valeur à un champ. On tient à faire remarquer qu'il n'y a pas d'autres possibilités de définir des actions spontannées dans ORACLE.

A chaque formulaire peuvent être associés des triggers, s'exécutant avant/ après la recherche, l'insertion, la modification ou la suppression de données. Le corps du trigger est une extension de ce qui peut être fait pour l'attachement d'une requête SQL à un champ : ce sont des expressions SQL d'interrogation ou de mise à jour.

Field name : **pre_insert*
SQL> *SELECT MAX (EMPNO) + 1*
SQL> *INTO empno*
SQL> *FROM EMPLOYEE*
SQL>
Message if value not found : *Impossible error;*
Must value exist Y/N : *Y*

L'exemple ci-dessus présente un trigger utilisé pour donner un numéro au nouvel employé que l'on veut insérer dans la base.

L'exécution des triggers se fait dès que l'utilisateur demande la validation de ses manipulations. Une transaction ORACLE débute par la réalisation effective des suppressions puis des insertions et/ou des modifications dans la base. Pour chacune des opérations, on déclenche les triggers correspondant (avant ou après). Si une expression SQL d'un trigger échoue alors on défait la transaction. L'utilisation du caractère * pour les aspects (2) et (3) d'une expression SQL dans un trigger permet de préciser l'effet de l'échec de ce trigger sur la transaction (annulation ou continuation). Ceci rend la sémantique du trigger pas toujours claire, un peu "bidouillesque". Tout cela parce que l'on veut programmer avec SQL des "si alors sinon". De plus, nous avons constaté que le traitement d'un ensemble de n-uplets, résultat d'une expression SQL de trigger, pose des problèmes.

Avec IAF, une application est définie par un fichier (extérieur à la base) contenant les spécifications de la présentation et de la sémantique de ses données. Ceci demande le maintien "manuel" de la cohérence entre les spécifications de l'application et la base. Pour deux applications manipulant les mêmes relations, on risque d'avoir de la redondance d'informations.

FASTFORM

L'utilitaire FASTFORM permet la définition d'une application IAF seulement en donnant le nom des relations désirées. Les informations nécessaires à la construction du fichier "questions-réponses" décrivant l'application sont extraites du catalogue de la base. Dans ce fichier il n'y a pas bien sûr de définition d'expressions SQL ou de triggers. Mais l'utilisateur peut toujours modifier le fichier (questions-réponses) construit par FASTFORM. L'exemple ci-dessous montre l'utilisation de cet utilitaire pour construire le formulaire de la Figure 1.13, correspondant à la relation EMPLOYEE.

Enter Application Name : *APPLI_EMPLOYEE*
Enter username : *CC*
Enter password :
Enter TABLE name or <return> : *EMPLOYEE*
Include all columns (y/n)? *Y*
Up to 18 rows can be displayed on page 1.
How many rows should be displayed? *12*
Enter TABLE name or <return> :
Creation of application APPLI_EMPOYEE completed

= EMPLOYEE =

EMPNO : --------
MNANE : -
ADDRESS : ----------
ST : --
HIREDATE : ---------
POSITION : ---------

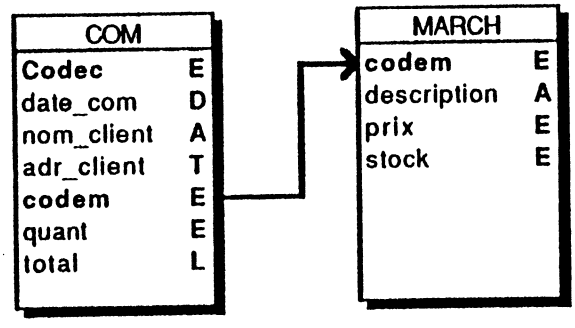
FNAME : ----------
LNAME : ----------
CITY : ----------
ZIP : ---
SALARY : ------
DEPTNO : -------

Figure 1.13 Formulaire pour la relation EMPLOYEE

I.2.3 Le logiciel 4ième DIMENSION (4D) [MAC 86]

Le logiciel 4ième DIMENSION (4D) conçu pour le Macintosh comme un SGBD, recouvre les concepts de : fichier, format, formule fichier/format, procédure et fonction, langage de programmation, menu.

Les fichiers : suites finies de fiches constituent la base de données. Une fiche est une suite finie de rubriques. Une rubrique peut être de type Alphanumérique, Texte, Numérique, Entier, Date, Image, et Racine pour l'introduction de sous-structure (pointeur vers un ensemble de fiches). On peut également imposer des liens entre rubriques des fichiers.



Exemples de fichiers

Le fichier *COM* permet de stocker une commande (d'une marchandise). Le fichier *MARCH* stocke les marchandises. Le lien entre les fichiers permet à partir du fichier COM, d'effectuer une recherche sur le fichier MARCH.

Le format est le moyen par lequel vont s'afficher ou s'imprimer les données des fiches. On peut associer différents formats à un fichier. 4D offre un éditeur graphique pour la définition de formats personnalisés et un générateur de format standard (à partir de la description d'une fiche). Un exemple de format standard pour le fichier *COM* est donné en Figure 1.14.

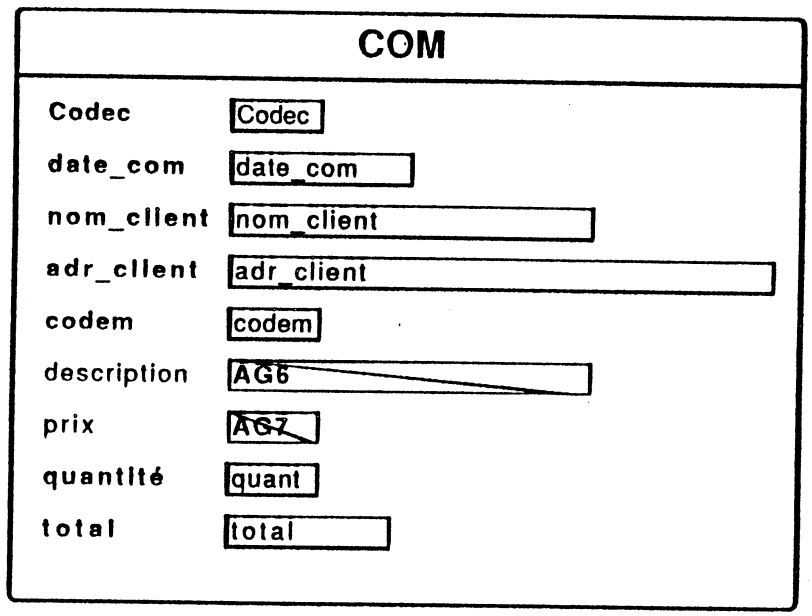


Figure 1.14 Le Format *FTCOM* pour le fichier *COM*

Dans un format, on trouve :

- la description des gabarits des rubriques. Par exemple, pour une rubrique de type image, on a le choix entre trois gabarits d'affichage : tronquée, Non tronquée et sur Fond (image fixe). Pour une zone texte, on a un défilement variable.

- des variables : pour la visualisation de données, non stockées dans le fichier auquel est associé le format. On peut également avec ces variables visualiser le résultat d'un calcul, etc.

- des boutons : pour la validation, la non validation, l'activation d'autres actions sans être obligé de quitter là saisie, pour rendre effective une option. Ces derniers sont appelés boutons radio. A chaque choix possible est associé un bouton : lorsque l'utilisateur clique sur un des boutons, l'option correspondante est retenue.

- des boîtes à cocher : pour rendre effective ou non une option. A la différence des boutons radios, on peut ici rendre effective plusieurs options.
- les zones de graphes (pour la génération de graphes : barres, barres empilées, secteurs, etc), de défilement.

4D offre un ensemble de routines standard pour la gestion de la sélection d'un fichier, des ensembles, des fiches, de la saisie, du contenu des formats, des sous-fiches, des liens. Il fournit également des opérations particulières sur les types de données : chaînes de caractères, images, numériques, dates. La fonction Ancien appliquée sur une rubrique retourne sa valeur avant modification. D'autres routines sont fournies pour le dialogue, les menus, l'impression, la communication, les interfaçages, etc. Nous laissons le lecteur se reporter au manuel pour avoir plus de détails. A partir de ces routines, le programmeur construit ses formules. Si une routine lui manque, il peut la programmer. Il construit également des procédures pour traiter ces fiches de manière automatique.

Les formules sont stockées dans un fichier particulier (ressources) et non avec les objets dont elles dépendent. Du point de vue conceptuel, la sémantique de l'application est dispersée entre "format" et "fichier". On distingue les formules :

- **format**, associées à un format et exécutées chaque fois que celui-ci est utilisé. On peut ainsi : imposer un gabarit (majuscules, minuscules) pour les rubriques, donner des valeurs par défaut à certaines rubriques ou variables, contrôler la saisie.
- **fichier**, associées au fichier et exécutées à chaque fois que le fichier est utilisé. Elles sont exécutées avant les formules formats.

La Figure 1.15 donne un exemple de formule pour le format *FTCOM* de la Figure 1.14. Cette formule permet de donner au champ *date_com* une valeur par défaut (la date du jour) et de vérifier que la commande demandée peut être satisfaite : on vérifie qu'il y a suffisament de marchandises en stock.

```
Si (Avant saisie)
   Si (date_com=!00/00/0000!)
      date_com:=Date du jour
   Fin de si
Fin de si
CHARGER SUR LIEN(codem)
AG6:=[MARCH]description
AG7:=[MARCH]prix
Si (Pendant saisie)
   Si (([MARCH]stock-[COM]quant)<0)
      ALERTE(" La commande ne peut peut pas être acceptée pour cette quantité")
      quant:=0
      REFUSER(quant)
   Fin de si
   total:=quant*AG7
Fin de si
```

Figure 1.15 : Formule Format

4D est sans nul doute un outil de développement d'application puissant. Il nécessite du programmeur une adaptation au langage qui se fait rapidement compte tenu de l'interface proposée. Mais avant la programmation, il faut faire une bonne analyse globale de l'application dans les termes de 4D. Cet outil montre qu'un générateur d'application doit offrir un langage de manipulation de données (au sens bases de données), des fonctions de programmation classique, des fonctions de manipulation des objets interfaces (formats, boutons, dialogue, etc). Ce qui séduit toujours c'est le modèle de présentation du Macintosh.

I.3 Conclusions

Il est clair que l'utilisation du concept de formulaire pour aider à la modélisation d'une tâche bureautique [SHU 80, SHU 82, LUM82], d'un objet bureautique [ZLO 82, GEH 83] ou plus généralement d'un système bureautique [TSI 80,TSI 82, BAR 84] a permis de mieux cerner les besoins (en outils et méthodes) de l'homme ou de la femme face à son poste de travail. Par ailleurs, les générateurs d'applications [ROW 85, ORA 84] mettent bien en évidence les manques des SGBD et de leurs langages pour la gestion de la dynamique d'une application. En effet pourquoi toujours traduire la dynamique d'une application par son interface et ses programmes. On peut penser à la traduire par la dynamicité des objets bases de données qu'elle utilise et par la notion de présentation.

Les principales limitations rencontrées aujourd'hui dans les systèmes de gestion de formulaires sont dues essentiellement :

- au type de matériel utilisé. En général les interfaces sont développées pour des écrans classiques et la manipulation des formulaires est standard et figée. L'ensemble des systèmes offrent des opérations de manipulations (créer, éditer, envoyer, rechercher, ...) activables avec n'importe quel type de formulaire. Très souvent l'édition des champs se fait dans un certain ordre, due à la manière dont sont décrites les actions à réaliser sur ces champs. Il n'y a pas de possibilité de changer d'environnement de travail permettant d'avoir une utilisation plus souple du formulaire autorisant son reformatage (détruire, ajouter, déplacer un(des) champ(s) ou des informations préimprimées).

- à la nature très simple ou simplifiée du formulaire :

  Un champ ne peut pas être décrit avec le type Temps, Croix, Texte, Image, Graphique, Tableau (multimédia). La représentation interne du formulaire est : une aggrégation de champs [GEH 83], d'attributs [BAR 84, TSI 82], des relation(s) pour OBE [ZLO 82], ADE [ROW 82], IAF [ORA 84], des fichiers (4D). Dans un formulaire, on peut trouver des éléments qui sont en fait des listes d'éléments, des mises en séquence d'éléments, des aiguilleurs pour un choix entre différents éléments. Il n'y a pas de constructeurs offerts pour hiérarchiser la structure de donnée associée au formulaire. Dans [SHU 85, KIT 85], on offre en plus du constructeur n-uplet : - le constructeur d'ensemble (relation). Dans [BAR 84], on offre le constructeur choix.

- La dynamique du formulaire représentée par : - des programmes d'application ou procédures [TSI 82, SHU 85], - des attributs de contrôle et des opérations spécifiques [ROW 85], - des formules [4D], - des expressions SQL et des triggers dans IAF [ORA 84]. Cette dynamique est spécifiée au même niveau que les objets de présentation de l'application (formulaires). Elle n'est pas considérée comme objet de l'application, donc n'appartient pas à la base de données.

Pour une application, on constate donc que :

- le formulaire n'est pas considéré comme un objet de l'application mais comme un objet de présentation pour l'application,
- les objets de l'application sont des objets de SGBD classiques, pour lesquels l'expression de la sémantique repose sur l'utilisation de programmes d'applications.
- le support des objets de présentations sont des écrans classiques, des éditeurs de tableaux, de lignes de texte.

Compte tenu des évolutions en matière de poste de travail (écran graphique, souris, sortie vocale) d'une part et de celles des bases de données pour la prise en compte d'applications multimédia d'autre part, les recherches s'orientent dans de nouvelles directions :

- offrir des modèles de données plus riches pour capter la sémantique des nouveaux objets dit complexes comme un document, un circuit intégré, etc.
- offrir des outils (générateurs d'applications, d'interfaces) autour des SGBD utilisant toutes les possibilités des postes : UC, fenêtre, menu, icônes, ... et intégrant les environnements logiciels nécessaires à la manipulation de données multimédia : éditeurs, traitement d'images, tableurs, etc.

Des modèles d'objets complexes pour traiter l'objet dans sa totalité sont proposés [ABI 86a, ABI 87]. Dans [NEU 87], le modèle proposé est *orienté objet*. Il présente l'avantage de permettre la définition d'un objet dans toute sa totalité : description, structure et *methods* (opérations). Pour le domaine bureautique, souvent les modèles sont également *orientés objets* [GIB 84, TSI 85, TSI 87].

Pour les outils autour des nouveaux SGBD, des études sont en cours [LEP 87, PLA 87a, PLA 87b]. On cherche essentiellement à combler le fossé entre interface et SGBD. On propose des outils pour le développement d'interfaces et d'applications plus flexibles en offrant d'intégrer : le langage de programmation et le langage d'accès à la base, les aspects transactionnels, les environnements logiciels pour le traitement des données multimédia sous un même formalisme (structure de données, présentations), etc.

Que devient le formulaire ?

Dans cette mouvence de nouveaux modèles de données et de nouveaux outils autour des SGBD, le formulaire peut présenter encore un intérêt si l'on sait le généraliser.

- Le formulaire est **un objet complexe** :
  1- Les informations qu'il peut contenir sont de nature multimédia. Elles sont organisées de manière hiérarchique.
  2- Il a une certaine présentation, fonction de l'environnement dans lequel il se trouve (outil, utilisateurs).
  3- Il possède de la connaissance qui détermine son comportement face aux événements produits par l'environnement dans lequel il se trouve. Cette connaissance lui permet de gérer lui-même sa cohérence et son intégrité en déclenchant les actions nécessaires. On distingue la connaissance générale associée à sa structure interne et la connaissance plus spécifique liée à sa présentation sur un outil de restitution. Cette dernière détermine clairement les moyens d'actions de l'utilisateur sur le formulaire.

- Le formulaire est un **objet de la base de données** :
  D'objet de présentation, il devient objet d'application. Une application sera définie en termes de formulaires constituant non seulement son interface mais également le support des données qu'elle utilise et des traitements qu'elle réalise. Tous les formulaires de l'application sont stockés dans la base. Sur le poste, l'utilisateur manipule un formulaire. Il peut le remplir en utilisant des éditeurs particuliers, en indiquant que le contenu du champ est le résultat d'une recherche dans la base, présenté avec un certain gabarit. Le formulaire avec sa connaissance gère la cohérence des informations données, réalise des calculs, etc.

Partant de cette approche, nous voulons offrir un outil interactif sur un poste de travail relié (par un réseau) à un SGBD, et fournissant un mécanisme assez général pouvant être utilisé aussi bien au moment (1) de la conception d'une base de données pour décrire des objets particuliers formulaires devant être stockés et gérés par le SGBD, qu'au moment (2) de la conception du schéma externe d'une application pour décrire les propriétés structurelles, sémantiques et de présentation de ses objets.

Cet outil est basé sur le concept de **formulaires complexes**. Il répond à plusieurs exigences concernant leurs traitements. Ces exigences sont présentés dans le chapitre II. Nous y décrivons également les objectifs et les fonctionnalités généraux de notre outil : **GIF** (Gestionnaire Interactif de Formulaires). La suite de la thèse présente des commandes du noyau de GIF.

Pour illustrer notre présentation, nous supposons l'existence d'une base de données contenant une classe d'objets denommée PERSONNE. Chaque "personne" possède un numéro de sécurité sociale (no_ss), un nom, un prénom, une adresse, une fonction, un indice, un échelon, un salaire, un mode_paiement. On se propose de décrire les documents formulaires "Demande de mission" (Figure 1.16) et "Remboursement de frais" (Figure 1.17).

N° _____

MATHEMATIQUES APPLIQUEES – INFORMATIQUE
GRENOBLE

DEMANDE DE MISSION

Nom et prénom: _____

Adresse personnelle: _____

N° sécurité sociale: _____

Mode de paiement: _____

Grade: _____ **Echelon:** _____ **Indice:** _____

Date et heure de départ de Grenoble:

le _____ **à** _____

Date et heure de retour à Grenoble:

le _____ **à** _____

Si ces dates ont été modifiées, le signaler au retour.

Lieu et motif du déplacement: _____

_____

Imputation de la dépense: _____

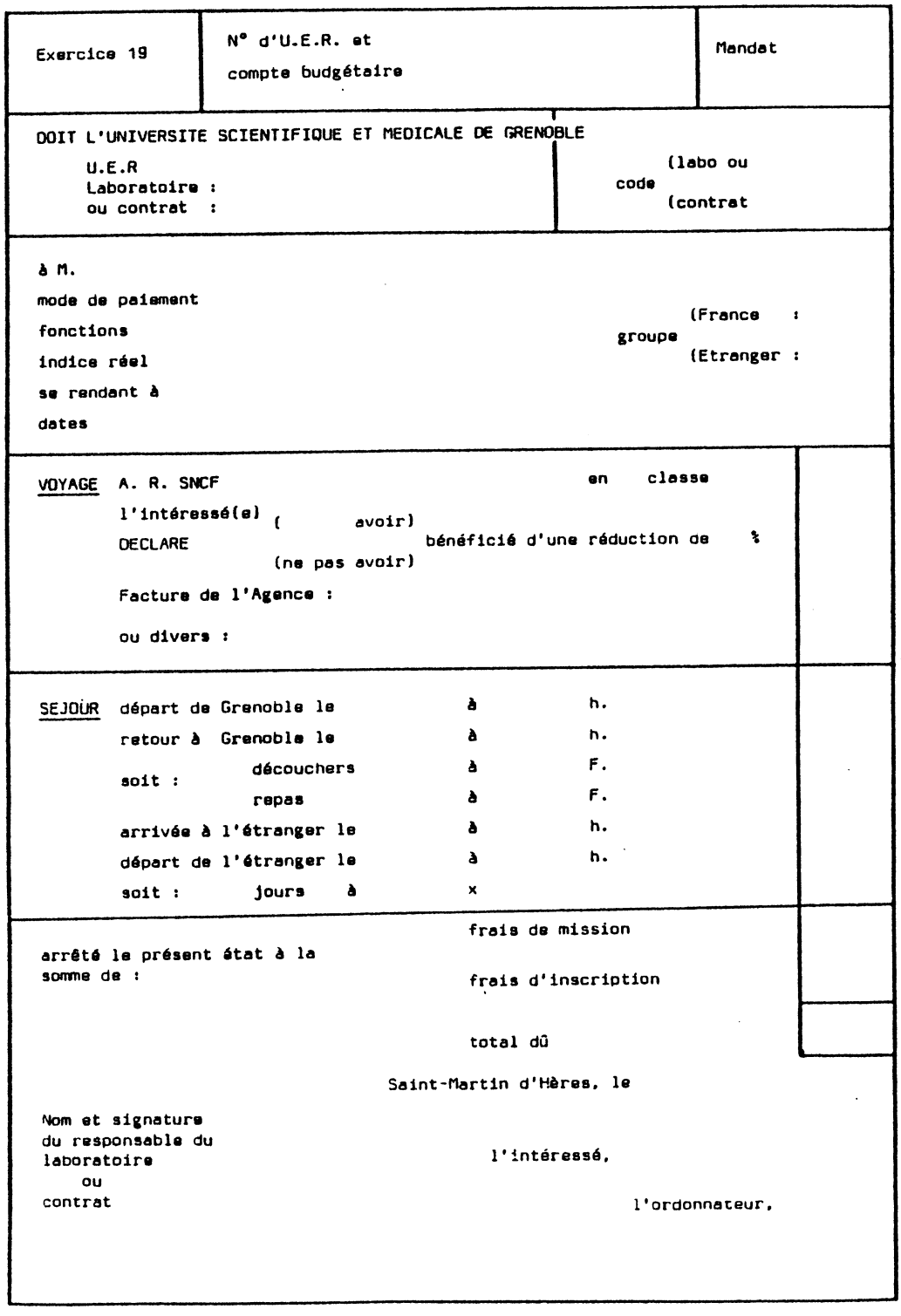
Visa du responsable de l'équipe de recherche avec mention des modalités éventuelles de remboursement: _____

_____

_____

Figure 1.16 : Le formulaire "Demande de mission"

Le formulaire "Demande de mission" est utilisé par une personne de l'université lorsqu'elle désire obtenir une autorisation pour une mission.



Exercice 19 | N° d'U.E.R. et compte budgétaire | Mandat

DOIT L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

U.E.R
Laboratoire : (labo ou code
ou contrat : (contrat

à M.
mode de paiement
fonctions
indice réel
se rendant à
dates

groupe (France :
(Etranger :

VOYAGE A. R. SNCF en classe
l'intéressé(e) (avoir) bénéficié d'une réduction de %
DECLARE (ne pas avoir)
Facture de l'Agence :
ou divers :

SEJOUR départ de Grenoble le à h.
retour à Grenoble le à h.
soit : découchers à F.
repas à F.
arrivée à l'étranger le à h.
départ de l'étranger le à h.
soit : jours à x

frais de mission
arrêté le présent état à la somme de :
frais d'inscription
total dû

Saint-Martin d'Hères, le

Nom et signature du responsable du laboratoire ou contrat

l'intéressé,

l'ordonnateur,

Figure 1.17 : Le formulaire "Remboursement de frais"

Le formulaire "Remboursement de frais" est utilisé par le personnel administratif pour calculer le montant des sommes dues à une personne ayant effectué une mission et pour établir le mandat correspondant. On propose également de fournir à l'utilisateur un formulaire facilitant la manipulation des "personnes" qui sont des professeurs (Figure 1.18).

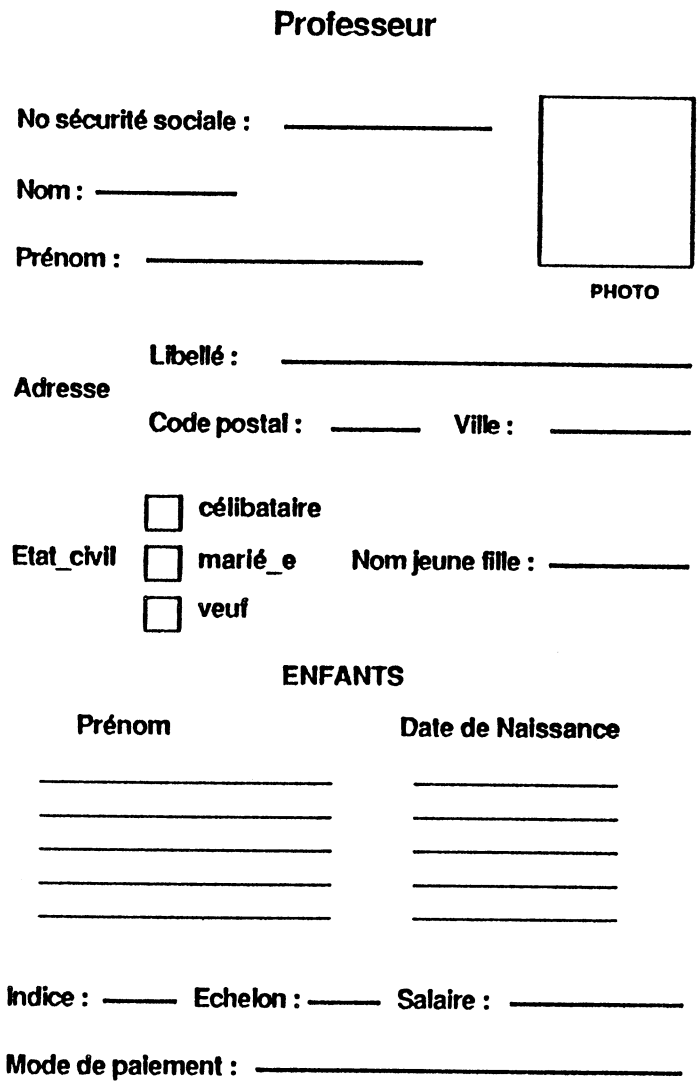


Figure 1.18 : Le formulaire "Professeur"

CHAPITRE II

FORMULAIRES COMPLEXES

Il a toujours fallu beaucoup plus d'imagination pour saisir la réalité que pour l'ignorer.

J. Giraudoux

CHAPITRE II

FORMULAIRES COMPLEXES

Ce chapitre présente les principaux besoins en description (section II.1) et en manipulation (section II.2) pour les formulaires complexes. Nous mettons en évidence les problèmes associés aux traitements des formulaires, du point de vue logique (interne) et non du point de vue "interface". Notre approche s'apparente donc à celle des modèles pour objets complexes, plutôt *orientés-objet*, où l'on cherche à décrire les objets dans leur totalié et à spécifier leur comportement (ou leur dynamique). Pour chacun des deux points traités, nous discutons des solutions offertes par divers travaux de recherche et de développement pour répondre à ce type d'exigence. Nous justifions ainsi les fonctions assignées à un système interactif de formulaires.

Dans la section II.3, nous présentons notre approche. Nous donnons le cadre dans lequel nous nous situons et les objectifs généraux. Nous précisons les limitations fixées pour la thèse.

II.1 Description des formulaires

II.1.1 Structure hiérarchique

D'un point de vue logique un formulaire peut être vu comme une composition de champs qui obeit à des règles précises (agrégation de champs, liste/ensemble de champs ou choix entre champs). Chaque composant ou élément du formulaire possède un nom. Dans le formulaire "Remboursement Frais" (Figure 1.17, page 47), on remarque de nombreux éléments qui sont un choix entre différents champs : *code*, *groupe*, *DECLARE*. Il est immédiat de constater que ce formulaire est une agrégation de 6 éléments (6 zones horizontales).

Pour la représentation interne d'un formulaire, il faut donc disposer d'un modèle de données offrant un formalisme adapté à la description de structures hiérarchiques.

Les composants élémentaires du formulaire ou champs sont décrits par un type qui détermine le domaine dans lequel ils prennent leurs valeurs. Les types de données nécessaires pour les formulaires sont :

- les types traditionellement pris en compte dans les langages de programmation et les modèles de données : entier, réel, chaîne de caractères, booléen, enregistrement, liste.

- de nouveaux types : texte, image, graphique, formule mathématique, programme, tableur, pour lesquels l'introduction dans les SGBD est encore du domaine de la recherche. Rares sont les langages qui offrent des structures de données pour ces types.

- des types spécialisés : une référence à une note jointe au formulaire, une date, une adresse, une croix, une signature ....

II.1.2 Connaissance attachée à la structure

Un formulaire possède un ensemble de propriétés sémantiques qui complètent sa structure déclarative. Les travaux de [FER 82, GEH 83] mettent en évidence bon nombre de propriétés sémantiques caractéristiques des formulaires.

Certaines propriétés sont inhérentes à la structure : conformité d'un formulaire à la structure utilisée pour le créer, conformité des valeurs de champs à leur type. Ceci implique que les fonctions codant les mécanismes de validation de types spécialisés soient connues. De manière explicite, on doit pouvoir décrire :

- L'unicité de valeur pour certains champs (clés). Un formulaire est en général identifié de manière unique par la valeur d'un ou de plusieurs de ses champs. Par exemple pour le formulaire "Demande de Mission", le champ N^o doit avoir une valeur unique.

- Une liste de valeurs possibles pour un champ :
  Dans la description de type d'un champ, on peut avoir une liste de valeurs explicites. Parfois cette liste ne peut pas être définie en extension mais plutôt en intention par une expression de recherche de données. Par exemple, la valeur du champ *imputation de la dépense* pour le formulaire "Demande de mission" doit appartenir à la liste des "postes" de dépenses (contrat, laboratoire, université).

- Le calcul de valeurs des champs :
  Les formulaires peuvent être liés entre eux ou avec d'autres objets de la base. Un champ de formulaire peut être décrit par une expression qui fait intervenir des champs de formulaires ou/et des objets de la base. On peut donc avoir des champs calculés et indépendants ou dépendants des objets utilisés pour leur calcul. Par exemple, pour le formulaire "Remboursement de Frais", la valeur de son champ *total dû* est obtenue par une expression de calcul (somme) sur les valeurs des champs *frais de mission* et *frais d'inscription*.

- Les politiques d'opérations pour les champs dépendants d'autres objets de la base. Les liens entre un formulaire et d'autres objets de la base, induits par des règles de calcul peuvent être **faibles** ou **forts** selon que les valeurs obtenues pour les champs sont stockées ou non dans le formulaire.

  Pour le formulaire "Demande de mission", les valeurs des champs *Adresse personnelle*, *Mode de paiement*, *Grade* et *Indice* sont obtenues par une expression de recherche de données dans la base sur l'objet PERSONNE. On peut choisir de ne pas stocker ces valeurs dans le formulaire (lien fort). Une modification de valeur du champ *Adresse Personnelle* doit obligatoirement être translatée sur la personne concernée.

  S'il existe un lien fort, toute opération sur le formulaire est à translater sur ses objets sources. Si par contre le lien est faible, on parle de report d'opérations du formulaire sur les objets. La décision de reporter ou non est affaire de sémantique explicite qui devra être donnée par le concepteur.

- Les conditions devant être vérifiées par les valeurs des champs du formulaire pour pouvoir exister. Par exemple : *Date et heure de départ* précède *Date et heure de retour*, une valeur de *réduction* existe si l'utilisateur *DECLARE avoir bénéficié d'une réduction* (formulaire "Demande de mission").

- Les actions sur des éléments de formulaires ou sur d'autres objets de la base à effectuer sous certaines conditions. Par exemple on doit décrire : *"si la personne demandant une mission n'existe pas encore dans la base, alors la créer"*. Cette action pourra être réalisée en utilisant par exemple un formulaire pour la mise à jour de l'objet PERSONNE.

  On pourra ainsi rendre obligatoire l'utilisation de certains champs, activer un nouveau formulaire, effectuer des traitements sur des objets parallèlement à l'utilisation du formulaire.

Les quelques exemples illustrant nos propos montrent que l'expression de la connaissance fait intervenir à la fois des opérations de manipulation d'objets de la base (au sens large), des opérateurs de calcul (fonctions prédéfinies sur les types de données possibles pour les champs), et des fonctions de programmation (si alors sinon, ..).

II.1.3 Présentations

Chaque utilisateur d'un formulaire a sa propre vision de cet objet, liée à l'utilisation qu'il en fait (par exemple le remplir, ou bien l'écouter et le signer, l'imprimer) et à ses droits (une partie de l'information peut lui être cachée). La structure d'un formulaire doit pouvoir être présentée de différente manière selon l'outil de restitution utilisé (formulaire affiché, papier, vocal, message) mais aussi selon l'utilisateur auquel il est destiné. Chaque formulaire obtenu par association de la structure avec une présentation est donc destiné à un groupe d'utilisateurs sur un outil particulier de restitution. Par exemple, il n'est pas nécessaire de présenter les champs *Imputation* et *Visa* à l'utilisateur demandant une mission.

Nous avons vu également qu'un champ pouvait avoir un type dit spécialisé dont la représentation interne doit être définie. Si l'on veut autoriser l'utilisateur final à manipuler interactivement ce champ, il faut lui donner une représentation externe (par exemple quel est l'équivalent électronique à ce niveau de la croix sur papier).

L'outil que nous voulons est interactif. La présentation des formulaires sur un écran n'est pas trivial pour diverses raisons :

- L'utilisation possible de types de données multimédia dans les champs d'un formulaire implique des environnements d'édition appropriés. L'utilisation d'un formulaire va donc nécessiter l'intégration d'outils qui présentent bien souvent des conventions et des formalismes différents. Ce qui risque de désorienter l'utilisateur. Pour cela il faut une méthode de présentation qui coordonne les outils. Il faut en particulier que les données puissent circuler entre le formulaire et les outils.

- Le propre du formulaire est de supporter un grand nombre d'informations explicatives ou fournies (contrat de location, feuilles de déclaration d'impôts). La méthode de présentation doit tenir compte des limitations physiques des outils de restitution mais aussi des facultés de perception de l'utilisateur. On

doit pouvoir présenter le formulaire sur différentes pages d'écran, utiliser des icônes pour présenter le contenu de certain champ (texte, image). On veut utiliser au maximum la souris : Pour remplir un formulaire, l'usager va cliquer sur un élément et lui donner une valeur. Il peut déplacer la souris comme il veut. Dans certaines applications, il doit être possible de figer le mode d'utilisation du formulaire : l'utilisateur ne peut plus traiter les champs dans l'ordre qu'il choisi mais cet ordre lui est imposé.

- La présentation du formulaire reflète très souvent sa structure logique : chaque champ a sa zone, les champs et les éléments (composition sur des éléments ou des champs) sont regroupés par zone. Chaque zone a des caractéristiques particulières : dimension, forme, couleur, ... On peut remarquer que les activités bureautiques sur les formulaires sont essentiellement guidées par la mémoire de l'utilisateur. Elles réfèrent des connaissances générales et connues depuis longtemps (mettre de l'information dans une zone, mettre une croix, retourner le formulaire à son expéditeur, ...) et des connaissances plus particulières liées à sa forme, sa couleur, son logo, ... On manipule donc des formulaires aussi par le biais de leurs aspects externes principalement pour ce qui est de la recherche (recherche de formulaires possèdant un certain logo). Il convient donc d'offrir une méthode de représentation externe du formulaire qui soit exploitable pour effectuer certaines actions. Pour les utilisateurs de type "expert", il convient d'offrir une méthode de présentation évolutive qui leur permet par exemple de personnaliser la présentation de leur formulaire (déplacement des zones, ajout de logo, ...).

La distinction entre formulaires du même type (des demandes de mission, des remboursements ...) ne se fera plus seulement par les données qu'ils contiennent mais également par le biais de leurs présentations particulières.

Connaissance et présentation

La notion de présentation ne peut pas être dissociée de celle de représentation de la connaisssance pour l'utilisateur. En effet, nous avons dit que le comportement du formulaire est essentiellement guidé par ses propriétés sémantiques. Ce comportement va dépendre de la connaissance attachée à sa structure. Mais cette connaissance ne pourra pas toujours être exploitée correctement.

En effet, la méthode de présentation permet de restituer, pour certains utilisateurs, une partie seulement de l'information contenue dans la structure. Lorsque le formulaire restitué n'est pas en correspondance biunivoque (1-1) avec la structure sous-jacente, l'exploitation de la connaissance peut poser des problèmes : comment valider une contrainte entre deux champs C1, C2 si l'un des deux champs n'est pas présent sur l'écran ?. Il faudra alors interdire certaines actions aux usagers, demander que les actions soient réalisées dans un certain ordre, etc.

On doit donc pour chaque représentation externe d'une structure de formulaire préciser les conditions d'applicabilité de la connaissance. De cette manière, le formulaire aura un comportement approprié à sa présentation. On attachera donc à chaque présentation une connaissance plus spécifique déterminant clairement les moyens d'actions de l'utilisateur sur le formulaire (externe).

En conclusion, une présentation détermine un environnement de travail particulier (médias + utilisateurs). Dans chaque environnement, le formulaire a un comportement particulier.

II.1.4 La dynamicité

La conception d'un formulaire est toujours orientée vers un besoin de satisfaire la communication d'informations de manière structurée, intelligible et cohérente. Dans le domaine de la communication, les restructurations sont "perpétuelles" : besoins plus grand en informations, informations plus précises. On ne peut donc pas concevoir l'automatisation de formulaires sans proposer des mécanismes pour leur restructuration possible, leur changement de présentation et de comportement. L'outil formulaire doit réaliser le concept de dynamicité des structures, des connaissances et des présentations.

II.1.5 - Discussion

1- Structure de données pour le formulaire

Il est clair aujourd'hui que le modèle relationnel est mal adapté pour la description des nouvelles applications des bases de données manipulant des objets souvent volumineux avec des structures complexes (hiérarchiques, cycliques), des informations multimédia (texte, image, graphique, voix).

Un premier niveau de hiérarchisation des données est offert par les modèles N1NF qui ont abandonné la condition de "première forme normale" du modèle relationnel (toute valeur d'attribut doit être atomique). On peut citer : le modèle NF2 [JAE 82, SCH 82], le modèle de Fisher et Thomas [FIS 83], le modèle V-relationnel pour le projet VERSO [BAN 82, ABI 84]. Le modèle de Roth, Korth et Silberschatz [ROT 85a, ROT 86]. Ces modèles autorisent une relation comme valeur d'attribut. Une relation non normalisée est obtenue en utilisant un constructeur de n-uplets et un constructeur d'ensembles (relations). On impose dans ces modèles l'alternance entre constructeur de n-uplets et constructeur d'ensembles. Des extensions visent à introduire de nouveaux constructeurs, à offrir la possibilité d'avoir des n-uplets de n-uplets ou bien des ensembles d'ensembles. Le modèle NF2 étendu [PIS 86] propose un constructeur de listes. Dans [ROT 85b], on considère les valeurs nulles dans les relations N1NF. Dans [ABI 87], on envisage des extensions pour gérer des structures cycliques et des ensembles hétérogènes.

Les modèles de données sémantiques offrent les concepts de classification, d'agrégation et de généralisation qui permettent de représenter des structures complexes. La plupart de ces modèles sont des extensions du modèle Entity-Relationship [CHE 76, COD 79, BRO 81, VEL 84, GIB 84, CHR 85, PAR 87] ou bien des modèles orientés objet [NEU 87] qui ont leur origine dans les langages de programmation objet comme SMALLTALK [GOL 83] ou les frames [MIN 75]. Pour ces derniers, l'entité est un objet et la relation entre entités est un objet de niveau supérieur.

Le modèle sémantique TIGRE [VEL 84] a été conçu pour la modélisation d'application manipulant des données multimédia. C'est une extension du modèle E-R avec la notion de type, la possibilité de donner des attributs à une association et d'agréger des associations entre elles. Un schéma d'une base TIGRE est une collection de types définis au moyen de constructeurs particuliers. Le constructeur **document** y est utilisé pour obtenir des structures de données hiérarchiques permettant de voir et de traiter un document dans sa totalité. Les opérateurs (sous-constructeurs) proposés pour la définition d'un type document sont : liste, mise en séquence et choix.

2- Représentation des connaissances

Les connaissances devant être attachées aux formulaires sont caractérisées en bases de données par le terme "contraintes d'intégrité sémantique".

Pour représenter les connaissances, on peut alors s'inspirer des approches suivies dans les SGBD pour la gestion des contraintes sur les données.

- L'approche "type abstrait" consiste à définir pour chaque structure de formulaire des opérations individuelles qui sont des programmes codant la connaissance qui lui est attachée. Cette approche présente l'inconvénient majeur d'avoir à gérer des programmes et d'avoir à les recompiler pour chaque modification de structure ou de sémantique (connaissance) du formulaire. De plus, comme un formulaire (du point de vue externe) n'est pas toujours le reflet de sa structure, on risque de se retrouver avec des programmes qui ne peuvent s'exécuter correctement. Une première solution consiste alors à introduire dans ces programmes tous les cas de figures liées aux diverses présentations possibles de la structure. Toute modification de structure implique un changement dans les programmes et leur recompilation. Une seconde solution est d'associer à chaque couple (structure, présentation) un ensemble d'opérations individuelles [COL 85]. Cette approche "type abstrait" pour le formulaire cache la structure de l'objet lui-même et on retrouve de la connaissance dupliquée.

- L'approche "trigger" permet la description d'un groupe d'actions lorsqu'une condition est vérifiée. Un "trigger" est activé (ses actions sont exécutées) chaque fois que la condition est vérifiée. C'est ce type d'approche que l'on rencontre dans les systèmes orientés formulaires [TSI 80, SHU 82, ZLO 82]. Elle présente l'avantage d'autoriser une modification plus aisée de la connaissance et une utilisation de cette connaissance indépendamment des présentations du formulaire. Ceci autorise des traitements sur les formulaires sans avoir à passer par l'interface utilisateur. On doit remarquer que la mise en oeuvre du mécanisme de "triggers" sous cette forme n'est pas facile puisque le système doit évaluer systématiquement toutes les conditions chaque fois que se produit une opération sur le formulaire. Il faut donc pouvoir décrire explicitement quand les conditions doivent être évaluées et les actions exécutées. L'extension de la notion de trigger proposée dans [GIB 84] va dans ce sens. Il définit un "trigger" par un "pattern" qui identifie l'opération pour son activation, une condition à vérifier pour accepter l'opération et une liste d'actions à exécuter après l'opération.

  Cette approche est intéressante car elle permet de modéliser un événement (notion duale de celle d'opération), ce qui dans le cas des formulaires

interactifs est primordial. En effet lorsqu'on a, par exemple, à remplir un formulaire la difficulté n'est pas de donner une information mais de savoir quand on doit la donner : - quand un événement (opération sur un champ du formulaire, sur un autre formulaire, sur la base, ...) s'est produit, - quand une condition portant sur d'autres informations appartenant au formulaire, aux objets de la base, aux utilisateurs, au système est vérifiée. Pour être appliqué dans le cadre d'un outil formulaires, le mécanisme de "trigger" doit être généralisé. Il faut pouvoir décrire un "pattern" identifiant une conjonction ou une disjonction d'événements ou une combinaison des deux. La partie condition doit pouvoir contenir une liste de conditions à évaluer. Pour l'expression des actions à exécuter, on ne peut pas se contenter d'utiliser le langage de manipulation de données du SGBD sous-jacent, il faut pouvoir exprimer des calculs, des expressions conditionnelles ou itératives qui ne se rencontrent pas dans ce type de langage.

- L'approche modèle de représentation de la connaissance de L'IA (Intelligence Artificielle). L'intêret actuel pour l'IA et en particulier pour l'utilisation de ses techniques dans le domaine des bases de données a abouti tout naturellement à s'intéresser aux règles de production comme support pour la vérification des contraintes d'intégrité. On rappelle qu'une règle de production a une précondition. Si celle-ci est satisfaite, la règle peut être activée. Ce qui entraîne obligatoirement un changement dans la structure de données de l'application. Cette approche necessite un système de contrôle qui choisit les règles à appliquer pour satisfaire un but (condition globale). Choisir une représentation de la connaissance attachée à la structure du formulaire sous forme de règles de production implique donc des techniques de mise en oeuvre et de contrôle du raisonnement.

  Une orientation dans ce domaine est l'utilisation de systèmes de programmation en logique (PROLOG, LISP) pour offrir aux SGBD un mécanisme de déduction et de vérification de contraintes sémantiques (on peut citer par exemple SYSLOG [OLI 85] pour TIGRE). Mais les services d'interface des interpréteurs des langages utilisés (LISP, PROLOG) avec d'autres langages disons procéduraux (support traditionnel pour l'implantation des SGBD) sont encore pauvres et ne facilitent pas le couplage SGBD - Systèmes logiques.

  Les idées développées autour des langages objets (Smalltak [GOL 83]) sont intéressantes. A chaque classe (d'objets) sont associées - des *methods* décri-

vant comment doivent être réalisées les opérations sur les objets - des *messages* pour la communication entre classes. Les objets héritent des méthodes et messages de la classe dont ils dépendent et disposent de leur propre mémoire privée. Le comportement de l'objet est donc codé par des *methods*. Les caractéristiques de ces langages sont proches de celles proposées pour les nouveaux modèles de données *orientés-objet* [GIB 84, RIC 87, NEU 87, TSI 87]. Un autre mécanisme courament utilisé en IA pour représenter la connaissance est le **frame** [MIN 75, FIK 85]. Les aspects sémantiques d'un objet sont décrits par des attachements procéduraux aux propriétés (*slots*) de l'objet [BEN 85].

On tient à faire remarquer que ces mécanismes de représentation de la connaissance restent néanmoins mal adaptés à la construction de systèmes manipulant des objets fortement typés (parce qu'on veut les contrôler) et volumineux en taille et en nombre. L'utilisation des langages objets, comme support pour le développement des futurs SGBD, fait actuellement l'objet de recherches.

3- Présentation

Les SGBD actuels proposent des mécanismes encore trop frustres pour la définition des présentations externes de leurs objets. La définition d'une présentation est bien souvent maintenue à l'extérieur de la base. Pour les applications bureautiques nous pensons que la notion de présentation de l'objet ne doit pas être négligée. Elle seule fait apparaître un objet électronique stocké dans une base comme un objet du bureau.

4- Dynamicité

Dans la plupart des SGBD traditionnels, toute altération du schéma de la base entraîne sa recompilation. Les SGBD relationnels réalisent le concept de dynamicité des schémas (ajout, suppression d'un attribut de relation). Pour les formulaires, on peut proposer des commandes pour la modification de sa structure, des connaissances et des présentations. Ces commandes seront réservées aux concepteurs des applications.

II.2 Gestion et Utilisation des formulaires.

II.2.1 Les opérations

La gestion d'un formulaire consiste à fournir un ensemble d'opérations pour la recherche de "structures, connaissances et présentations" dans la base et leur manipulation (insertion, suppression et modification). Il faut en plus des opérations de copie, renommage, envoi. Pour la recherche de données, il faut une méthode pour retrouver un ensemble de formulaires et en selectionner un dans cet ensemble. Cette recherche peut nécessiter des opérateurs tel que l'union, l'intersection, la différence, la restructuration (reformatage),..sur des formulaires. Il faut en plus une opération pour l'association d'une présentation avec une structure, une opération pour montrer le résultat de cette association (réalisation sur un média), une opération pour détruire une réalisation. Il faut également une opération pour la recherche de formulaires vérifiant certaines conditions de présentation.

II.2.2 Gestion de la connaissance

L'utilisation d'un formulaire se fait via les moyens d'actions dont il dispose. En effet, un formulaire possède la connaissance qui détermine son comportement. Elle permet de décrire comment un champ est valué par d'autres objets ainsi que l'ensemble des comportements qu'il peut avoir face aux événements dans divers environnements. A l'utilisation du formulaire, il faut pouvoir exploiter cette connaissance. On doit offrir des mécanismes permettant d'évaluer des expressions de calcul, de déclencher des actions et de gérer des dépendances formulaires-objets (mécanisme du type composition, translation et report d'informations entre formulaires et objets).

II.2.3 Gestion des données

Pour la base de données, le formulaire est esssentiellement un ensemble de données multimédia vérifiant une structure. Il faut que le SGBD puisse supporter :

- **des formulaires contenant des valeurs nulles "inapplicables" et des valeurs nulles "inconnues".**

  Dans la réalité, on utilise très souvent le même formulaire comme support d'informations pour différentes tâches. De ce fait, selon son utilisation, certains champs ne sont pas significatifs. Pour une tâche, on peut donc ne pas considérer ou ne pas **appliquer** un élément de formulaire (champ ou composition de champs). Si un élément n'est pas applicable pour une tâche de

saisie de données, il ne doit pas exister de valeur pour cet élément et il contient alors une valeur nulle dite "**inapplicable**".

De plus pour une même tâche, un formulaire peut être utilisé par divers intervenants. Il se peut donc qu'à un moment de la vie de ce formulaire, on ne connaisse pas encore les valeurs de certains champs : les valeurs de ces champs existent (doivent exister) mais sont encore "**inconnues**". Le formulaire est alors dans un état d'incomplétude temporaire.

La gestion des valeurs nulles a fait l'objet de nombreuses recherches. Des solutions ont été proposées [COD 79, ZAN 83, TES 86]. Mais peu de systèmes supportent correctement la notion de valeurs nulles. On offrira un codage spécifique pour ces valeurs et on essayera autant que possible de prendre en compte ces valeurs dans les algorithmes réalisant les opérations sur les données de base (entier, réel, chaîne , etc).

Nous considérons également une valeur nulle particulière décrivant la **non-information** (la non-existence de valeur). Cette valeur est nécessaire si l'on veut pouvoir gérer des formulaires dont certains champs sont "vides" parce qu'il n'existe point de valeurs (dans le monde réel que nous considérons) qui puisse être valeurs pour ces champs. Bien sûr l'absence de valeurs doit être controlée : le formulaire possède de la connaissance lui permettant d'accepter ou de rejeter une valeur non-informative pour un champ.

- **des formulaires incohérents temporairement**

Un formulaire contenant des valeurs nulles "inconnues" est incomplet. Cette incomplétude peut entraîner une incohérence du formulaire : certaines propriétés sémantiques ne sont pas vérifiées parce qu'il manque des données. Cette incohérence doit pouvoir être acceptée. Elle est temporaire et disparaîtra dès le remplacement des valeurs "inconnues" par des valeurs explicites.

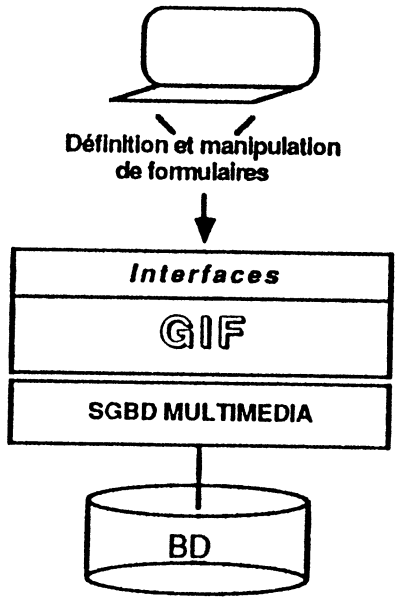
Dans d'autres cas de figures, le formulaire devient incohérent parce que certaines données ne respectent pas sa sémantique. Ce peut être une incohérence momentanée (une valeur "inconnue" pour une donnée obligatoire) ou liée à un cas non prévu. On parle alors d'**exceptions** [BOR 85, ESC 87]. Si l'on veut pouvoir gérer ces cas d'exceptions, il faut en premier lieu pouvoir les reconnaître. Ceci ne pourra se faire que si le formulaire dispose de la connaissance adéquate.

II.3 GIF

Les exigences auxquelles doit répondre un outil interactif pour la définition et la manipulation de formulaires complexes dans un environnement bases de données ont été mises en évidence. Dans cette section, nous présentons **GIF** (Gestionnaire Intéractif de Formulaires). Nous précisons le cadre de travail et les objectifs que nous nous sommes fixés pour cette étude (section II.3.1). Nous introduisons ensuite les éléments du modèle choisi pour développer GIF.

II.3.1 Cadre de l'étude

Nous considérons que l'utilisateur de GIF dispose d'un poste de travail évolué (écran bit-map, souris, et sortie vocale) avec un logiciel de gestion de fenêtres et de menus, des primitives de gestion d'événements, des éditeurs pour le texte, le graphique, l'image, ... Ce poste est relié par un réseau local à un serveur de données. Nous avons choisi de nous intéresser beaucoup plus à l'aspect organisation logique et à la dynamique des données pour un formulaire qu'aux mécanismes pour la gestion de données multimédia. Nous considérons donc que le serveur, comme par exemple le SGBD TIGRE [VEL 84], est capable de gérer et de stocker des informations de nature alphanumérique classique et de nature multimédia (texte, image, graphique, programmes, ...).



Objectif

Pour **GIF**, on vise essentiellement à élaborer des mécanismes internes adaptés à la description, la gestion et l'utilisation de formulaires complexes. Le noyau de GIF fournit donc un ensemble d'opérations de définition et de manipulation de données, constituant un niveau de traitement logique. GIF utilise un modèle de formulaire que nous présentons succintement dans la section suivante. Ce modèle répond aux exigences mentionnées dans les sections précédentes.

Nous ne nous préoccupons pas des mécanismes d'"Interfaces" : contrôleur de saisie, affichage du formulaire dans une fenêtre, répercussion de toute manipulation des structures de données internes à GIF sur le formulaire affiché et inversement, gestion des outils d'édition, etc.

Les données sur le poste proviennent du serveur. Seul le serveur a accès aux données physiques. Nous supposons disposer d'une couche de présentation des opérations du serveur qui traduit les manipulations faites dans GIF en termes d'opérations sur le serveur de données.

II.3.2 Introduction au modèle de formulaires

Le modèle de formulaires comprend un ensemble d'éléments que nous introduisons ici. Nous avons choisi de les classer sous les rubriques objets et opérations. La figure 2.1 donne une vue d'ensemble des concepts les plus importants du modèle.

II.3.2.1 Objets

Type spécialisé

Pour les formulaires, nous avons montré le besoin de traiter des données d'un type particulier lié à l'application. Nous avons choisi de traiter ces types, qualifiés de spécialisés comme des types abstraits ou domaines [GIB 84]. La définition d'un domaine particulier consiste à spécifier sa représentation interne et ses représentations externes possibles, les fonctions de transformation entre les deux formes de

représentation, les fonctions sémantiques pour la validation d'une valeur du domaine, les fonctions de manipulation.

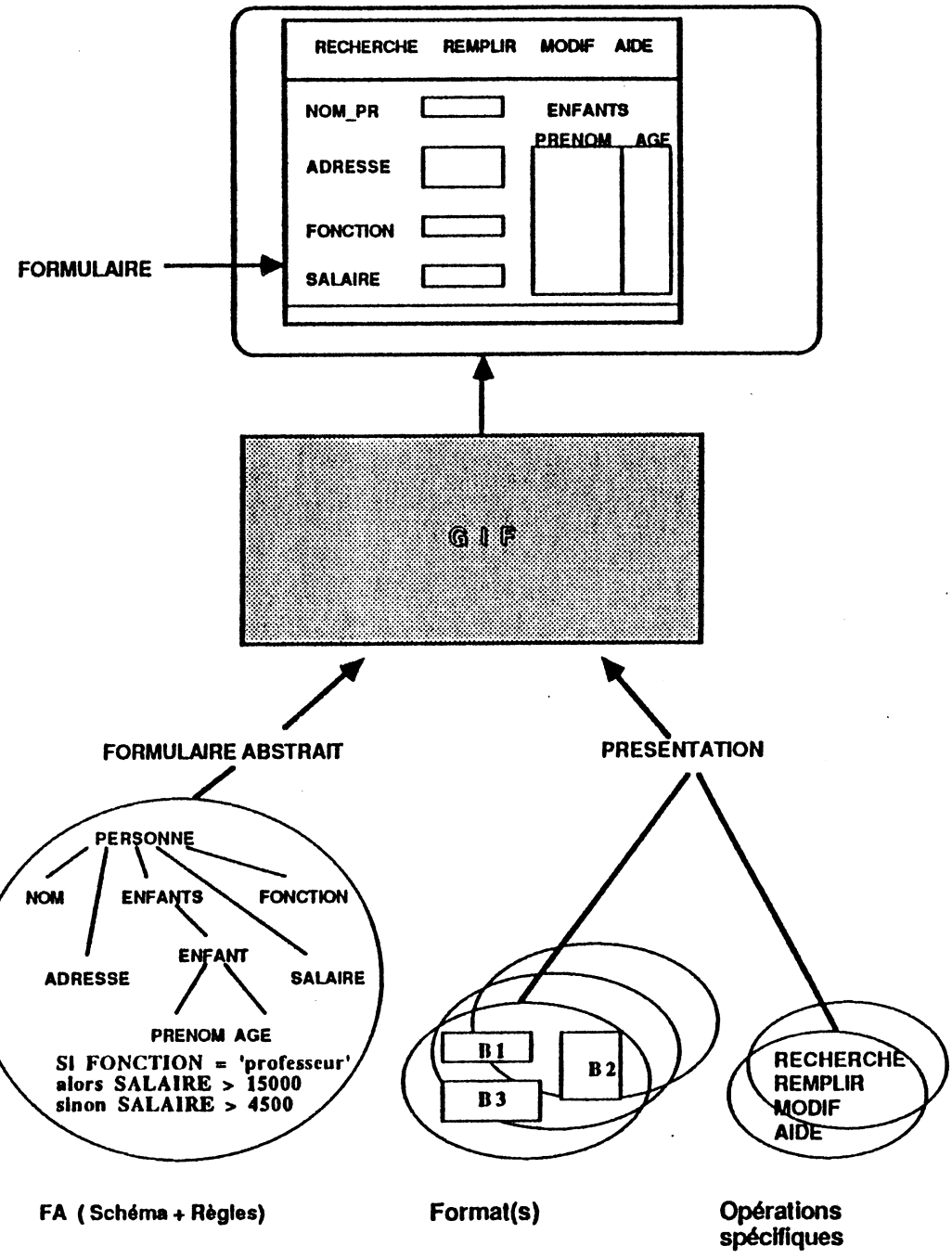


Figure 2.1 : Concepts du modèle de formulaires complexes

Formulaire Abstrait (FA)

Une classe de formulaires (des commandes, des déclarations d'impôts) présentant des propriétés structurelles et sémantiques semblables est modélisée par un **Formulaire Abstrait**. Un formulaire abstrait possède un **schéma** qui modélise la structure hiérarchique du formulaire, des **règles** qui modélisent les divers aspects de la connaissance attachée à cette structure et une **valeur** (des occurrences).

Un schéma sera obtenu en utilisant les constructeurs : agrégat (mise en séquence), choix, liste, ensemble et vue. Les trois premiers constructeurs correspondent au modèle de document TIGRE [BRV 83]. Le constructeur "vue" a été introduit pour autoriser un élément de formulaire à être une vue sur la base. Nous verrons qu'il est important puisqu'il permet d'introduire dans le modèle : la généralisation (spécialisation, l'intersection et l'union) ainsi que la possibilité d'avoir des structures avec cycles. Nous rendons le modèle de formulaire orthogonnal en autorisant la définition de listes de listes (ce qui n'est pas fait dans le modèle de document TIGRE) et d'ensembles d'ensembles.

Les règles permettent d'associer aux éléments du schéma des fonctions complexes, d'assurer l'intégrité et la cohérence du formulaire et de déclencher des actions lors de l'échec ou la réussite de certains événements ou de certaines conditions. Une règle sera définie en utilisant les opérations du modèle (voir section II.3.2.2).

Nous avons choisi de rassembler dans un même objet (le FA) les divers aspects de la connaissance concernant un formulaire. En ce sens nous avons opté pour une représentation *orientée objet*. Le concept de règle vient enrichir celui de schéma. Ce concept est à rapprocher de celui de "facettes" [BEN 85] ou de ceux de "methods ou active values" dans les systèmes basés sur les frames [FIK 85].

Une valeur de formulaire abstrait est un ensemble d'occurrences. Une occurrence est construite à partir du schéma du FA. Elle peut être vue comme une structure "dérivée" du schéma du FA dont les feuilles contiennent des valeurs (éventuellement nulles) de nature alphanumérique ou multimédia.

Lorsqu'on doit considérer un FA d'un point de vue sémantique, on doit distinguer sa signification de sa valeur. On utilisera :

1- **SCHEMA_FA** pour désigner l'intention ou la signification d'un formulaire abstrait, c'est à dire les composantes **schéma** et **règles**

2- **FA** pour désigner sa valeur.

Les notions de SCHEMA_FA et FA cohabitent en permanence. Par abus de langage et comme on le fait pour le modèle relationnel, le mot FA sera utilisé pour désigner aussi bien une valeur de formulaire abstrait que le schéma dynamique asssocié.

Format (FT)

Pour la présentation externe du formulaire, nous avons le constructeur **Format** qui permet de décrire comment doit être restitué un FA sur un outil particulier. Dans cette thèse, nous considérons uniquement l'outil écran.

Un format reflète donc tout ou partie du schéma de FA. Il se comporte comme un type. Une occurrence de format reflète tout ou partie d'une occurrence particulière de FA. Par abus de langage, on assimile format (type) avec occurrence de format. Sa définition correspond à décrire un agencement de boîtes caractérisées chacune par des valeurs particulières d'attributs (position, contours, contenu, ...).

Pour une occurrence particulière de FA, les valeurs des attributs peuvent évoluées. En effet l'utilisation d'un formulaire peut également consister à apporter des éléments nouveaux à son aspect externe. C'est donc en permettant de nouvelles valeurs pour les attributs caractérisant les boîtes d'un format (type) que l'on autorise la personnalisation des formulaires.

Opération spécifique (OPS)

Un formulaire affiché (FA associé à un format) a pour chaque catégorie d'utilisateurs un comportement particulier qu'il faut préciser. Le concept **d'opérations spécifiques** est utilisé pour modéliser ce comportement. De cette manière, on précise les moyens d'actions des utilisateurs sur le formulaire. Toute utilisation de formulaire se fera en activant une opération spécifique.

La méthode choisie pour décrire une opération spécifique repose sur la notion de schéma de comportement. Dans ce schéma, on précise le nom de l'opération, le nom du format pour lequel elle est définie, les utilisateurs autorisés à l'activer, les paramètres de l'opération, les objets locaux à l'opération et l'agenda des actions

réalisables par ses utilisateurs. Ces actions sont liées entre elles par des structures de contrôles correspondant aux constructeurs utilisés dans la définition d'un schéma de FA.

Un schéma de comportement définit de la connaissance qui devra être interprétée au moment de l'activation de l'opération. Il peut signifier que GIF **attend** ou **force** la réalisation des actions déclarées. On contrôle ou on dirige le comportement de l'utilisateur vis à vis du formulaire. Il faut préciser dans la définition d'une opération spécifique des points de contrôle permettant de savoir si les actions décrites entre ces points sont déclenchées par l'utilisateur ou par le système.

II.3.2.2 Les opérations

Pour les opérations du modèle, il convient de faire une distinction entre :

- les opérations pour la définition des objets présentés ci-dessus et la mise à jour des définitions.
- les opérations pour la manipulation des objets et l'utilisation des opérations spécifiques.
- les opérations pour la gestion des transactions.

1- Définition des objets

Les opérations pour la définition des objets correspondent à l'ensemble des commandes proposées au concepteur d'une application formulaires. Elles sont présentées dans cette thèse et débutent toutes par le mot clé **DEF**. Pour la mise à jour des définitions, on propose les commandes débutant par les mots clés : **ADD** (ajout), **MOD** (modification) ou **DEL** (suppression) de définitions. Ces commandes donnent à GIF son caractère évolutif.

2- Les opérations de manipulation

Par opérations de manipulation, on entend les opérations pour la manipulation des FA (plus précisement d'occurrences de FA), des formats et des opérations spécifiques.

Pour le FA

Nous avons les opérations pour l'expression de requêtes sur les données et les opérations de mise à jour. Ces opérations sont appelées **FA_opérations.**

Pour l'expression de requêtes, nous avons les opérations : **sélection, union, intersection, différence, produit, renommer, élaguer, grouper, éclater, ordonner** qui sont à rapprocher des opérations des algèbres N1NF.

INTERROGATION	SIGNIFICATION
sélection	sélection d'occurrences de FA
union	union de deux FA
intersection	intersection de deux FA
différence	différence de deux FA
renommer	renommage de FA et/ou de ses éléments
produit	produit cartésien de deux FA
élaguer	projection sur un FA
grouper	groupement de données
éclater	éclatement de données
ordonner	ordonner des données

La mise à jour d'un FA se fait par les opérations : **créer, modifier, détruire, affecter, supprimer.**

MISE A JOUR	SIGNIFICATION
créer	création d'une occurrence de FA
modifier	modification d'une occurrence de FA
détruire	destruction d'une occurrence de FA
affecter	affectation de valeur à un élément d'occurrence
supprimer	suppression d'une valeur d'élément d'occurrence

Le pouvoir d'expression des FA_opérations est augmenté en offrant la possibilité de les combiner avec : (1) des opérations sur les différents types de données de base (texte, temps, liste) et (2) des opérations représentant les structures de contrôles des langages de programmation. On obtient alors des **FA_expressions.**

Pour la Présentation : FT + OPS

Nous avons regroupé sous la notion de **Présentation**, les éléments utiles à la présentation externe d'un FA : le format et les opérations spécifiques associées au format. Pour un même FA, on peut trouver plusieurs présentations. Le résultat de l'association entre un FA et l'une de ses présentations est l'objet que nous appelons **formulaire**.

On propose les opérations : **associer** qui associe un FA à une présentation. Le résultat de cette opération est une représentation intermédiaire du formulaire. Pour produire un formulaire sur écran, il faut appliquer à cette représentation l'opération **présenter**. Pour effacer un formulaire sur écran, on utilise l'opération **effacer**.

Une manipulation de formulaire consiste à **activer** une opération spécifique (OPS). Cette opération **activer** réalise l'interprétation du schéma de comportement donné par l'OPS. Ceci va permettre d'accepter ou de rejeter une action usager, ou encore de valider cette action en faisant appel au mécanisme de vérification des règles attachée au FA sous-jacent au Formulaire. A tout moment, il est possible d'**annuler** la réalisation d'une OPS.

3- Les opérations pour les transactions

Lorsqu'un utilisateur travaille avec GIF, il se trouve dans une session durant laquelle il effectue des manipulations de formulaires. Il peut donc utiliser des formulaires différents. Considérons qu'il décide d'utiliser le formulaire F. Sur F, il peut exécuter une opération spécifique, par exemple REMPLIR qui correspond à réaliser une création d'occurrence. Supposons que l'on soit en train de créer une occurrence. La connaissance attachée au FA (Règles) permet de réaliser les contrôles sémantiques au plus tôt. En fin de remplissage (activation d'une autre opération spécifique), GIF signale à l'utilisateur si sa création est un succès ou un échec. En cas d'échec, libre à lui de décider s'il continue à travailler avec l'occurrence ou s'il annule son opération. En cas d'annulation, il faut que GIF soit capable de défaire toute les actions réalisées. En cas de succès, l'occurrence est alors stockée dans l'espace de travail de GIF. Pour une prise en compte des manipulations (faites durant REMPLIR) au niveau de la base, il faut que l'utilisateur puisse valider son opération.

Supposons maintenant que depuis le début de son travail avec le formulaire F, l'utilisateur a activé plusieurs opérations spécifiques qui se sont déroulées avec succès et qui non pas été validées pour la base. L'utilisateur peut alors décider de valider ou d'annuler son travail.

On voit apparaître la nécessité d'avoir dans GIF, la notion de transaction à différents niveaux : la session de travail, le formulaire et l'opération spécifique. Il nous faut considérer des transactions imbriquées et des transactions de longue durée. Les mécanismes pour la gestion de ces types de transaction font encore l'objet de recherches. Pour GIF, nous ne développerons pas ces aspects, nous essayerons d'identifier les problèmes.

Nous considérons la notion de transaction pour une opération spécifique. A chaque opération spécifique correspond une transaction. Nous nous appuyons sur la notion de transaction proposée dans les SGBD relationnel et considérons les opérations **début_T** (début de transaction), **fin_T** (fin de transaction) et **annuler_T** (assassinat d'une transaction).

Nous devons également considéré une opération **valider** qui s'applique à une ou un groupe d'opérations spécifiques. Une opération spécifique qui se déroule avec succès, pourra être validée pour la base de données.

II.3.3 Limitations

Dans la suite, nous présentons les éléments du noyau de GIF. Le chapitre III présente le FA d'une manière informelle et sans ses aspects dynamiques. Le chapitre IV présente les FA_opérations et les FA_expressions. Le chapitre V présente les règles. Finalement le chapitre VI présente la notion de présentation : le format et les opérations spécifiques où nous déclarons des actions (utilisateurs) attendues par GIF.

La définition des types spécialisés n'est pas traitée. On dispose d'une méthode de définition de domaines [ANT 86] utilisable pour GIF.

Nous ne nous préoccupons pas des problèmes de partage et de concurrence d'accès aux informations. Un utilisateur de GIF dispose donc de toutes les données (de la base) à tout moment. Cette approche nous autorise à travailler dans un **contexte mono-utilisateur**. Notre étude devra être poursuivie dans une optique de partages de données dans un environnement réparti : plusieurs postes de travail

connectés au réseau. On devra en particulier définir les protocoles d'échange entre les postes.

Nous nous plaçons également dans un contexte **mono-formulaire**. A un moment donné sur le poste, on utilise un formulaire. Pour les aspects transactionnels, il nous suffit donc de considérer seulement les transactions au niveau des opérations spécifiques. Toute opération spécifique correspond à une transaction GIF. Nous examinerons le cas des transactions de mise à jour d'un formulaire qui encapsulent des opérations sur une occurrence de FA (créer, modifier, détruire, affecter, supprimer, etc). Nous laissons de côté le problème de la validation globale d'opérations spécifiques (validation d'un groupe d'opérations). Nous considérons qu'une opération spécifique qui s'est déroulé avec succès est implicitement validée par GIF. Pour la base de données, GIF peut être vue comme une zone de travail "tampon". Une validation d'une opération correspond alors à réaliser de manière effective sur la base, les mises à jour faites durant cette opération (transaction).

CHAPITRE III

LE FORMULAIRE ABSTRAIT

Caminante no hay camino, se hace camino al andar[1]

Machado

[1] *Il faut accepter de cheminer sans chemin, de faire le chemin dans le cheminement.*

CHAPITRE III

LE FORMULAIRE ABSTRAIT (FA)

GIF a une vision de la base en termes de Formulaire Abstrait. Pour GIF tous les objets stockés dans la base sont des FA. Pour notre exemple, supposons que l'objet PERSONNE soit représenté par une relation. Dans la mesure où le modèle proposé est plus riche que le modèle relationnel, GIF verra cette relation comme un FA. Cette approche offre un niveau d'homogénisation pour les objets manipulés dans GIF, facilitant les spécifications et les manipulations du formulaire.

Un **formulaire abstrait (FA)** modélise une classe de formulaires (par exemple des Demandes de mission, des Remboursements de frais) qui ont des propriétés structurelles et sémantiques semblables. Un objet de la classe est une occurrence de FA. La définition d'un FA se fait avec la primitive suivante :

DEF_FA <schéma> [<règles>]

- <schéma> définit la structure de données des occurrences du FA,

- <règles> définit un ensemble de règles sur le schéma du formulaire abstrait. Elles expriment la connaissance qui lui est attachée.

Dans la suite de ce chapitre (section III.1), nous présentons de manière informelle comment est obtenu un **schéma** de formulaire abstrait sans aborder les aspects sémantiques (règles). Nous y reviendrons au chapitre V. Dans la section III.2, nous présentons les occurrences de FA. Nous discutons des propriétés d'une occurrence. En section III.3, nous expliquons le constructeur **vue** pour la définition d'un FA comme schéma externe d'une base de données. Nous précisons son comportement lors d'une manipulation du FA.

III.1 Le schéma de FA

III.1.1 Définition informelle

Un schéma de FA décrit une structure de données pour un formulaire. Cette structure est celle à partir de laquelle sont construites toutes les occurrences du FA. On peut qualifier un schéma de **simple** ou de **complexe**.

1- Schéma simple

Un schéma simple est décrit par son nom et le nom du domaine dans lequel les occurrences prennent leur valeur. On dispose des domaines suivants :

- de base : integer - domaine des entiers,
  string(X) - domaine des chaînes de caractères de longueur X,
  real - domaine des réels, bool - booléen,
  time - domaine du temps. On considère également les autres aspects du temps : temps restreint (par exemple time > day), intervalle, période, constant) [ADI 87].
- multimédia : text, image, graphic, voice, etc

Exemple 3.1 : **DEF_FA** message : text;

Remarque :

On peut également donner comme nom de domaine, le nom d'un domaine particulier, défini par ailleurs et décrivant un type spécialisé. Par exemple : argent (intervalle de valeur), signature, adresse, etc. Ce domaine sera défini avec le mécanisme proposé dans [ANT 86].

2- Schéma complexe

Un schéma complexe possède un nom auquel on associe une structure de données obtenue en appliquant des constructeurs à des schémas simples ou complexes. On dispose des constructeurs suivants :

- **agrégat**[1] : "begin S1; S2; ...;Sn end;" pour une mise en séquence des schémas S1, S2, ..., Sn.

- **choix**[1] : "case of S1; S2; ...; Sn; end;" pour un choix entre différents schémas S1, S2, ..., Sn.
- **liste**[1] : "list (min, max) of S;" pour une liste d'occurrences vérifiant toutes le schéma S. min et max sont les cardinalités minimale (>= 0) et maximale de la liste.
- **ensemble** : "set of S;" pour un ensemble d'occurrences vérifiant toutes le schéma S;
- **vue** : "view of <expression_vue>;" pour une vue sur la base de données.

Le constructeur **ensemble** a été introduit pour autoriser la modélisation d'ensemble de valeurs non ordonnées et non identiques, par opposition à une liste. Par analogie avec la notion de vue dans le modèle relationnel, nous proposons le constructeur **vue** qui permet la description d'une vue sur des FA. L'expression de la vue est un énoncé d'interrogation de FA. Toute évaluation d'énoncé d'interrogation de FA produit un FA. Ici comme pour la notion de vue du modèle relationnel, <expression_vue> n'est pas évaluée. Elle permet de déterminer la structure de données à associer au schéma "vue". En section III.3, nous montrons que ce constructeur permet d'utiliser le concept de FA pour décrire des schémas externes sur la Base de données. Nous discutons des liens existants entre les FA ainsi définis et les données de la base.

3- Notion d'élément

Un schéma est un arbre ou les noeuds intermédiaires et finaux sont désignés par des éléments. A tout élément (tout noeud) correspond un schéma de FA. Les éléments terminaux (**simples**) de l'arbre correspondent à ce que nous avons appelés jusqu'ici des champs. Les éléments intermédiaires correspondent à des FA avec un schéma complexe. Un élément intermédiaire peut être qualifié en fonction du constructeur utilisé pour définir son schéma. On a des éléments complexes : **agrégat**, **choix**, **liste**, **ensemble** ou **virtuel**. La racine de la hiérarchie est un élément particulier qui désigne le schéma du FA.

[1] Ces constructeurs correspondent à ceux proposés dans le modèle document TIGRE [BRV 83].

III.1.2 Exemples de schéma

Nous donnons ici la définition du schéma (complexe) pour le FA correspondant au formulaire "Demande de mission" (voir Figure 1.16 chapitre I ou annexe 1).

Exemple 3.2 :

```
DEF_FA demande_mission
begin
        key id_mission : integer endkey;
        no_secu : integer;
        nom_prénom : string(20);
        adresse : string(50);
        fonction : string(10);
        indice : integer;
        mode_paiement : string(40);
        lieu_déplacement : begin
                           ville : string(20);
                           pays : string(10);
                           end;
        motif : texte;
        date-mission : begin
                       début_mission : time;
                       retour_mission : time;
                       end;
        imputation : begin
                     type : case of laboratoire;
                                    contrat;
                            end;
                     ident_type : string(10);
                     code_type : integer;
                     end;
        groupe : string (15);
        signature_responsable : signature [nullallowed];
end;
```

Cet exemple montre que le formulaire abstrait "demande_mission" est de type agrégat (demande_mission : begin ... end;). Le champ *imputation* du formulaire "Demande de mission" a été modélisé par trois éléments ("type", "ident_type" et "code_type") agrégés dans l'élément "imputation" parce qu'il est nécessaire de différencier ces informations pour spécifier des actions à réaliser (voir chapitre V). Par contre les champs correspondant à *la date et heure de départ de Grenoble* sont modélisés par un seul élément simple de nom "début_mission" sur le domaine temps (time).

Exemple 3.3 : Définition du schéma de FA pour le formulaire "Remboursement de frais" (Figure 1.17, chapitre I ou annexe 1).

```
DEF_FA remboursement_frais
begin
        key id_frais : integer endkey;
        entête : begin
                  exercice : time;
                  no-uer : integer;
                  mandat : ident_code;
                  end;
        université : begin
                     identification : string (20);
                     code : ident_code;
                     end;
        envers : begin
                 nom : string(20);
                 paiement : string(30);
                 fonction : string(10);
                 indice : integer;
                 lieu_mission : string(20);
                 date_mission : time_interval;
                 groupe : begin
                          type_groupe : case of france; étranger; end;
                          valeur_groupe : case of 1; 2; 3; 4; 5; end;
                          end;
                 end;
        voyage : begin
                 train : begin
                         aller_retour : string(30);
                         classe : case of 1; 2; end;
                         prix_train : argent;
                         réduction : Case of
                                     oui : begin
                                           pourcentage : integer;
                                           montant : argent;
                                           end;
                                     non;
                                   end;
                         suppléments : list(0,5) of sup : begin
                                                          ident_sup : string(20);
                                                          prix_sup : argent;
                                                          end;
                         montant_supplément : argent;
                         end;
                 prix_avion : argent;
                 divers : begin
                          séjour_personnel: time_interval;
                          autre : text;
                          end;
                 end;
        séjour : begin
                 france : begin
                          départ_grenoble : time;
                          retour_grenoble : time;
                          nb_découchers : integer;
                          prix_découcher : argent;
                          montant_découcher : argent;
```

```
                    nb_repas : integer;
                    prix_repas : argent;
                    montant_repas : argent;
                    end;
            étranger : begin
                    arrivée_étranger : time;
                    départ_étranger : time;
                    nb_jours : integer;
                    prix_jour : argent;
                    montant_étranger : argent;
                    end;
            end;
        récapitulatif : begin
                    frais_mission : argent;
                    somme_de : string(60);
                    frais_inscription : argent [nonapplicable];
                    total : argent;
                    avance : argent;
                    reste_dû : argent;
                    date_remboursement : time > hour;
                    responsable_de : string(10);
                    nom_responsable : string(10);
                    signature_responsable : image [nullallowed];
                    signature_intéressé : image [nullallowed];
                    signature_ordonnateur : image [nullallowed];
                    end;
end;
```

Dans le formulaire abstrait "remboursement_frais", l'élément "réduction" est de type choix. A la création d'une occurrence pour le FA, on choisit un schéma pour cet élément. Si la personne a bénéficié d'une réduction (choix : 'oui'), on donne des informations sur cette réduction. Par contre si le choix est 'non' alors on ne donne rien. Les occurrences de ce FA peuvent avoir deux formes différentes selon que la personne a bénéficié ou non d'une réduction. On remarque que dans le cas du choix 'non' on n'a pas d'informations à donner, 'non' est une donnée en soi. On autorise donc un élément de choix à être défini simplement par une constante. De cette manière un domaine énuméré est décrit comme un choix entre plusieurs constantes d'un domaine de base ou spécialisé. Ce qui a été fait, par exemple, pour les schémas des éléments "type_groupe" et "valeur_groupe".

Nous verrons dans le chapitre IV, qu'à toute constante correspond un domaine particulier qui nous autorise à décrire une constante par un schéma.

L'élément "suppléments" est de type liste. Une occurrence peut avoir jusqu'à cinq suppléments, chacun ayant une identification ("ident_sup") et un prix ("prix_sup").

III.1.3 Les options pour les éléments

1- Valeurs clés

Pour un utilisateur, un formulaire a en général un identifiant unique : no de facture, no de la demande de mission, no de contrat, (no d'étudiant - année universitaire) pour le formulaire d'inscription, ... On peut également avoir des clés pour des parties de formulaires (par exemple : le numéro d'une marchandise pour une liste de marchandises dans une commande).

Les éléments clés d'un schéma portent l'option **key ..... endkey**. Par exemple, **key** id_frais : integer **endkey**; décrit l'élément "id_frais" comme clé pour le FA "remboursement_frais".

Pour décrire un ou des éléments comme clés d'une partie du schéma de FA, on introduit la clause **for** qui indique le nom de l'élément correspondant à cette partie. Par exemple, on peut écrire : **key** ident_sup : string(20) **endkey for** suppléments; qui décrit l'élément "ident_sup" du FA "remboursement_frais" comme clé du schéma "suppléments". On ne pourra pas avoir dans la liste, des suppléments portant la même identification ("ident_sup").

2- Valeurs nulles

Lorsqu'on utilise un formulaire papier plusieurs cas de figures peuvent se présenter :

- On ne donne pas de valeur à un champ parce qu'il n'existe pas de valeur pour ce champ. Par exemple si une mission n'a pas été prolongée par un séjour personnel de la personne l'ayant effectuée, il n'existe pas de valeur pour le champ "séjour_personnel" du formulaire "Remboursement de frais" correspondant à la mission.

  Pour le stockage et la gestion des formulaires par notre système GIF, il faut pouvoir donner à un élément de FA une valeur qui signifie qu'il n'existe point de valeur pour cet élément. Nous considérons donc la valeur nulle particulière ϕ représentative de la **non-information** ou la non-existence de valeur. En acceptant une telle valeur, il faut bien s'assurer qu'elle n'est pas utilisée pour les éléments qui doivent posséder une valeur pour que le formulaire soit complet. Par exemple, si pour un formulaire "Remboursement de frais", on ne donne pas de valeurs aux champs : "prix_train", "prix_avion", "mon-

tant_découcher", "montant_repas", "montant_étranger", il est clair qu'un tel remboursement ne peut pas être traité puisqu'il n'est pas complet. Accepter des valeurs **non_informatives** dans des formulaires signifie qu'il faut obligatoirement associer à un schéma de FA de la connaissance qui décrit sous quelles conditions un formulaire (occurrence) construit à partir de ce schéma est considéré comme complet.

- Il est demandé de ne pas donner de valeur à un champ. Ce qui signifie qu'il ne doit pas exister de valeur pour ce champ, que le champ n'est pas appliqué. Par exemple : si les frais d'inscription liés à une mission, n'ont pas été payés par la personne ayant effectuée la mission, il ne doit pas exister de valeur pour le champ "frais_inscription" dans le formulaire "Remboursement de frais" associé à la mission. Le champ prend alors une valeur signifiant qu'il n'est pas appliqué et cela ne rend pas la demande de "remboursement de frais" ambiguë (incohérente) ou incomplète. Lorsqu'on définit la signification du formulaire, il faut donc préciser les éléments qui peuvent ne pas être appliqués parce que cela n'influe pas sur sa cohérence et sa complétude.

  Dans le schéma de FA, ces éléments seront décrits avec l'option **[nonapplicable]**. On dira qu'ils sont *inapplicables*, qu'ils peuvent ne pas être appliqués. Ils pourront prendre une valeur nulle **inapplicable** qui est le symbole ∂. La décision d'appliquer ou de ne pas appliquer un champ n'appartient pas à l'utilisateur du formulaire : ce n'est pas un choix. A tout élément décrit avec l'option [nonapplicable] doit être associée une propriété sémantique décrivant ses conditions d'applicabilité. Un élément possédant une valeur inapplicable ne peut pas être mise à jour par l'utilisateur. Si l'élément est appliqué, il doit exister une valeur pour cet élément (pas de non-information).

  Remarque :

  Un élément décrit par le schéma E : Case choix of S1; S2; ... ; Sn; donne la possibilité à l'utilisateur de choisir d'appliquer le schéma S1 ou S2 ou ... Sn. Ceci montre que lorsqu'on fait un choix de schéma S1 pour E, implicitement on choisit de ne pas appliquer les autres schémas (S2, S3, ..., Sn). Les éléments correspondant à ces schémas sont indéfinis, et on peut voir un élément indéfini comme un élément possédant la valeur nulle ∂. On constate également qu'un élément d'un schéma choix ne peut pas avoir l'option [nonapplicable] : il ne peut pas être choisi (donc appliqué) et avoir la valeur ∂. Par ailleurs un élément inapplicable peut très bien être modélisé par un choix.

Le choix n'est pas fait par l'utilisateur mais par le système. L'option [nonapplicable] et le constructeur choix sont complémentaires. La décision d'utiliser l'un ou l'autre pour décrire une réalité est laissée au concepteur du formulaire.

- On ne peut pas donner de valeur à un champ, parce que pour l'instant cette valeur nous est inconnue. Dans le cas de formulaire concernant plusieurs intervenants, on peut se retrouver avec des champs qui s'appliquent, pour lesquels il doit exister une valeur ($\neq \phi$) et dont on ne connaît pas encore la valeur. Par exemple, lorsqu'une personne fait une "demande de mission", elle ne remplit par le champ "signature". Du point de vue de la complétude du formulaire, ce champ doit avoir une valeur ($\neq \phi$) pour être considéré comme complet et pour être accepté. Il faut donc donner au champ une valeur qui permette d'accepter le formulaire dans un état d'incomplétude temporaire. Il existe donc une valeur pour le champ, et cette valeur est inconnue. Elle sera connue lorsque le responsable signera la mission.

  Dans un schéma de FA, les éléments pouvant avoir une valeur **inconnue** seront décrits avec l'option **[nullallowed]**. Ces éléments sont considérés comme *inconnus*. Le symbole "?" représente la valeur nulle inconnue. La présence de ce symbole dans un formulaire ne le rend pas incohérent mais plutôt incomplet temporairement.

  Un élément de FA peut avoir les deux options [nullallowed] et [nonapplicable]. Ce qui signifie que l'élément peut être appliqué ou bien qu'il peut ne pas être appliqué. Si l'élément est appliqué, il doit exister une valeur pour cet élément (pas de non-information) et cette valeur peut être inconnue.

3- Valeur par défaut

Dans d'autres cas de figures, on peut vouloir préciser une valeur par **défaut** du champ, dans ce cas on le décrit avec l'option : **[default : < valeur par défaut>].**

III.2 Occurrence de FA

III.2.1 Définition informelle

Un schéma définit une classe de formulaires. Un objet particulier de cette classe est appelé occurrence. Une occurrence est elle aussi décrite par un schéma, construit à partir de celui de la classe. Chaque occurrence contient des valeurs au niveau des éléments terminaux (champs) du schéma. Du fait des nombreux constructeurs proposés pour la définition d'un schéma, des occurrences de la même classe peuvent ne pas avoir exactement le même schéma, et ceci en particulier lorsque le constructeur **choix** est utilisé. Une occurrence porte un identificateur (géré par GIF) permettant de l'identifier de manière unique dans le FA. Pour l'utilisateur, une occurrence est identifiée de manière unique par les valeurs de ses éléments **clés** (éléments clés du schéma). Lors de la création d'une occurrence, ils doivent obligatoirement prendre une valeur.

Exemple 3.4 : L'objet "demande_mission#10" est une occurrence du FA "demande_mission".

```
demande_mission#10
begin
        id_mission : M01;
        nom_prénom : DUPONT Jean_louis;
        adresse : 36, Avenue Joseph vallier, 38000, GRENOBLE;
        no_secu : 1550405243567;
        mode_paiement : CCP 150131 U;
        fonction : stagiaire;
        indice : 374;
        lieu_déplacement : begin
                           ville : Paris;
                           pays : France;
                           end;
        motif : cours BD - LOGIQUE;
        date-mission : begin
                       début_mission : '1986/01/21 17h48';
                       retour_mission : '1986/01/30 10h10;
                       end;
        imputation : begin
                     type : contrat;
                     ident_type : INRIA;
                     code_type : 960RO;
                     end;
        signature_responsable : ? ;
end;
```

Pour désigner une valeur d'élément de cette occurrence ou sous-occurrence, on utilise un trajet qui exprime le trajet emprunté dans le schéma pour atteindre l'élément. On donne au chapitre IV les trajets admis. Par exemple le trajet *demande_mission#10.lieu_déplacement.ville* désigne la donnée "Paris".

III.2.2 Contexte de manipulation

Toute manipulation de formulaire se fait en activant des opérations spécifiques. Ces opérations précisent les actions que l'utilisateur peut réaliser sur le formulaire et en particulier sur le FA sous-jacent. Toute action de mise à jour sur un formulaire correspond à une opération de mise à jour de FA.

Considérons une opération d'affectation de valeur (affecter) à un élément de FA pour une occurrence particulière. On distingue deux cas de figures :

- La valeur provient d'une action utilisateur. L'utilisateur l'a donnée explicitement ou en spécifiant une expression pour son calcul. Dans ce dernier cas, l'utilisateur a indiqué au système qu'il veut donner une valeur à l'élément par calcul (mot_clé CALCUL ou bien selection de cette option dans le menu associé à l'élément). Le système lui demande de donner l'expression de calcul et de spécifier ses conditions d'évaluation : immédiate ou interactive.
  Dans le premier cas le résultat de l'évaluation de l'expression est présenté comme valeur du champ. On conserve pour l'élément sa valeur et l'expression. A l'utilisation de l'occurrence, on fait apparaître ces deux informations. La valeur n'est pas modifiable. Dans le second cas, on conserve uniquement l'expression de calcul. On la reévalue à chaque utilisation de l'occurrence. On fait donc apparaître une valeur d'élément qui est fonction du contexte d'utilisation de l'occurrence.

- La valeur est donnée par le système parce que celui-ci dispose d'informations pour son calcul. Ces informations sont décrites par une règle de valeur. Nous y reviendrons au chapitre V. Nous montrons en particulier qu'une occurrence de FA peut être utilisée pour la valuation d'une autre occurrence.

La création de valeurs pour les éléments est faite par un utilisateur à un certain moment. Mais une création de valeur pour un élément peut changer son statut : de valuable (que l'on peut lire et sur lequel on peut écrire), il peut devenir non modifiable (sur lequel on ne peut plus écrire). Nous verrons au chapitre V que cela correspond à effectuer un changement de droit d'accès à l'élément pour

l'occurrence. Cela sous-entend qu'à chaque élément d'occurrence est associée un *mode d'accès* dont il faut tenir compte lors de son utilisation.

A chaque occurrence de FA, nous associons un contexte dit de **manipulation** qui décrit : dans quel environnement elle a été utilisée (présentation, opération spécifique, utilisateur, date), l'opération qu'elle a subi, son mode_d'accès. On maintient également pour chacun de ses éléments : son mode, comment sa valeur a été obtenue (opération : *affecter ou supprimer* faite de manière explicite ou par l'exécution d'une fonction ou d'une règle). Chaque fois qu'une occurrence est manipulée, ce contexte s'enrichit : toute opération de mise à jour peut modifier le contexte. On peut également songer à voir ce contexte comme un historique à versions successives [ADI 87]. Une version correspondrait alors à une nouvelle valeur du contexte après une opération spécifique de mise à jour.

III.2.3 Etat d'une occurrence

Une des propriétés fontamentale d'un formulaire est de posséder une connaissance qui assure la non-ambiguité ou cohérence de ses informations, destinées à une multitude de tâches bureautiques (avec ou sans formulaires). Dans la réalité, un formulaire peut être rempli par différents intervenants. Avant de devenir complet et cohérent, le formulaire peut donc passer par des phases successives d'incomplétude durant lesquelles, il lui est difficile d'assurer sa cohérence puisqu'il lui manque des informations.

Parfois, le formulaire (modèle d'échange d'informations) ne répond pas toujours aux besoins : situations non prévues, cas rares, informations érronées parce qu'on attend de nouvelles informations, etc. La cohérence du formulaire est alors fortement remise en question.

1- Cohérence et complétude

Considèrons l'opération "Remplir" (transaction GIF) pour le formulaire Demande de mission. On réalise un ensemble d'actions sur les champs du formulaire. Sur le FA, on va créer une occurrence et faire des affectation de valeurs aux éléments. L'occurrence doit vérifier certaines propriétés (règles) sémantiques. Ces règles décrivent la connaissance attachée aux éléments du FA. Par exemple :

"retour_mission" doit avoir une valeur inférieure ou égale (dans le temps) à celle de "début_mission".

L'intérêt d'une manipulation interactive de formulaire est de pouvoir vérifier au plus tôt les règles sémantiques. Les actions qui seront faites durant le remplissage ne sont pas connues à l'avance. Il est donc impossible pour GIF de décider (par une analyse préalable de "remplir") quand il doit valider une règle. Une première approche consisterait à valider toutes les règles en fin de remplissage, mais cela va à l'encontre du concept de formulaire "intelligent" possédant la connaissance pour un contrôle immédiat des actions utilisateurs. Une vérification au plus tôt des règles pourra se faire si l'on trouve dans la définition d'une règle, ses conditions de validation (voir chapitre V). Pour l'exemple ci-dessus, on aura la règle suivante :

"Quand l'élément "retour_mission" a été valué et l'élément "début_mission" a été valué, vérifier la propriété contraignant les deux éléments".

Pour une occurrence, supposons que l'opération d'affectation de valeur à "début_mission" se soit réalisée avec succès. Si une opération d'affectation de valeur à "retour_mission" est réalisée alors on traite cette opération et on exécute la règle (la règle devient active). La propriété est vérifiée. Selon le résultat de cette vérification (vrai / faux), l'opération réussit ou échoue. Dans ce dernier cas, l'opération est rejetée : "retour_mission" a la valeur précédant l'affectation.

Avec cette approche, toute action utilisateur qui "viole" une règle est rejetée au plus tôt, c'est à dire dès qu'elle se produit. Mais on ne peut pas dire que l'occurrence que l'on construit est toujours cohérente parce que certaines règles ne peuvent être vérifiées qu'en fin de remplissage. Parmi ces règles, on trouve essentiellement celles qui décrivent les propriétés inhérentes au schéma :

- un choix au moins a été réalisé pour une règle **case** n'ayant pas d'option.in +2 [nullallowed] ou [nonapplicable],
- pour un élément **list**, vérifier que sa valeur (liste de valeurs) respecte.in +2 les cardinalités (min, max),
- tous les éléments définis avec l'option [nullallowed] ou [nonapplicable] doivent obligatoirement avoir une valeur $\neq \phi$.

Une autre règle importante est celle qui définit les éléments pour lesquels il doit obligatoirement exister une valeur (ils ne peuvent pas avoir une valeur non-informative).

En fin de remplissage, on peut donc rejeter l'occurrence que l'on vient de créer ou bien l'accepter. Dans ce dernier cas de figures, on est en mesure de déterminer son état :

- **complète** : elle n'a pas de valeurs nulles inconnues. Elle est forcément **cohérente** puisque toutes les propriétés sémantiques pour les valeurs présentes ont été vérifiées. Les valeurs non-informatives ont été acceptées et les règles sémantiques concernant des éléments possédant des valeurs nulles inapplicables ne sont pas appliquées.

- **incomplète** (temporairement) : elle possède des valeurs nulles inconnues. Est-elle incohérente ou cohérente ?

  * elle est dans un état de cohérence **forte** (cohérente) s'il n'a pas de règles sémantiques concernant des éléments *inconnus*. Quelque soit les valeurs données à ces éléments, elles seront acceptées et sans effet sur les autres éléments. Elles ne risquent donc pas d'amener l'occurrence dans un état incohérent.

  * elle est dans un état de cohérence **faible** (et non d'incohérence) dans le cas contraire. En effet, on ne peut pas dire que l'occurrence est incohérente : certaines propriétés ne peuvent pas être vérifiées ce qui ne veut pas dire qu'elles ne sont pas vérifiées. L'occurrence risque de devenir incohérente mais ne le deviendra jamais puisqu'une affectation de valeur à un élément *inconnu* va entraîner l'éxécution d'une règle. La valeur est acceptée (l'occurrence passe dans un nouvel état complet ou incomplet avec cohérence faible ou forte) ou rejetée (l'occurrence reste dans l'état qui était le sien avant l'affectation). Cet état de cohérence faible dû à une incomplétude temporaire n'est donc pas problématique.

  Considérons le FA "demande_mission", on peut avoir la règle :

  la valeur de "signature_responsable" doit être celle d'un des responsables du laboratoire ou du contrat, identifié par "ident_type" et "code_type".

L'occurrence de l'exemple 3.4 est alors incomplète avec une cohérence faible : la règle ci-dessus liée à "signature_responsable", élément *inconnu* n'a pas pu être vérifiée. Dès que "signature_responsable" est valué, la règle est vérifiée. Si l'affectation de valeur est acceptée, et les propriétés de fin de remplissage vérifiées, on aboutit à une occurrence complète et cohérente. Dans le cas contraire, on reste avec une occurrence incomplète avec cohérence faible.

Si par contre, on a pas de règle pour "signature_responsable", l'occurrence de l'exemple 3.4 est incomplète et cohérente (avec cohérence forte).

Remarque :

Selon les propriétés sémantiques (règles) d'un FA on peut déterminer quel type d'incomplétude on peut avoir pour ses occurrences : avec cohérence forte (il n'y a pas de règles liées aux éléments *inconnus*) ou avec cohérence faible (il existe des règles liées aux éléments *inconnus*). On ne peut donc jamais avoir dans un FA des occurrences incomplètes cohérentes de manière forte et des occurrences incomplètes cohérentes de manière faible. On peut alors se demander s'il est nécessaire de maintenir des indicateurs de cohérence forte ou faible pour une occurrence incomplète. En règle générale, seuls les formulaires complets interviennent dans les traitements puisqu'ils véhiculent toute l'information nécessaire. Mais le manque d'analyse de cas réels, ne nous permet pas d'exclure les cas particuliers de traitements portant également sur des formulaires incomplets avec une cohérence forte.

2- Les exceptions

Dans la réalité, le formulaire, modèle d'échange, de contrôle et de structuration de l'information, introduit une certaine rigidité dans le traitement de l'information. Bien souvent, le formulaire n'est pas ou n'est plus adapté à certaines situations. Alors on accepte de l'information qui ne respecte pas le modèle, ou bien on définit un nouveau formulaire (remplaçant l'ancien).

Pour le formulaire électronique, il faut bien sûr se donner les moyens de modifier son schéma, d'apporter de nouvelles connaissances (règles) au formulaire mais également de rendre le contrôle moins rigide de manière à pouvoir accepter des occurrences de FA **exceptionnelles**. Une occurrence est qualifiée d'exceptionnelle lorsque certaines de ses valeurs : - ne sont pas conformes à ses propriétés sémantiques, - se rencontrent rarement, - sont provisoires. Une occurrence exceptionnelle doit pouvoir être acceptée de manière définitive ou provisoire par GIF et par la Base.

Dans le domaine des bases de données, peu de travaux traitent du problème des données exceptionnelles [BORG 85, ESC 87]. Nous n'apporterons pas de nouvelles solutions à ce problème. Le concept de tolérance sémantique introduit dans [ESC 87] donne une solution à la modélisation et à la gestion des données

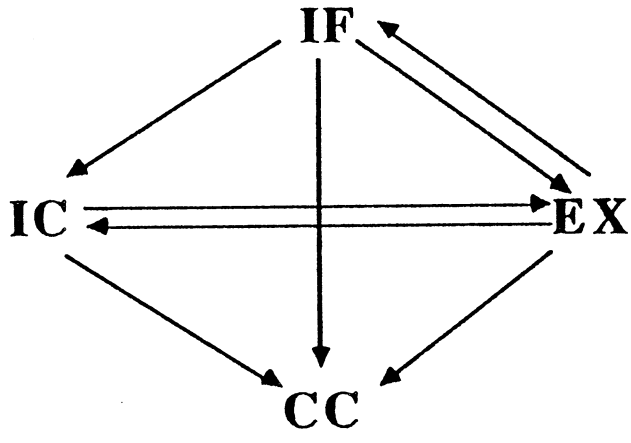
exceptionnelles dans les SGBD. Le contrôle rigoureux des données est assoupli en introduisant de la tolérance dans les contraintes d'intégrité. Ainsi l'utilisateur n'a plus à recourir à des contournements : données exceptionnelles non gérées par le SGBD, données fictives, limitation des contraintes.

Nous montrons au chapitre V qu'il est possible d'attacher au schéma d'un FA des règles introduisant de la tolérance dans le contrôle du formulaire. On peut ainsi accepter des occurrences exceptionnelles. Finalement, avec des règles autorisant des incohérences, on n'est plus capable, en fin de transaction de mise à jour d'une occurrence, de déterminer de façon sûre son état. Pour éviter ce problème et faire en sorte qu'une occurrence exceptionnelle provoque un comportement exceptionnel de GIF, on indiquera (via la règle), l'exceptionnalité d'une ou de plusieurs données. On sera alors en mesure de lui attribuer l'état **exceptionnel**. Cela sous-entend que pour chaque élément d'occurrence, on trouve un indicateur d'exceptionnalité.

3- Conclusion

Chaque occurrence de FA possède un **ETAT** : Complète-Cohérente (**CC**), Incomplète avec une cohérence Forte c'est à dire Incomplète-Cohérente (**IC**), Incomplète avec une cohérence Faible (**IF**), EXceptionnelle (**EX**). Les états IC et IF ne peuvent pas se rencontrer pour les occurrences d'un même FA.

La figure ci-dessous donne le graphe de précédence entre les états possibles d'une occurrence.



Une occurrence dans un état "Incomplète avec cohérence forte" peut passer dans un état CC. Une occurrence dans un état "Incomplète avec cohérence faible" peut rester dans son état, passer à l'état IC, CC, ou EX. Une occurrence

exceptionnelle peut rester dans son état ou bien devenir NON exceptionnelle avec un état IF, IC ou CC.

III.2.4 Conclusion

On doit attacher à chaque occurrence son état : CC, IC, IF, EX. On doit maintenir également un contexte de manipulation de l'occurrence. Il constitue le seul moyen de gérer l'évolution d'une occurrence et les liens avec les objets utilisés pour valuer certains de ses éléments. En fait, on peut introduire l'état de l'occurrence comme une information du contexte de manipulation.

Le contexte doit être stocké avec l'occurrence. Il peut être vu comme un élément particulier de tout schéma de FA.

```
contexte : list (1,*) of manipulation :
   begin  état : (CC, IC, IF, EX);
          environnement : begin
                          qui : integer;
                          format : integer;
                          ops : integer;
                          quand : time;
                          end;
          éléments : list (1,*) of élément :
                     begin  identE : integer;
                            exception : case of oui; non; end;
                            mode : string(6);
                            valuation : case V of :
                                        explicite : integer;
                                        system : begin
                                                 identF/R : integer;
                                                 type : case of R; F; end;
                                                        /*Règle, Fonction*/
                                                 end;
                                        end;
                            action : integer;
                     end;
   end;
```

III.3 FA : schéma externe d'une base de données

L'objectif de GIF est de fournir au SGBD un support pour la définition de ses applications en termes de formulaires. Chaque formulaire sert alors à visualiser ou manipuler une portion de la base de données.

Cela implique de pouvoir décrire à l'aide du concept de Formulaire Abstrait la façon dont les données de la base sont perçues par l'application ainsi que leur sémantique. Pour le domaine des Bases de données cela veut dire que le concept de FA doit recouvrir la notion de schéma externe ou vue. Le constructeur **vue** permet d'obtenir des structures de données qui présentent une partie du schéma conceptuel de la base.

On peut donc avoir un schéma de FA obtenu (1) avec le constructeur **vue** ou bien (2) contenant un ou plusieurs éléments définis avec le constructeur **vue**. Ces éléments sont qualifiés de **virtuels**. Les éléments non virtuels sont dits **complémentaires** parce qu'ils décrivent des données particulières pour le FA. De cette manière, on obtient un **schéma externe** pour la base de données apportant des informations supplémentaires par rapport à son schéma conceptuel. Dans la suite de cette section, nous nous intéressons aux éléments virtuels. Un FA virtuel n'est qu'un cas particulier d'élément virtuel.

III.3.1 Définition d'un élément virtuel

Un élément virtuel E est défini par le schéma suivant :

E : **view of** <expression-vue>.

<expression_vue> est l'expression d'une opération sur la base.

Pour GIF, tous les objets de la base sont des FA. <expression_vue> est une requête d'interrogation sur un ou plusieurs FA. Le résultat d'une requête est un FA (chapitre IV). La structure de données résultat de <expression-vue> détermine la structure qui sera associée à l'élément. Tout comme pour la notion de vue dans un SGBD relationnel, la requête correspondant à <expression_vue> ne sera pas matérialisée. Les objets de la base intervenant dans la requête sont appelés les objets **sources**. La valeur de l'élément n'appartient pas à une occurrence du FA mais aux objets sources. De nombreux travaux ont montré la difficulté de réaliser des mises à jour de vues [CAR 79, ADI 81, BAN 82, MAS 83, MAS 84]. Dans notre contexte, les mises à jour d'éléments virtuels sont autorisées parce que leurs effets sur les objets sources sont décrits explicitement dans des règles (voir section V.3.4, chapitre V).

Exemple 3.5 :

En considérant notre base exemple, il est possible de définir un FA "professeur" pour la description des données associées au formulaire "Professeur" (Figure 1.18, chapitre I). On rappelle que ce formulaire sera utilisé pour manipuler les "personnes" ayant pour fonction "professeur".

```
DEF_FA
professeur : begin
  car_person : view of
               élaguer ( sélection (personne , = (fonction, professeur)):
                         fonction);
  photo : image;
  état-civil : case of célibataire; marié_e; veuf_ve; end;
  nom_jf : string (20) [nonapplicable];
  enfants : list (0,5) of enfant :
                         begin
                             key prénom : string (20)end key for enfants;
                             date-naissance : time > hour;
                         end ;
  end;
```

La structure de données pour "car-person" correspond à la structure de données pour le FA, résultat de l'exécution de :

élaguer (sélection (personne, = (fonction, *professeur*)) : fonction)

A la définition du FA "professeur", on traite cette expression, de manière à obtenir une description complète (du point de vue structurel) de l'élément virtuel "car-person". On obtient le schéma suivant pour "car_person":

```
car_person : begin
             key no_ss : integer end key;
             nom : string (10);
             prénom : string (20);
             adresse : string (50);
             indice : integer;
             échelon : integer;
             salaire : integer;
             mode-paiement : string (30);
             end;
```

L'objet "personne" est objet source pour l'élément "car_person". Le domaine de valeur de "car_person" est l'ensemble des occurrences définissant la valeur du FA résultat de l'expression.

Une occurrence **Pf** du FA "professeur" contiendra des données concernant un professeur. La valeur pour l'élément "car_person" appartient à une occurrence de personne. Les valeurs pour les éléments "photo, état-civil, enfants" appartiennent à l'occurrence **Pf**.

Soit l'objet personne#1 une occurrence de personne

```
personne#1 : begin
             no_ss : 2550574243567;
             nom : bidou;
             prénom : arthur;
             adresse : 40 rue couche-tard 38100 grenoble;
             fonction : professeur;
             indice : 470;
             échelon : 4;
             salaire : 15000;
             mode-paiement : CCP 150131 U;
             end;
```

Une occurrence de "professeur" (dans la base) est

```
professeur#1 : begin
               car_person : personne#1
               photo : <photo du professeur>;
               état-civil : célibataire;
               nom_jf : ∂;
               enfants : {ϕ};   (l'ensemble vide)
end ;
```

III.3.2 Lien élément de FA virtuel - objets sources

Comme nous l'avons déjà dit, un élément de FA virtuel correspond à la notion de vue dans les SGBD. Il existe une dépendance de type et une dépendance de valeur forte [ADI 81] entre une vue et ses objets sources. Un élément de FA virtuel et ses objets sources sont liés du point de vue structurel (schéma), dynamique (règles) et valeur (occurrences). Nous abordons dans cette section les liens concernant les aspects dynamiques de manière très informelle voir superficielle. Nous y reviendrons au chapitre V.

1- Dépendance de schéma

Toute modification de structure d'un objet source pour un élément de FA virtuel conduit aux cas de figures suivants :

- <expression_vue> est toujours exécutable, la modification est admise.
- <expression_vue> n'est plus exécutable (suppression de donnée). Cela signifie que l'on fait référence dans la requête d'interrogation à une donnée qui est inconnue de la Base. L'élément virtuel n'est plus défini structurellement pour le FA. On aboutit à une définition de FA incomplète.

Chaque fois qu'un objet source est modifié du point de vue structurel, il convient de reflèter cette modification dans les FA le référençant. Un report automatique de modification de structure n'est pas trivial. De plus, il risque d'introduire des incohérences dans la définition du FA.

Nous pensons qu'il faut signaler à l'administrateur de la Base, les FA touchés par une modification de structure d'objets sources. Ceci implique qu'il faut maintenir pour chaque schéma d'objet source son **contexte d'utilisation** : les schémas qui l'utilisent et comment. Cette notion de contexte est à rapprochée de celle utilisée dans [RIE 84] pour les objets CAO. Pour l'objet "personne", son contexte est décrit par :

utilisé dans le FA "professeur" pour l'élément virtuel "car-person"

Disposant du contexte d'utilisation du schéma de l'objet (FA) qu'il modifie, et des commandes de modification de schéma et de règle (chapitre VII), l'administrateur possède les outils pour assurer lui même la cohérence de ses définitions.

2- Dépendance de valeur

L'élément virtuel n'a pas de valeur propre. Ce n'est qu'un contenant, une nouvelle structuration logique pour des valeurs d'objets sources.

- Interrogation d'élément virtuel

  Pour traiter une requête sur un élément virtuel, il faut disposer comme pour les vues d'une opération de composition. Pour retrouver toutes les professeurs dont le salaire est > 10000F, il faut d'abord retrouver dans l'objet "personne" les occurrences ayant pour fonction = "professeur" et "salaire" >

10000F. Pour chaque occurrence retrouvée, il faut construire l'occurrence du FA résultat. Cette phase de construction sera plus facile si on dispose pour chaque occurrence de personne, de l'identificateur de l'occurrence du FA "professeur" qui l'utilise.

- Mise à jour d'élément virtuel

Toute mise à jour d'élément virtuel doit être propagée sur ses objets sources. On sait que l'automatisation de cette propagation n'est pas triviale. On laisse donc au concepteur de l'application, la charge de définir explicitement des règles (voir chapitre V) décrivant comment doivent être effectuées les propagations.
Pour le FA "professeur", si on autorise une modification de l'élément "adresse" de "car_person", il faut écrire une règle sémantique précisant comment et quand doit être reportée cette modification sur la personne correspondante. Ce report sera accepté s'il ne rend pas l'occurrence de la personne incohérente. Dans le cas où le report est refusé, la mise à jour d'"adresse" doit être rejetée. Ceci montre que toute mise à jour d'élément virtuel sera confirmée si celle-ci est acceptée pour le FA et pour l'objet source (les exécutions de règles dans le FA et l'objet se sont déroulées avec succès).

- Mise à jour de(s) objet(s) source(s)

Toute mise à jour des objets sources est immédiatement reflétée dans l'élément virtuel. Elles peuvent avoir des conséquences sur les autres éléments du FA.

insertion

Reprenons notre exemple. Supposons que l'on crée une nouvelle occurrence de personne **P** avec la fonction "professeur". Ceci n'entraîne pas automatiquement la création d'une occurrence de FA "professeur". Cependant, si au travers du FA "professeur", on recherche par exemple tous les professeurs, on retrouve **P** mais sans valeur pour les éléments "photo", "état-civil" et "enfants". On pourra alors modifier **P**. A ce moment là seulement une nouvelle occurrence sera crée pour **P** dans le FA "professeur". Cette occurrence réfère la personne **P**. Ceci montre en fait que toute occurrence d'objets sources est **potentiellement** valeur d'élément virtuel de FA.

Suppression

Si l'on supprime une personne **P**, référencée dans une occurrence **Pf** de "professeur", il faut supprimer la valeur de l'élément "car-person" pour **Pf**. Pour **Pf**, les valeurs des éléments complémentaires n'ont plus de sens, il faut les supprimer. C'est la connaissance associée au FA (règle) qui sera alors prise en compte : "quand une valeur de "car_personne" est supprimée alors supprimer l'occurrence à laquelle il appartient". On supprimera donc l'occurrence **Pf**.

Modification

Supposons maintenant que l'on modifie une personne **P**, référencée dans une occurrence **Pf** de "professeur". Deux cas de figures sont à envisager :

1. On modifie une donnée de **P** qui intervient dans l'expression de sélection (de l'opération **sélection** définissant l'élément virtuel "car_person"). Par exemple, on modifie la valeur de "fonction" de la personne **P** (professeur --> directeur), alors **P** n'appartient plus au domaine de valeur de "car-person". Si l'on ne veut pas introduire des références folles, il faut supprimer dans **Pf** la valeur de l'élément "car-person". Pour réaliser cette suppression de manière automatique, il faut disposer de la connaissance :

"la personne **P** est utilisée dans l'occurrence **Pf** sous la contrainte : *fonction = professeur*".

Ceci implique qu'au moment de la définition du FA "professeur", on interprète l'expression de la vue de manière à définir pour l'objet "personne" une règle spécifiant que tout changement de valeur de "fonction" (*professeur -----> nouvelle valeur*) a pour effet une supression de valeur sur l'élément virtuel "car_person" de "professeur" (si celle-ci existe). La construction d'une telle règle n'est pas triviale. On devra définir explicitement cette règle pour l'objet personne.

2. On modifie une donnée de **P** qui n'intervient pas dans l'expression de sélection. On change par exemple la valeur du "salaire" de **P**. Si au niveau du FA "professeur", il existe une propriété sémantique pour "salaire", il se peut que la nouvelle valeur entraîne une incohérence de **Pf**, ne pouvant être admise. On ne peut donc pas accepter la modification sur **P**. Pour éviter ces problèmes, il faut s'assurer que les propriétés définies sur l'élément virtuel et sur l'objet source ne sont pas contradictoires. Par exemple on ne peut pas

avoir les propriétés : *salaire > 10000F* pour "personne" et *salaire > 15000* pour "professeur". Lorsqu'on définira des règles pour un élément virtuel, il faudra tenir compte de celles existantes sur le ou les objets sources.
Supposons qu'il existe également un autre FA **F**, qui utilise le "salaire" d'une personne pour calculer, par exemple, la valeur d'une "prime". Une nouvelle valeur de "salaire" pour **P** a pour effet de changer la valeur de "prime" dans l'occurrence **O** de **F** correspondant à **P**. Ce changement sur **O** peut déclencher la réalisation d'autres opérations sur **O** ou sur d'autres objets. On voit donc que la modification d'une donnée d'un objet source peut se propager sur différents objets avec des effets de bord désastreux. Que se passe-t-il par exemple si le changement sur **O** déclenche une modification de "salaire" dans **P**? On aboutit à un cycle.

Pour la modification et la suppression, on se trouve confronté au problème de la translation des mises à jour d'un objet sur un autre objet. Une recherche plus approfondie doit être menée dans ce sens. Pour l'instant, nous prenons les **hypothèses** suivantes :

- toute règle définie sur un élément virtuel de FA pourra provoquer une cascade d'actions si et seulement si ces actions portent sur le FA lui-même. De cette manière on limite les effets de bord d'une modification d'occurrence d'objet source aux occurrences de FA qui lui sont directement liées.
- toute règle définie sur un élément virtuel ne doit pas être en contradiction avec une règle existante pour sa donnée source.

Ces hypothèses seront vérifiées à la définition d'une règle concernant un objet virtuel. Nous reviendrons sur ce point dans le chapitre V.

Pour chaque valeur de personne, il faut maintenir un contexte d'utilisation qui indique dans quelle(s) occurrence(s) de FA(s) elle est utilisée. On pourra alors utiliser ce contexte pour spécifier le reflet d'une mise à jour de donnée source dans les éléments virtuels liée à cette donnée.

On voit donc que la notion de **contexte d'utilisation** doit être appliquée également aux valeurs ou occurrences des objets sources si l'on veut gérer les liens sémantiques entre ces valeurs et le(s) schéma(s) externe(s) les utilisant.

III.3.3 Conclusion

La gestion d'un FA nécessite d'associer à chacune de ses occurrences, un contexte de manipulation et des indicateurs de complétude et cohérence. Dans cette section, nous avons montré que l'utilisation et la gestion de FA comme schéma externe d'une BD necessite de disposer, en plus, de la notion de contexte d'utilisation pour les objets sources (qui peuvent être des FA) au niveau du schéma et de la valeur.

On trouvera donc également dans tout schéma d'objet de la BD (FA, catalogues , objets autres que FAs) un élément avec le schéma suivant :

```
cont_utilisation : list (0,*) of utilisation :
                   begin FA : integer; /*identificateur de FA*/
                         élément : integer;
                         occ_FA : integer [nonapplicable];
                   end;
```

Les problèmes énoncés pour notre l'exemple sont ceux rencontrés lorsqu'on doit gérer des hiérarchies de spécialisation. Ceci montre que le constructeur **vue** nous offre entre autre l'opérateur de spécialisation. Il offre également l'opérateur union.

Supposons par exemple, un autre FA "secrétaire" de schéma identique au FA "professeur". On peut alors définir un FA "employé" regroupant les "professeurs" et les "secrétaires". Son schéma sera :

```
employé : view of union (professeur, secrétaire);
```

Si par contre le FA "secrétaire" n'a pas le même schéma, on peut alors définir "employé" par :

```
begin employé : case of
                   professeur : view of professeur;
                   secrétaire : view of secrétaire;
                   end;
end;
```

Schéma cyclique

Avec les constructeurs proposés nous ne pouvons pas décrire de structure hiérarchique cyclique. Avec le constructeur **view**, on introduit de la récursivité dans un schéma :

Exemple 3.6 :

```
DEF_FA  personne : begin
                   nom : string(10);
                   prénom : string(20);
                   datenaissance : time > hour;
                   mère : view of personne;
                   end;
```

CHAPITRE IV

MANIPULATION de FA

Nous avons besoin d'un principe de connaissance qui non seulement respecte mais, révèle le mystère des choses.

E. Morin

CHAPITRE IV

MANIPULATION de FA

Au chapitre III, nous avons donné la commande pour la définition d'un FA et nous avons introduit les notions de schéma, règles et occurrence de FA. Au niveau conceptuel, les formulaires existants dans notre esprit (des demandes de missions) sont classés dans un ensemble : le FA ("demande_mission").

Dans la section IV.1 de ce chapitre, nous donnons une définition plus formelle d'un FA en montrant que les notions de **SCHEMA_FA** : schéma + règles et **FA** : ensemble d'occurrences, cohabitent en permanence. Pour la présentation des opérations, nous ne considérons pas la composante dynamique (règles) de l'intention d'un FA. Le mot FA recouvre donc les notions de schéma et d'ensemble d'occurrences.

Ensuite nous présentons les FA_opérations : opérations pour l'interrogation des FA (section IV.2) et opérations pour la mise à jour des FA (section IV.3). Dans la section IV.4, nous montrons comment le pouvoir d'expression des FA_opérations est étendu par l'utilisation de fonctions : opérations pour la manipulation des types de données que l'on rencontre dans un formulaire (texte, temps, image, ...).

Bien entendu, il s'agit ici d'avoir pour les FA_opérations une approche analogue à celle de l'algèbre relationnelle. En particulier, le formalisme proposé ne constitue pas une interface conversationnelle pour l'utilisateur final avec GIF. Pour définir notre algèbre de FA, nous nous sommes inspirés des travaux réalisés sur les relations non en première forme normale [SCH 84, SCH 86, FIK 83, ABI 84] et plus particulièrement des travaux sur les objets complexes [ABI 86a, ABI 87].

IV.1 SCHEMA_FA et FA

Nous supposons l'existence d'ensembles infinis de valeurs appelés domaines. On a les domaines : entier, réel, chaîne de caractères, texte, image, son, ... Les valeurs de ces domaines ne sont pas décomposables. Ce qui signifie qu'on ne s'intéresse pas à leur structure. Pour simplifier la présentation, nous considérons par la suite en général qu'un seul domaine que nous appelons **D**.

Nous considérons les valeurs **nulles** suivantes :

- ∂ - valeur nulle inapplicable.
- ? - valeur nulle inconnue.
- ϕ - valeur représentant la non-information, présente dans tous les domaines que nous considérons.

Dans le premier cas, la présence du symbole ∂ dans un élément (champ ou partie) de formulaire, signifie que cet élément ne s'applique pas au formulaire : il ne doit pas exister de valeur pour l'élément. Dans le second cas (symbole ?), l'élément s'applique au formulaire, il a une valeur qui pour l'instant est inconnue. Finalement, le symbole ϕ signifie "ce qui n'existe point". La présence d'un tel symbole dans un élément est interprétée comme : "l'élément n'a pas de valeur". Un ensemble vide est considéré comme un ensemble contenant de la non-information [ROT 85]. Un ensemble est défini par rapport à ses éléments : on parle par exemple d'un ensemble P de personnes. Si cet ensemble P est vide, cela signifie qu'il est vide de "personne". On peut voir P comme un ensemble contenant un élément représentatif d'une personne "qui n'existe point". Une interprétation correcte serait de dire que notre perception du monde réel (avec toutes ses personnes) étant limité, nous ne pouvons décider pour chaque personne si elle appartient ou non à l'ensemble. Par contre, nous pouvons affirmer que pour la partie du monde réel qui nous intérresse pour l'instant, il n'existe point de personne appartenant à l'ensemble.

Nous considérons également le symbole particulier $\square$ pour l'introduction de constantes dans le schéma.

IV.1.1 Schéma et élément

Dans cette partie, nous nous intéressons à la notion de shéma. Cette notion est à rapprocher de la notion de type pour objets complexes. Nous utilisons le formalisme proposé dans [ABI 87]. Nous étudions des schémas obtenus en utilisant les constructeurs : agrégat, list, choix et ensemble. Nous ne considérons pas des schémas cycliques. Nous nous intéressons donc uniquement à des schémas ayant une structure d'arbre.

1- Définition d'un schéma

L'ensemble **S** des schémas possibles est défini par :

A, A1, A2, ..., An des noms distincts et Si = Ai:*di* ∈ **S**, i∈[1 ...n]

- A:D et A:□ ∈ **S**
- A: [S1, S2, ..., Sn] ∈ **S** /*agrégat*/
- A: S1 | S2 | ... | Sn ∈ **S** /*choix*/
- A: < S1 >$_n$ ∈ **S**, S1 ≠ A1:□ /*liste d'au plus n éléments*/
- A: { S1 } ∈ **S**, S1 ≠ A1:□ /*ensemble*/
- (S1)$_{nonapplicable}$ et (S1)$_{nullallowed}$ ∈ **S**

Les définitions de schémas *choix*, *liste* et *ensemble* doivent remplir la condition :
Si non de la forme Si$_{nonapplicable}$

Les exemples suivants illustrent la notion de schéma.

Exemple 4.1 :

(1) (adresse: [rue:chaîne, codepostal:entier, ville:chaîne])$_{nullallowed}$ un agrégat

(2) situation: célib:□
| marié_e: [(nom_jf:chaîne)$_{nonapplicable}$]
| divorcé_e :□ un choix

(3) notes: < (note:entier)$_{nullallowed}$ >3 une liste

(4) personne : [nom: chaîne,
activités: { act: [nomact:chaîne, nban:entier]}] un ensemble

Le schéma (1) permet de représenter une adresse qui peut être inconnue. Le schéma (2) représente un choix ternaire. Le constructeur choix nous permet de représenter des formulaires ou certaines parties doivent être remplies selon les circonstances : l'utilisateur choisit ou bien le choix est déterminé par le système suite au remplissage d'autres parties. Avec le constructeur choix dans un schéma S, on peut obtenir des objets hétérogènes construits à partir S. Le schéma (4) représente une personne avec un nom et un ensemble d'activités.

Notation

Tout schéma Si de **S** est de la forme Ai:*di*,

On utilise les notations :

- **nom(S)** pour le premier composant de S. Par nom(S), on obtient 'Ai', le nom du schéma.
- **des(S)** pour la seconde composante de S. Par des(S), on obtient *di*, la description de S.

Le schéma (3) de l'exemple 4.1 se nomme **notes** et sa description est $< (\text{note: entier})_{\text{nullallowed}} >_3$.

2- élément d'un schéma

Intuitivement, on appelle élément d'un schéma S, un nom de schéma composant S. Pour le schéma (1) : "adresse" de l'exemple 4.1, "rue", "codepostal" et "ville" sont des éléments.

Pour la définition des éléments d'un schéma, on introduit la fonction **over** qui associe à tout schéma, l'ensemble des schémas le composant.

Cette fonction **over** : **S** -------> **S** est définie par :

S ∈ **S**,de la forme

- A:D ou A:□, **over**(S) = S
- A: [S1, S2, ..., Sn] ou A : S1 | S2 | ... | Sn, **over**(S) = { S1, S2, ..., Sn }
- A: $< S1 >_n$ ou A: { S1 }, **over**(S) = S1
- $(S1)_{\text{nonapplicable}}$ ou $(S1)_{\text{nullallowed}}$, **over**(S) = **over**(S1)

Soit S, un schéma. X ∈ over(S) est appelé un **schéma d'élément** et nom(X) est appelé **élément**. L'ensemble des éléments d'un schéma S est alors défini par :

Elem(S) = { E | ∃ X ∈ over(S) et E = nom(X) }

Par exemple, l'ensemble des éléments du schéma (4) "personne" de l'exemple 4.1 est :

Elem(personne) = {nom, fonction, activités}

Soit S ∈ **S** et S1, S2, ..., Sn ∈ over(S). D'après la définition d'un schéma, nom(S1) ≠ nom(S2) ≠ ... ≠ nom(Sn). Ces noms sont des éléments de S. Connaissant un élément (= nom(Si)) de S, on peut donc toujours retrouver son schéma (Si). La fonction **Sch_élem** associe à un élément, son schéma. Elle est définie par :

$$E \in Elem(S) \Rightarrow \exists\, X \in over(S) \text{ et } nom(X) = E \text{ et } \mathbf{Sch\_élem}(E) = X$$

3- Représentation d'un schéma

Jusqu'ici nous avons donné une représentation linéaire d'un schéma. On peut également représenter un schéma par un arbre (Figure 4.1). Les constructeurs sont représentés de la manière suivante :

× : agrégat
\+ : choix
$*_n$: liste
* : ensemble

Les arbres de la Figure 4.1 représentent les schémas (2), (3) et (4) de l'exemple 4.1.

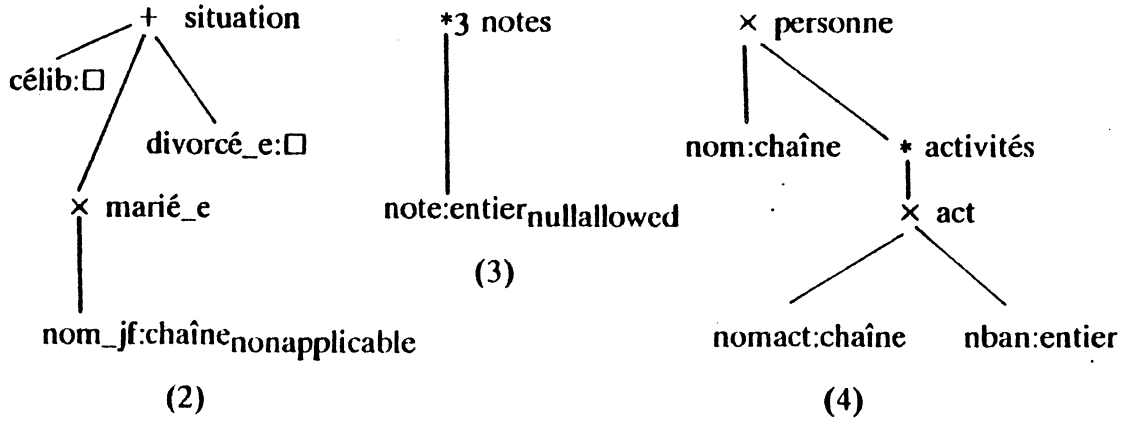


Figure 4.1 Trois schémas.

On constate que pour S = (A : *da*)$_{\text{nullallowed}}$ ou S = (A : *da*)$_{\text{nonapplicable}}$, on met l'option avec le noeud A.

IV.1.2 Occurrence et valeur d'élément

Un objet construit à partir d'un schéma est appelé une occurrence. L'ensemble des occurrences possibles pour un schéma S définit le **domaine** de S.

Définition : Le domaine d'un schéma S est défini par :

- $S = A{:}D$, $dom(S) = \{ A{:}a \mid a \in D \}$
- $S = A{:}\square$, $dom(S) = \{ A{:}\square \} = \{A\}$
- $S = A{:}[\, S1, S2, \ldots, Sn \,]$,
  $dom(S) = \{ A{:}[\, s1, s2, \ldots, sn \,] \mid \forall i \in [1\ldots n], si \in dom(Si) \}$
- $S = A : S1 \mid S2 \mid \ldots \mid Sn$,
  $dom(S) = \{ A{:}s \mid \exists i \in [1\ldots n], s \in dom(Si) \}$ /*domaine d'objets hétérogènes*/
- $S = A{:}\langle S1\rangle_n$,
  $dom(S) = \{ A{:}\langle s1, s2, \ldots, sk \rangle) \mid$ pour $0 \leq k \leq n$ et $\forall i \in [1\ldots k]$, $si \in dom(S1) \}$
- $S = A : \{ S1 \}$,
  $dom(S) = \{ A : \{s1, s2, \ldots, sk\} \mid$ pour $k \geq 0$ et $\forall i,j \in [1..k]$, $i \neq j$, $si \neq sj$ et $si, sj \in dom(S1) \}$
- $S = (S')_{nonapplicable}$,

  1- $S' = A{:}D$, $dom(S) = dom(S') \cup \{A{:}\partial\} - \{A{:}\phi\}$

  2- $S' = A{:}[\, S1, S2, \ldots, Sn \,]$,
  $dom(S) = dom(S') \cup$
  $\{A{:}[\, v1, v2, \ldots, vn \,] \mid vi \in (dom(Si_{nonapplicable}) - dom(Si))$,
  pour $i \in [1\ldots n]\}$

  3- $S' = A{:}\, S1 \mid S2 \mid \ldots \mid Sn$, $dom(S) = dom(S') \cup \{A{:}\partial\}$

  4- $S' = A{:}\langle S1\rangle_n$,
  $dom(S) = dom(S') \cup$
  $\{A{:}\langle v1, v2, \ldots, vn\rangle \mid vi \in (dom(S1_{nonapplicable}) - dom(S1))$
  pour $i \in [1\ldots n]\}$

  5- $S' = \{S1\}$,
  $dom(S) = dom(S') \cup$
  $\{A{:}\{v1\} \mid v1 \in (dom(S1_{nonapplicable}) - dom(S1))\}$

6- $S' = (S1)_{nonapplicable}$
$dom(S) \doteq dom(S') \cup \{ dom(S1_{nonapplicable}) - dom(S1))\}$

- $S = (S')_{nullallowed}$,

1- $S' = A:D$, $dom(S) = dom(S') \cup \{A:?\} - \{A:\phi\}$

2- $S' = A: [S1, S2, ..., Sn]$,
$dom(S) = dom(S') \cup$
$\{A: [v1, v2, ..., vn] \mid vi \in (dom(Si_{nullallowed}) - dom(Si))$,
pour $i \in [1...n]\}$

3- $S' = A: S1 \mid S2 \mid ... \mid Sn$, $dom(S) = dom(S') \cup \{A:?\}$

4- $S' = A: <S1>_n$,
$dom(S) = dom(S') \cup$
$\{A: <v1, v2, ..., vn> \mid vi \in (dom(S1_{nullallowed}) - dom(S1))$,
pour $i \in [1...n]\}$

Une liste inconnue est une liste contenant n éléments inconnus.

5- $S' = \{S1\}$,
$dom(S) = dom(S') \cup$
$\{ A: \{v1\} \mid v1 \in (dom(S1_{nullallowed}) - dom(S1))\}$

Un ensemble inconnu est un ensemble contenant un élément inconnu

La définition d'un domaine pour un schéma S (non choix) avec l'option *nullallowed* (*nonapplicable*) montre qu'une occurrence inconnue (inapplicable) pour S est composée de sous-occurrences inconnues (inapplicables). Dans le contexte d'une telle occurrence, on peut considérer que les éléments de S porte eux mêmes les options *nullallowed*, *nonapplicable*. Dans l'expression des requêtes sur S, on pourra donc considérer que ses éléments portent eux-mêmes l'option.

Les exemples suivants illustrent la notion d'occurrence.

Exemple 4.2 :

(1) adresse: [rue: *10 Av Couchetard*, codepostal: *38100*, ville: *Grenoble*]
adresse: [rue :?, codepostal:?, ville:?]

Ces occurrences ont été construites selon le schéma "adresse" de l'exemple 4.1.

La seconde occurrence signifie que "adresse" est pour l'instant inconnue. On constate que cette occurrence est également une occurrence pour le schéma :

adresse: [(rue:chaîne)$_{nullallowed}$, (codepostal:entier)$_{nullallowed}$,
(ville:chaîne)$_{nullallowed}$]

Ce qui signifie nullement que l'on puisse, à partir du schéma "adresse", construire l'occurrence adresse : [rue:*la charrière*, codepostal:*01630*, ville:?] qui définit une adresse dont la ville est inconnue puique le schéma de l'élément "ville" ne porte pas explicitement l'option *nullallowed*.

(2) situation: marié_e: [nom_jf:∂]
situation: marié_e: [nom_jf:*benoit*]
situation: célib: □

Ces occurrences ont été construites à partir du schéma (2) de l'exemple 4.1. On constate qu'elles ne possèdent pas le même schéma. Ceci est dû au fait que "situation" est un schéma *choix*. Pour la première occurrence, on a choisi le schéma de nom "marié_e" mais l'élément "nom_jf" de ce schéma ne s'applique pas à cette occurrence (il possède la valeur nulle inapplicable ∂). Dans la troisième occurrence, le schéma "célib" a été choisi.

(3) notes : < note:*10*, note:*12*, note:*09* >
notes : < note:*15*, note:? >

sont des occurrences construites selon le schéma (3) de l'exemple 4.1. On remarque que dans la seconde occurrence, on définit une liste possédant seulement deux notes, la seconde note étant inconnue.

(4) personne: [nom:*marie*,
activités: { act: [noma:*ski*, nban:*15*],
act: [noma:*delta*, nban:*3*],
act: [noma:*danse*, nban:*15*],
act: [noma:*théâtre*, nban:*3*] }]

personne: [nom:*marine*, activités : { act: [noma:ϕ, nban:ϕ] }]

Les deux occurrences ci-dessus ont été construites selon schéma "personne" (4) de l'exemple 4.1. On constate que la première occurrence définit une personne de nom *marie*, pratiquant quatres activités qui sont 4 occurrences construites à partir

du schéma de l'élément "act". La seconde occurrence définit une personne pour laquelle l'ensemble "activités" est vide.

Ces exemples montrent que l'on conserve de manière explicite les noms (de schémas) dans une occurrence. Ceci nous permet, étant donné une occurrence, de (presque) savoir le schéma auquel elle correspond.

Notation

Soit O une occurrence de dom(S). Si S est de la forme :

(1) A:D alors O = A:a et O.A = a

(2) A:□ alors O = A:□ et O.A = A

(3) A: [S1, S2, ..., Sn], Si = Ai:*di* alors
O = A: [A1:*v1*, A2:*v2*, ..., An:*vn*] et O.Ai = Ai:vi.
Ai désigne donc la ième sous-occurrence Ai:*vi* de 0. Cette sous-occurrence a été construite à partir du schéma Si = Sch_élem(Ai).

(4) A: S1 | S2 | ... | Sn, Si = Ai:*di* alors
O = A:I, I = A1:*v1* ou A2:*v2* ou ... An:*vn*. O.Ai = Ai:*vi* ou bien O.Ai est indéfini.
Ai désigne donc une sous-occurrence I de O (qui peut être indéfinie), correspondant à Sch_élem(Ai).

(5) A: < A1:*d1* $>_n$ alors
- O = A: < A1:*v1*, A1:*v2*, ..., A1:*vk* >, k ≤ n
  O.A1 est une sous-occurrence I_i = A1:*vi* quelconque de la liste O et I_i correspond à Sch_élem(A1).
- O<j> = A1:*vj* (1 ≤ j ≤ k) est la jème sous-occurrence de O.
  et A<j> désigne O<j>.

Une occurrence peut représenter une liste vide, qui contient n élements représentatifs de ce "qui n'existe point". Pour simplifier l'écriture, on mettra dans la liste un seul élément représentant la non-information. De plus on ne tient pas compte de la description (*d1*) du schéma A1, élément de S et on note : A: <A1:ϕ> la liste vide pour le schéma S. Plus généralement on note <ϕ> une liste vide.

(6) A: { A1:*dl* } alors O = A: { A1:*v1*, A1:*v2*, ..., A1:*vk* },
O.A1 est une sous-occurrence I_i = A1:*vi* quelconque de l'ensemble O et I_i correspond à Sch_élem(A1).

Une occurrence peut représenter un ensemble vide, contenant une sous-occurrence décrivant la non-information. On utilise la notation simplifiée A:{A1:ϕ} pour représenter l'ensemble vide construit à partir de S. {ϕ} dénote l'ensemble vide.

(7) $(S')_{nonapplicable}$ alors O est une occurrence (non "vide") de S' ou bien O est une occurrence *inapplicable* dont la forme dépend de S'. Nous énoncons ci-après la forme simplifiée utilisée pour représenter une telle occurrence.

1- si S' = A:D, O = A:∂

2- si S' = A: [S1, S2, ..., Sn], O = A:[∂]

3- si S' = A: S1 | S2, ... | Sn, O = A:∂

4- si S'= A: $<A1:dl>_n$, O = A: $<A1:\partial>_n$ = A:<∂>

5- si S' = A: {A1:*dl*}, O = A:{A1:∂} = A:{∂}

(8) $(S')_{nullallowed}$ alors O est une occurrence (non "vide") de S' ou bien O est une occurrence *inconnue* dont la forme dépend de S'. Nous énoncons ci-après la forme simplifiée utilisée pour représenter cette occurrence.

1- si S' = A:D, O = A:?

2- si S' = A: [S1, S2, ..., Sn], O = A:[?]

3- si S' = A: S1 | S2 | ... | Sn, O = A:?

4- si S'= A: $<A1:dl>_n$, O = A: $<A1:?>_n$ = A:<?>

5- si S' = A: {A1:*dl*}, O = A:{A1:?} = A:{?}

Avec cette notation l'occurrence pour le schéma "adresse" (1) de l'exemple 4.2, s'écrit simplement : adresse:[?].

Pour une occurrence O, construite selon un schéma S de la forme (3) ou (4), on remarque qu'un élément de ce schéma permet de désigner une sous-occurrence de O (qui peut être indéfinie dans le cas du choix). Par contre si S est de la forme (5) ou (6), un élément de ce schéma ne permet pas de désigner de façon précise une

sous-occurrence de O. De manière générale on dira que certains éléments "identifient" une sous-occurrence de O.

Valeur d'élément

Dans le contexte d'une occurrence, construite selon un certain schéma S, un élément de ce schéma représente (de manière générale) une valeur précise : une sous-occurrence de O lorsque celle-ci existe.

Soit S = A:*da* un schéma, Ai ∈ Elem(S) et O ∈ dom(S)

- O.Ai pour S schéma *agrégat*, *choix*, *liste* ou *ensemble*
- O<i> pour S schéma *liste* et *da* = < Ai:*di*>$_n$

sont des sous-occurrences de O contruites selon le schéma Si = Sch_élem(Ai). Une telle sous-occurrence est appelée **valeur d'élément** Ai. Lorsque S (non choix) porte une option *nullallowed* ou *nonapplicable*, Si hérite de l'option (dans le contexte de l'occurrence).

Exemple 4.3 :

Reprenons le schéma "situation" :

situation: célib:□
| marié_e: [(nom_jf:D)$_{nonapplicable}$]
| divorcé_e :□

"célib", "marié_e", "divorcé_e" sont des éléments de ce schéma.

L'occurrence O = situation: marié_e: [nom_jf:∂] appartient à dom(situation). Pour cette occurrence, O.marié_e = marié_e: [nom_jf:∂] est une valeur d'élément "marié_e" par contre les sous-occurrences O.célib et O.divorcé_e sont indéfinies. Les éléments "célib" et "divorcé_e" désignent aucune sous-occurrence dans le contexte de O.

Considérons le schéma notes: < (note: entier)$_{nullallowed}$>. Dans le contexte de l'occurrence notes: <note:*13*, note: *14*, note: *15*>, l'élément "note" désigne une note parmi une liste de 3 notes. Par contre "notes<2>" désigne la sous-occurrence note:*14*. Dans le contexte de l'occurrence notes:<note:*15*, note:?>, "notes<2>" désigne la sous-occurrence note:?, une note dont la valeur est pour l'instant inconnue.

Considérons maintenant le schéma activités: { act: [nomact:chaîne, nban:entier] }. Une occurrence O possible pour ce schéma est :

activités: { act: [noma: *ski*, nban: *15*],
act: [noma: *delta*, nban: *3*],
act: [noma: *danse*, nban: *15*],
act: [noma: *théâtre*, nban: *3*] }

Dans le contexte de O, l'élément "act" du schéma désigne une activité parmi un ensemble d'activités. On pourra écrire par exemple : $\forall I \in O$, O.act = I. "act" désignera successivement les 4 sous-occurrences de O.

De manière générale, pour S = {A1:*dl*}, O = A: {I_1, I_2, ..., I_k} et pour S = A : <A1:*dl*>$_n$, O = A : <I_1, I_2, ..., I_k>, A1 désigne une valeur parmi k valeurs possibles de l'ensemble ou de la liste. L'utilisation de l'élément A1 pour désigner une sous-occurrence I_i de O sera possible dans un certain contexte qui précise que O.A1 = I_i.

IV.1.3 Formulaire Abstrait

Lorsqu'on décrit une application bases de données en termes de formulaires, on regroupe les objets de même nature et même sémantique dans un ensemble nommé FA. Par exemple : des demandes de mission sont regroupées dans le FA "demande_mission". Comme nous l'avons vu (chapitre III), le FA est porteur d'un ensemble d'informations décrivant l'organisation et la sémantique des données pouvant être présentes dans le formulaire et les données elles-même (occurrences). Derrière la notion de FA, il faut distinguer l'intention du FA (la signification de l'ensemble), de son extension (l'ensemble lui-même)

1- SCHEMA_FA

Un SCHEMA_FA correspond à l'intention du FA. Il est défini par le concepteur à l'aide de la commande DEFINE_FA (chapitre III). Nous avons vu que cette définition correspondait à décrire deux composantes : le schéma du FA et les règles sémantiques qui lui sont associées. Nous définissons un SCHEMA_FA par :

FA = <S, R>,

- $S \in \mathbf{S}$
- R est un ensemble de règles

FA est un SCHEMA_FA valide (au sens du chapitre III) si et seulement si S n'est pas de la forme (A : □), $(Si)_{nonapplicable}$ ou $(Si)_{nullallowed}$.

2- FA

Le FA correspond à un moment donné à un ensemble de formulaires stockés dans la base. Il se définit à partir d'un SCHEMA_FA <S, R>, comme un ensemble d'occurrences.

Chaque occurrence est construite selon le schéma S et "vérifie" les règles R. Nous verrons dans le chapitre V en quoi consiste cette "vérification". Parmi ces règles, on trouve la règle classique d'appartenance de l'occurrence au domaine du schéma S (inhérente à la définition d'une occurrence). On trouve également des règles décrivant des contraintes que les données de l'occurrence doivent vérifier. En fait toute exécution (vérification) d'une règle se traduit par un échec ou un succès. Une occurrence est acceptée en tant que telle parce qu'elle est dans un certain état traduisant une exécution avec succès des règles la concernant. On notera R[O] le résultat d'une vérification avec succès des règles pour une occurrence O. La définition formelle d'un FA est donc :

$$FA = \{ O \mid O \in dom(S) \text{ et } R[O] \}$$

La différence entre SCHEMA_FA de FA et FA est comparable à celle que l'on rencontre avec les concepts de schéma de relation et de relation du modèle relationnel et plus généralement entre type de données et données. Dans notre modèle de FA, les notions de SCHEMA_FA et FA cohabitent en permanence. Ce qui traduit bien le fait que dans le langage courant on utilise le mot "formulaire" aussi bien pour désigner (1) un formulaire type F où tous les champs sont vides, (2) les formulaires (définis à partir de F) où les champs contiennent des informations.

Pour exprimer cette complémentarité entre SCHEMA_FA et FA, nous avons choisi d'utiliser également le terme "FA" pour décrire le couple (FA, FA) qui correspond bien à ce que nous entendions jusqu'ici par FA.

Notre objectif dans ce chapitre est de présenter les opérations pour la manipulation d'un FA. Une opération traite un ensemble d'occurrences et on utilise pour la définir le schéma qui sert à construire les occurrences. On s'intéresse donc ici essentiellement à la composante schéma (S) du SCHEMA_FA <S, R> et au FA

lui même (ensemble d'occurrences). Dans la suite de ce chapitre, lorsqu'on parle de FA, on fait référence au couple :

(S, Δ)

- $S \in \mathbf{S}$
- $\Delta \subseteq dom(S)$

Notation

On utilisera les notations suivantes lorsqu'on aura à distinguer le schéma S d'un FA de son extension.

Soit Γ (S, Δ) un FA

(1) **sch**(Γ) dénote le premier composant de Γ : son schéma S

(2) **val**(Γ) dénote le second composant de Γ : l'ensemble d'occurrences Δ, le FA a propement parlé.

Nous avons vu en section V.1.1 que nom(S) dénote le premier composant d'un schéma S (son nom) et des(S) nous donne sa description. Alors on obtient par :

nom(sch(Γ)), le nom de Γ.

des(sch(Γ)), la description *da* du schéma de Γ.

Pour plus de commodité, on étend les définitions des fonctions *nom* et *des* (définies pour un schéma) pour un schéma de FA :

nom(Γ) := nom (sch(Γ))

des(Γ) := des (sch(Γ))

Exemple 4.4 :

Nous présentons ici le FA "étudiant" : ($S_{étudiant}$, $\Delta_{étudiant}$). Le schéma, $S_{étudiant}$, est donné en figure 4.2.

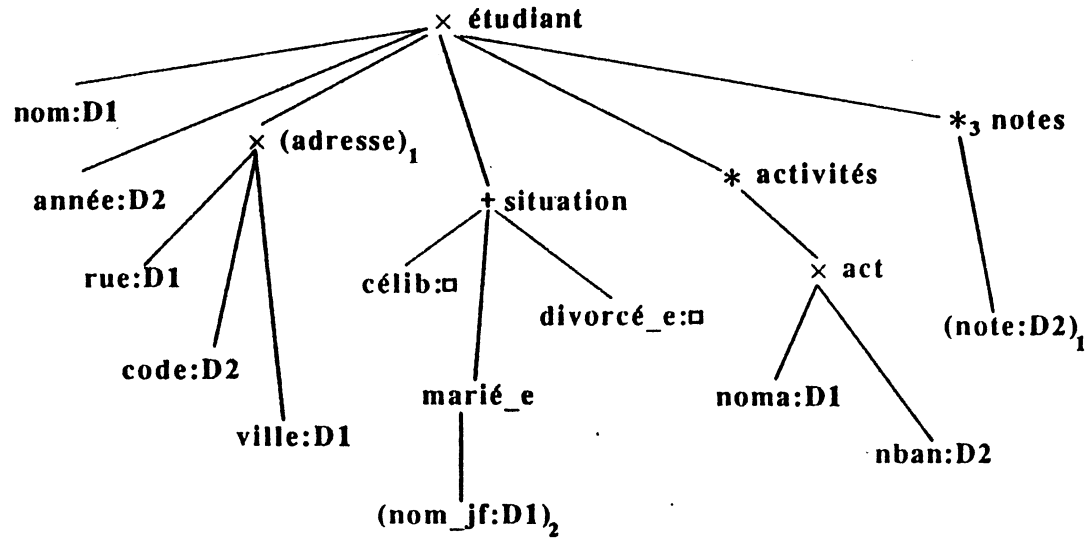


Figure 4.2 Schéma du FA "étudiant".

Il décrit la structure de données à partir de laquelle seront construites les occurrences "étudiant" (représentant des étudiants ou des étudiantes). Chaque "étudiant" devra posséder un nom, une année de naissance, une adresse qui peut être inconnue, une situation familiale, et une liste de notes pouvant contenir des notes inconnues. On veut de plus connaître pour chaque "étudiant", l'ensemble des activités qu'il pratique (actuellement). Cet ensemble peut être vide : l'"étudiant" pratique aucune activité. Pour chaque activité décrite, on donne son nom et le nombre d'années de pratique.

Une valeur possible du FA "étudiant" (Δétudiant) est donnée ci-après. Pour représenter les valeurs *inconnues* ou les valeurs *non_informatives*, nous avons utilisé les notations simplifiées.

{
étudiant: [nom:*balou*, année:*1960*,
adresse: [rue:*10 av couchetard*, code:*38100*, ville:*Grenoble*],
situation: marié_e: [nom_jf:∂],
activités: {act:ϕ}, notes: <note:*10*, note:*06*, note:*08*>]

étudiant: [nom:*geser*, année *1958*, adresse:[?], situation: célib:□,
activités: { act: [noma:*ski*, nban: *15*],
act: [noma:*delta*, nban:*3*],
act: [noma:*danse*, nban: *15*],
act: [noma:*théâtre*, nban:*3*] }
notes: <note: *13*, note: *13*, note: *13*>]]

étudiant: [nom:*dobey*, année *1956*,
adresse: [rue:*50 cr J. Jaures*, code:*38100*, ville:*Grenoble*],
situation: célib:□,
activités: { act: [noma:*photo*, nban:*3*],
act: [noma:*ski*, nban: *11*] },
notes: <note: *10*, note: *11*, note:*07*>]]

étudiant: [nom:*nay*, année *1956*, adresse:[?],
situation: marié_e: [nom_jf: *roy*],
activités: { act: [noma:*golf*, nban:*5*] },
notes: <note: *12*, note: *14*, note:?>]]

}

valeur du FA "étudiant"

Dans ce FA, la première occurrence représente l'étudiant de nom *balou*, né en 1960, marié, habitant au *10 av couchetard 38100 Grenoble*, ne se livrant à aucune activité et ayant obtenu les note *10, 06, 08*. Le schémas "nom_jf" n'a pas été appliqué. La seconde occurrence représente l'étudiant, de nom *geser*, célibataire pour lequel on ne connaît pas encore l'adresse. Il pratique quatre activités et a obtenu trois fois la même note. La dernière occurrence définit une étudiante, dont la liste de note contient une note inconnue.

Il est immédiat de constater que la valeur du FA "étudiant" peut être vue comme une occurrence construite à partir du schéma *ensemble* :

"étudiant" : { sch(étudiant) }.

Ceci nous permet de dire de suite qu'une opération de manipulation sur un FA $\Gamma = (S, \Delta)$ correspond à appliquer cette opération sur une occurrence construite selon le schéma *ensemble* Γ : { S }.

élément de FA

Dans les sections V.1.1 et V.1.2, nous avons introduit les notions d'"élément" et de "valeur d'élément". Ces notions peuvent être utilisées pour un FA.

Soit le FA $\Gamma = (S, \Delta)$, l'ensemble des éléments de Γ est défini par :

$$Elem(\Gamma) = Elem(S)$$

Exemple 4.5 :

Elem(étudiant) = { nom, année, adresse, situation, activités, notes }

On rappelle qu'une valeur d'élément Ai d'un schéma S est défini dans le contexte d'une occurrence particulière O pour ce schéma. Si sch(Γ)= A:*da*, Ai $\in$ Elem(Γ) et O $\in$ val(Γ), on a alors les valeurs d'éléments Ai :

$$\begin{cases} O.Ai \\ O<i> \text{ pour } des(\Gamma) = da = < Ai:di>_n \end{cases}$$

On pose : S_{Ai} = Sch_élem(Ai). $S_{Ai} \in S$

V_{Ai} = { O.Ai } ou V_{Ai} = {O<i>}.

$V_{Ai} \subseteq dom(S_{Ai})$ puisque O.Ai ou O<i> $\in dom(S_{Ai})$.

De la définition d'un FA, on en déduit que :

$\Gamma_{Ai} = (S_{Ai}, V_{Ai})$ est un FA

Exemple 4.6 :

Reprenons notre FA "étudiant" (Exemple 4.4) et l'occurrence :

étudiant: [nom:*geser*, année *1958*, adresse:?, situation: célib:□,
activités: { act: [noma:*ski*, nban:*15*],
act: [noma:*delta*, nban:*3*],
act: [noma:*danse*, nban:*15*],
act: [noma:*théâtre*, nban:*3*] }
notes: <note:*13*, note:*13*, note:*13*>] }]

Dans le contexte de cette occurrence, on peut définir le FA

$\Gamma_{activités} = (S_{activités}, \{ V_{activités}\})$ où

- $S_{activités}$ = activités: { act: [noma:D1, nban:D2] }
- $V_{activités}$ = { activités: { act: [nomc:*ski*, nban:15],
act: [nomc:*delta*, nban: 3],
act: [nomc:*danse*, nban:15],
act: [nomc:*théatre*, nban:3] } }

La valeur du FA $l'_{activités}$ est un ensemble contenant une occurrence. Dans le contexte de cette occurrence, on pourra définir 4 FA l'_{act}.

Remarque :

On constate donc qu'un FA défini à partir d'un élément a lui même des éléments qui définissent des FA. Pour l'exemple 4.6 ci-dessus, "act" est élément du FA $l'_{activités}$ mais non du FA "étudiant".

Tout noeud d'un arbre représentant un schéma est donc élément d'un FA. C'est pourquoi de manière informelle, lorsqu'il n'y a pas d'ambiguité possible, nous utilisons le terme **élément** pour désigner un noeud quelconque de schéma. On parle par exemple de l'élément "act" pour le FA "étudiant".

IV.1.4 Conclusion

Dans les deux premières parties de cette section, nous avons donné les définitions de schéma et d'élément, de domaine pour un schéma, d'occurrence et de valeur d'élément comme support pour la définition d'un FA. Pour les besoins de ce chapitre, nous nous avons introduit une définition simplifiée d'un FA : le couple (S, Δ), S étant un schéma et Δ un ensemble d'occurrences. Nous avons montré qu'un schéma d'élément de FA et une valeur d'élément de FA forment à leur tour un FA. Ce FA peut avoir des éléments et on peut lui appliquer les définitions et les notations présentées ici.

Les FA_opérations d'interrogation présentées en section IV.2, peuvent être appliquées de manière récursive, c'est à dire au FA que nous voulons traiter mais également aux FA définis par ses éléments. Cette approche est comparable à celle proposée dans [SCH 84] pour une algèbre de relations non en première forme normale. Nous ne nous limitons pas à présenter une opération et sa syntaxe pour son application à un noeud de l'arbre mais nous montrons comment elle s'applique récursivement en introduisant cette possibilité directement au niveau de la syntaxe. Chaque opération est définie par le schéma et la valeur du FA résultat de son application. Le tableau suivant résume les opérations que nous proposons pour l'expression de requêtes sur des FA.

INTERROGATION	SIGNIFICATION
sélection	sélection d'occurrences de FA
union	union de deux FA
intersection	intersection de deux FA
différence	différence de deux FA
renommer	renommage de FA et/ou de ses éléments
produit	produit cartésien de deux FA
élaguer	projection sur un FA
grouper	groupement de données
éclater	éclatement de données
ordonner	ordonner des données

FA_opérations d'interrogation des FA

Pour les opérations de mise à jour des FA, nous proposons les opérations du tableau ci-dessous. Ces opérations ne sont pas définies de manière récursive. Nous considérons une occurrence comme un arbre dans lequel il faut naviguer pour atteindre les éléments à mettre à jour.

MISE A JOUR	SIGNIFICATION
créer	création d'une occurrence de FA
modifier	modification d'une occurrence de FA
détruire	destruction d'une occurrence de FA
affecter	affectation de valeur à un élément d'occurrence
supprimer	suppression d'une valeur d'élément d'occurrence

FA_opérations de mise à jour des FA

Pour la présentation de l'ensemble des FA_opérations, nous nous appuyons sur le FA "étudiant" que nous avons présenté dans l'exemple 4.4. Pour plus de commodité nous présentons la valeur de ce FA sous forme d'une table (Figure 4.3).

× étudiant

nom	année	× adresse			+ situation	* activités	* notes 3
		rue	code	ville			
balou	1960	10 av couchetard	38100	Grenoble	marié_e:[nom_jf: ∂]	act: ϕ	<note:10 note:06 note:08 >
geser	1958	[?]			célib: □	act: [noma:ski, nban:15] act: [noma:delta, nban:3] act: [noma:danse, nban:15] act: [noma:théâtre, nban:3]	<note:13 note:13 note:13 >
dobey	1956	50 cr J. Jaurès	38100	Grenoble	célib: □	act: [noma:photo, nban:3] act: [noma:ski, nban:11]	<note:10 note:11 note:07 >
nay	1956	[?]			marié_e:[nom_jf: roy]	act: [noma:golf, nban:5]	<note:12 note:14 note: ? >

Figure 4.3 valeur du FA "étudiant"

IV.2 Interrogation de FA

Comme pour les langages algébriques, nous considérons que l'information que nous recherchons peut s'exprimer sous forme d'un FA obtenu par application successives d'opérations (unaire ou binaire) dont les opérandes sont des FA. Comme nous l'avons dit les opérations peuvent s'appliquer à tous les niveaux d'un arbre représentant un schéma de FA. De ce fait, le FA opérande d'une opération peut être : (1) le FA résultat d'une opération, (2) le FA valeur d'élément dans le contexte d'une occurrence, (3) un FA stocké dans la base. Dans ces deux derniers cas de figures, il faut désigner le FA. Le nom d'un FA (ou un élément) est donc une **expression élémentaire** de l'algèbre de FA.

Une valeur Δ d'un FA Γ peut être vue comme une occurrence $\Gamma : \Delta$ pour le schéma $\Gamma : \{sch(\Gamma)\}$. Pour être rigoureux, toute opération OP sur Γ devrait être décrite en termes de la syntaxe définie pour OP lorsque celle-ci s'applique à une occurrence d'un schéma *ensemble*. Par exemple pour le FA "étudiant", on devrait écrire pour chaque opération OP : OP ("étudiant", OP(étudiant, ...)). Ce qui

alourdit l'écriture des opérations en introduisant un niveau supplémentaire que l'on peut considérer comme implicite.

Nous distinguons les opérations qui préservent le schéma du FA : sélection, union, intersection, différence et ordonner (section IV.2.1) et celles qui le transforment : renommer, produit cartésien, élaguer, grouper, éclater (section IV.2.2). Nous donnons en conclusion (section IV.2.3) d'autres opérations pour la transformation de schéma : $sup_{\times}$, sup_{+}, sup^{*}. La définition d'un opération consiste à donner le "FA" résultant de son application, c'est à dire le schéma et la valeur de ce FA.

IV.2.1. Opérations ne transformant pas le schéma

1- Sélection

Cette opération s'applique à un FA et sélectionne les occurrences de ce FA qui satisfont une propriété donnée.

Soit Γ (S, Δ) un FA, S = A: *da*, le schéma de ce FA. Une sélection sur ce FA s'écrit :

SEL (Γ) ≡ **sélection** (A, ES).

ES est une *Expression de Sélection* dont la forme dépend de S.

Avant de présenter les différentes syntaxes possibles pour ES, nous donnons la définition d'une Expression de Sélection élémentaire.

Expression de Sélection élémentaire :

Définition :

Soit A1, A2 des éléments du FA Γ et S1 = Sch_élem(A1), S2 = Sch_élem(A2). Une expression élémentaire ESe a l'une des trois formes suivantes :

(1) comp (A1, v1),

(2) comp (A1, A2)

(3) $\nexists$ A1 pour $S1_{nonapplicable}$ et $\exists?$ A1 pour $S1_{nullallowed}$

comp est un comparateur parmi $=, \neq, >, \geq, <, \leq, \in, \subset, \subseteq, \supset, \supseteq$.
v1 est une constante : v1 $\in$ dom (S1) ou v1 $\subseteq$ dom (S1).

Les comparaisons de valeur doivent remplir les conditions suivantes :

- *comp* est un comparateur défini sur S1 et S2
- S1 et S2 sont compatibles

Dans l'annexe 2, on trouve les règles de compatibilité de schémas et la table des comparateurs pour chaque schéma.

La notion de satisfaction d'une expression de sélection élémentaire par une occurrence O de l' est définie par :

O $\models$ comp (A1, v1) si et seulement si comp(O.A1, v1)
O $\models$ comp (A1, A2) si et seulement si comp(O.A1, O.A2)
O $\models \nexists$ A1 si et seulement si O.A1 $\in$ {A1:∂, A1:[∂], A1:<∂>, A1:{∂} }
O $\models \exists?$ A1 si et seulement si O.A1 $\in$ { A1:?, A1:[?], A1:<?>, A1:{?} }

Exemples d'ESe sur le FA "étudiant" :

=(nom, *toto*),
≤(année, *1965*),
$\exists?$ adresse,
=(activités, {act:ϕ}) ou =(activités, {ϕ})
$\in$(notes, { <note:*10*, note:*06*, note:*08*>, <note:*07*, note:?, note:*15*> }

On constate que dans une ESe de la forme comp(A1, v1), on omet le nom dans la valeur v1 car celui-ci est explicite de par le contexte.

Expression de sélection

Nous énonçons ici les diverses formes d'une expression de sélection **ES** pour une opération de **sélection** sur l' (S, Δ).

Si S est de la forme

- A:D, ES := ESe - forme (1) ou (3) -

- A : [(A1 : *d1*), (A2 : *d2*), ..., (An : *dn*)],
  ES := ESe | not '(' ES ')' | ou '(' ES ',' ES ')' | et '(' ES ',' ES ')'
  | SEL (Γ_{Ai})
  On rappelle que Γ_{Ai} est le FA défini par le schéma de l'élément Ai et la valeur de cet élément (pour une occurence O de Γ).

  Exemples de sélection :

  sélection (étudiant, ∈(nom, { *balou*, *geser*, *nay* }))
  sélection (étudiant, ∃? adresse)) sélectionne les occurrences de "étudiant" pourlesquelles l'élément "adresse" a une valeur inconnue.
  sélection (étudiant, = (activités, {ϕ}) répond à la question : quels sont les étudiants qui non pas d'activités.

- A : (A1 : *d1*) | (A2 : *d2*) | ... | (An : *dn*),
  ES := not '(' ES ')' | ou '(' ES ',' ES ')' | et '(' ES ',' ES ')'
  | Ai
  | SEL (Γ_{Ai})

  Exemples de sélection :

  sélection (étudiant, **sélection** (situation, **sélection** (marié, ∄ nom_jf)))
  sélection (étudiant, **sélection** (situation, not(marié))

- A : <A1 : *d1*>$_n$
  ES := not '(' ES ')' | ou '(' ES ',' ES ')' | et '(' ES ',' ES ')'
  | ∀ SEL (Γ_{A1})
  | ∃ SEL (Γ_{A1})
  | SEL($\Gamma_{A<i>}$)
  | ESe pour l'élément A<i>

  Exemples de sélection :

  sélection (étudiants, (**sélection** (notes, ∀ **sélection** (note, = (note, *13*)))))
  sélection (étudiant, **sélection** (notes, ∈ (notes<2>, {*10*, *12*, *13*, *14*, *16* })))
  sélection (étudiant, **sélection** (notes, ∃? note<3>)) sélectionne les occurrences d'"étudiant" pour lesquelles la troisième note de la liste "notes" est inconnue.

- A : {A1 : *dl*}
  ES := not '(' ES ')' | ou '(' ES ',' ES ')' | et '(' ES ',' ES ')'
  | ∀ SEL(Γ$_{A1}$)
  | ∃ SEL(Γ$_{A1}$)
  | SEL (Γ$_{A1}$)

  Exemple :

  sélection (étudiant, **sélection** (activités, ∃ **sélection** (act, > (nban, *10*))))
  sélection (étudiant, **sélection** (activités, sélection (activité, > (nban,10))))
  remplace l'ensemble "activités" E de chaque occurrence d'"étudiant", par un ensemble contenant seulement les activités de E pour lesquelles *nban* > *10*.

Définition de la sélection :

SEL(Γ) ≡ sélection (A, ES) est définie par :

- sch (SEL(Γ)) = sch(Γ)

- ES := ESe,

  val (SEL(Γ)) = { O ∈ val(Γ) et O ⊨ ESe }

- ES := et (ES1, ES2) - ES1, ES2 sont des expressions de sélection -

  val (SEL(Γ)) = val (sélection (A, ES1)) ∩ val (sélection (A, ES2))

- ES := ou (ES1, ES2),

  val (SEL(Γ)) = val (sélection (A, ES1)) ∪ val (sélection (A, ES2))

- ES := not (ES1),

  val (SEL(Γ)) = val (Γ) − val (sélection (A, ES1))

- ES := Ai,

  val (SEL(Γ)) = { O ∈ val (Γ) | O.Ai est défini }

- ES := SEL(Γ$_{A<i>}$)

  val (SEL(Γ)) = { O ∈ val(Γ) | val (SEL(Γ$_{A<i>}$)) ≠ {ϕ} }

- ES := ∃ sélection (Γ_{A1}, ES)

  val (SEL(Γ)) = { O ∈ val(Γ) | ∃ I ∈ O et val(SEL (Γ_{A1}) ≠ {ϕ}
  pour val(Γ_{A1}) = I}

- ES := ∀ sélection (Γ_{A1}, ES)

  val (SEL(Γ)) = { O ∈ val(Γ) | ∀ I ∈ 0, val(SEL (Γ_{A1}) ≠ {ϕ}
  pour val(Γ_{A1}) = I}

- ES := SEL(Γ_{Ai})

  S = A: [A1:*d1*, A2:*d2*, ..., An:*dn*],
  val (SEL(Γ)) = { O ∈ val(Γ) | val (SEL(Γ_{Ai})) ≠ {ϕ} }

  S = A: (A1:*d1*) | (A2:*d2*) | ...| (An:*dn*),
  val (SEL(Γ)) = { O ∈ val(Γ) | val (SEL(Γ_{Ai})) ≠ {ϕ}, pour val(Γ_{Ai}) défini }

  S = A : { A1:*d1*}
  val (SEL(Γ)) = { A: {I1, I2, ..., In} | ∃ O ∈ val(Γ), ∃ I_i ∈ O
  et val (SEL(Γ_{A1})) ≠ {ϕ}, pour val(Γ_{A1}) = I_i }

La Figure 4.4 donne le résultat de l'application de la sélection ci-dessous au FA "étudiant".

SEL1 ≡ **sélection** (étudiant, et (**sélection** (situation, célib),
sélection (activités, ∃ **sélection** (act, et(= (noma, *ski*), > (nb_an, *10*))))))

correspond à la question : *quel sont les étudiants célibataires et "pratiquant" le ski depuis plus de 10 ans.*

× étudiant

nom	année	× adresse			+ situation	* activités	$*_3$ notes
		rue	code	ville			
geser	1958	[?]			célib: □	act: [noma:ski, nban:15] act: [noma:delta, nban:3] act: [noma:danse, nban:15] act: [noma:théâtre, nban:3]	<note:13 note:13 note:13 >
dobey	1956	50 cr J. Jaurès	38100	Grenoble	célib: □	act: [noma:photo, nban:3] act: [noma:ski, nban:11]	<note:10 note:11 note:07 >

Figure 4.4 : Résultat de la sélection SEL1 sur "étudiant".

Limitations

Une ESe autorise l'expression d'une comparaison entre éléments d'un même schéma. Si l'on se réfère à l'arbre représentant un schéma, on ne peut donc pas comparer des éléments fils de noeuds différents. Cette limitation est liée au système de nommage choisi. Nous donnons en conclusion (section I.1.4) une extension de la sélection pour pallier cet inconvénient.

2- Opérations ensemblistes

Un FA est un ensemble d'occurrences. Il est donc possible d'appliquer des opérations ensemblistes à un FA. Nous avons l'union, l'intersection et la différence.

syntaxe : **union** (F1, F2)
intersection (F1, F2) F1, F2 sont des FA
différence (F1, F2)

Ces opérations peuvent être appliquées si et seulement si :
sch(F1) = sch(F2) et dom(sch(F1)) = dom(sch(F2))

Définition :

- **union** (F1, F2)

  sch(union (F1, F2)) = sch(F1)
  val(union (F1, F2)) = val(F1) ∪ val(F2)

- **intersection** (F1, F2)

  sch(intersection (F1, F2)) = sch(F1)
  val(intersection (F1, F2)) = val(F1) ∩ val(F2)

- **différence** (F1, F2)

  sch(différence (F1, F2)) = sch(F1)
  val(différence (F1, F2)) = val(F1) - val(F2)

3- Ordonner

Cette opération s'applique à un FA et permet d'ordonner ses occurrences selon un certain critère.

Par exemple, pour le FA "étudiant", on veut pouvoir ordonner ses occurrences selon le nombre d'activités pratiquées. Pour chaque occurrence, on veut par exemple obtenir l'ensemble des activités par nombre d'années de pratique décroissant ou bien les notes par ordre croisant.

Pour une opération **ordonner**, on a besoin d'opérateurs et de mots-clés particuliers :

- count : compte le nombre d'objets dans une liste ou un ensemble,
- asc : croissant
- des : décroissant

Soit Γ (S, Δ) et S = A: *da*, une opération **ordonner** sur ce FA s'écrit:

ordonner (Γ) $\equiv$ ordonner (A: CR_Γ)

CR_Γ est l'expression d'un critère sur Γ ayant la forme suivante :

CR_Γ := asc (LE) | des (LE),
| ordonner (Γ_{Ai}), Ai $\in$ Elem(Γ) et Sch_élem(Ai)
non de la forme Ai:D, Ai:<Si>$_n$ ou Ai:{Si}
| ordonner (Ai, $CR_{\Gamma A1}$), Ai $\in$ Elem(Γ)
et Sch_élem(Ai) = Ai: <A1:*dl*>$_n$ ou Ai : {A1:*dl*}

LE := Ai, Ai $\in$ Elem(S) et Sch_élem (Ai) = Ai:D.
| count (Ai), Ai $\in$ Elem(S) et Sch_élem(Ai) de la forme
Ai:<Sp>$_n$ ou Ai:{Sp}
| LE (','LE)$^+$

Définition de ordonner

sch(ordonner(Γ)) = sch(Γ)

val(ordonner(Γ)) = val(Γ), les occurences ont été arrangées selon le critère CR_Γ

Pour ordonner les occurrences selon des valeurs d'éléments (de la liste LE) croissantes ou décroissantes, on procède de façon traditionnelle en comparant ces valeurs. Cette comparaison se fait selon les relations d'ordre (associées à **asc** et **des**) définies sur le domaine auquel appartiennent les valeurs. Dans les domaines, nous pouvons avoir des valeurs nulles : ϕ, ∂, ? et il faut redéfinir la sémantique de **asc** et **des** par rapport à ces valeurs (voir annexe 2).

Exemples :

ordonner (étudiant : **asc (count** (activités)))
ordonner (étudiant : **ordonner** (notes : asc(note)))
ordonner (étudiant : **ordonner** (activités : desc (nban)))

Le FA résultat de cette dernière opération est donnée en Figure 4.5.

× étudiant

nom	année	× adresse			+ situation	* activités	$*_3$ notes
		rue	code	ville			
balou	1960	10 av couchetard	38100	Grenoble	marié_e:[nom_jf: ∂]	act : ϕ	<note:10 note:06 note:08 >
geser	1958	[?]			célib: □	act: [noma:ski, nban:15] act: [noma:danse, nban:15] act: [noma:delta, nban:3] act: [noma:théâtre, nban:3]	<note:13 note:13 note:13 >
dobey	1956	50 cr J. Jaurès	38100	Grenoble	célib: □	act: [noma:ski, nban:11] act: [noma:photo, nban:3]	<note:10 note:11 note:07 >
nay	1956	[?]			marié_e:[nom_jf: roy]	act: [noma:golf, nban:5]	<note:12 note:14 note:? >

FA résultat de **ordonner** (étudiant : **ordonner**(activités :desc (nban)))

Figure 4.5

IV.2.2 Opérations transformant le schéma

1- renommer

L'opération **renommer** s'applique à un FA Γ et réalise le renommage du FA et/ou de ses éléments. Ce qui signifie que l'on renomme le schéma et les occurrences du FA. Elle s'écrit :

renommer(Γ) ≡ renommer (ER)
ER est une expression de renommage sur Γ. Elle prend différentes formes selon sch(Γ).

Définition de renommer :

Le résultat d'une opération renommer(Γ) est un FA Γ'' si et seulement si card (Elem(Γ)) = card (Elem(Γ'')).

Pour chaque forme de sch(Γ), nous donnons la forme de ER et la définition de renommer(Γ).

Soit le FA Γ (S, Δ),

- S = A:*da*, ER := A → A'

  sch (renommer(Γ)) = A':*da*

  val (renommer(Γ)) = { A':va | $\exists$ O $\in \Gamma$ et O = A:va }

- S = A: [S1, S2, ..., Sn], ER := (A | A → A') : LR
  LR est une suite d'opérations **renommer**(Γ_{Ai}), Ai $\in$ Elem(Γ).

  Considérons LR := renommer(Γ_{Ai})
  sch (renommer(Γ)) = A/A': [S1, ..., Si-1, sch (renommer(Γ_{Ai})), Si+1, ..., Sn]

  O = A: [I_1, I_2, ...,I_i, ...,I_n] $\in$ val(Γ)
  $\Rightarrow$ A/A' : [I_1, I_2, ..., val (renommer(Γ_{Ai})), ..., I_n] $\in$ val (renommer(Γ))

- S = A: S1 | S2 | ...| Sn, ER := (A | A → A') : LR
  LR est une suite d'opérations **renommer**(Γ_{Ai}), Ai $\in$ Elem(Γ).

  Considérons LR := renommer(Γ_{Ai})
  sch (renommer(Γ)) = A/A': S1 |...| Si-1 | sch (renommer(Γ_{Ai})) | Si+1 |...| Sn

  O = A: I $\in$ val(Γ)
  A/A': a $\in$ val (renommer(Γ)) si I $\neq$ Ai:*vi*
  $\Rightarrow$
  A/A': val (renommer(Γ_{Ai})) $\in$ val (renommer(Γ)) si I = Ai:*vi*

- S = A: $<S1>_n$, ER := (A | A → A') : renommer(Γ_{A1})

  sch (renommer(Γ)) = A/A': sch (renommer(Γ_{A1}))

  O = A: $<I_1, I_2, ..., I_k>$ $\in$ val(Γ), k $\leq$ n
  $\Rightarrow$ A/A': $<I_1', I_2', ..., I_k'>$ $\in$ val (renommer(Γ))
  où I_i' = val (renommer(Γ_{A1})) pour val(Γ_{A1}) = I_i

- S = A : {S1}, ER := (A | A → A') : renommer(Γ_{A1})

  sch (renommer(Γ)) A/A': sch (renommer(Γ_{A1}))

  O = A: $\{I_1, I_2, \ldots, I_n\} \in$ val(Γ)
  ⇒ A/A': $\{I_1', I_2', \ldots, I_n'\} \in$ val (renommer(Γ))
  où I_i' = val (renommer(Γ_{A1})) pour val(Γ_{A1}) = I_i

Exemple :

renommer (étudiant : [**renommer** (nom → nom_prénom),
renommer (activités → activité : [**renommer** (nban → années)]))

2- produit cartésien

Cette opération définit le produit cartésien de deux FA Γ1 et Γ2 si et seulement si sch(Γ1) ≠ sch(Γ2). Il suffit que leurs noms soient différents. On peut toujours ce ramener à cette hypothèse en réalisant un opération **renommer** sur l'un des FA.

syntaxe : **produit** (Γ1, Γ2)

Définition :

Soit Γ1 = (A1 : *d1*) et Γ2 = (A2 : *d1*),

sch (produit(Γ1, Γ2)) = A1A2: [sch(Γ1), sch(Γ2)]

val (produit(Γ1, Γ2)) = {O | ∃O1 ∈ Γ1 et ∃O2 ∈ Γ2) et O.A1=O1 et O.A2=O2}

Le produit cartésien de deux FA correspond au produit cartésien entre deux relations. Nous ne donnons pas d'exemple.

3- Elaguer

L'opération élaguer s'applique à un FA Γ (S, Δ), S = A:*da*. Elle correspond à l'opération de projection de l'algèbre relationnelle.

élaguer (Γ) ≡ **élaguer** (A : EL)
où EL est une expression d'élagage avec la syntaxe suivante :
EL := Ai, Ai ∈ Elem(Γ) - si card(over(S)) >1
| élaguer (Γ_{Ai}), Ai ∈ Elem(Γ)
| EL (, EL) $^{+}$, liste d'expressions d'élagage

Remarque :

Si S = A : <A1 : *dl*>$_n$ ou S = (A : {A1 : *dl*}), on ne peut pas écrire : élaguer (A : A1) qui aurait pour effet de détruire A1 et de rendre S indéfini. Ceci montre que si card(over(S)) = 1, l'opération élaguer ne peut prendre qu'une seule forme : élaguer (A, élaguer(Γ_{A1})).

Définition :

- EL := Ai

  1. S = A: [S1, S2, ..., Sn] et Si = (Ai: *di*)

     sch(élaguer (Γ)) = A: [S1, S2, ..., Si-1, Si+1, ...Sn]

     O = A: [I_1, I_2, ...,I_i, ...,I_n] ∈ val(Γ)
     ⇒ A: [I_1, I_2, ..., I_{i-1}, I_{i+1}, ...,I_n] ∈ ∈ val (élaguer(Γ))

  2. S = A : S1 | S2 | ... | Sn et Si = (Ai: *di*)

     sch(élaguer (Γ)) = A: S1 | S2 | ...| Si-1 | Si+1 | ... | Sn

     O = A: I ∈ val(Γ) et I ≠ Ai:vi ⇒ O ∈ val (élaguer(Γ))

- EL := **élaguer (Γ_{Ai})**,

  1. S = A : [S1, S2, ..., Sn] et Si = (Ai: *di*)

     sch(élaguer (Γ)) = A: [S1, S2, ..., Si-1, sch(élaguer(Γ_{Ai})), Si+1, ...Sn]

     O = A: [I_1, I_2, ...,I_i, ...,I_n] ∈ val(Γ)
     ⇒ A: [I_1, I_2, ..., val(élaguer(Γ_{Ai})), ...,I_n] ∈ ∈ val (élaguer(Γ))

  2. S = A : S1 | S2 | ... | Sn et Si = (Ai: *di*)

     sch(élaguer (Γ)) = A: S1 | S2 | ...| Si-1 | sch(élaguer(Γ_{Ai})) | Si+1 | ...Sn

     O = A: I ∈ val(Γ)
     O ∈ val (élaguer (Γ)) si I ≠ Ai:vi
     ⇒
     A: val(élaguer(Γ_{Ai})) ∈ val(Γ) si I = Ai:vi

  3- S = A : <S1>$_n$ S1 = (A1: *dl*) /* EL := élaguer (Γ_{A1}) */

sch(élaguer (Γ)) = A: <sch(élaguer(Γ_{A1}))>$_n$

O = A: <I_1, I_2, ...,I_k> $\in$ val(Γ), k $\leq$ n
$\Rightarrow$ A: <I_1',I_2', ...,I_k'> $\in$ val (élaguer(Γ))
où I_i' = val (élaguer(Γ_{A1})) pour val(Γ_{A1}) = I_i

4- S = A : { S1 }, S1 = (A1: *dl*) /* EL := élaguer (Γ_{A1}) */

sch(élaguer (Γ)) = A: {sch(élaguer(Γ_{A1}))}

O = A: { I_1, I_2, ...,I_k} $\in$ val(Γ)
$\Rightarrow$ A: { I_1',I_2', ...,I_k'} $\in$ val (élaguer(Γ))
où I_i' = val (renommer(Γ_{A1})) pour val(Γ_{A1}) = I_i

- EL := EL1, EL2

élaguer (A : EL1, EL2) = élaguer (élaguer (A :EL1) : EL2)

Exemples : **élaguer** (étudiant : **élaguer** (situation : marié_e, divorcé_e)
élaguer (étudiant : **élaguer** (activités : **élaguer** (nban)))

La Figure 4.6 donne le FA résultat cette dernière opération.

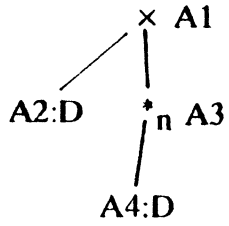
× étudiant

nom	année	× adresse			+ situation	* activités	$*_3$ notes
		rue	code	ville			
balou	1960	10 av couchetard	38100	Grenoble	marié_e:[nom_jf: ∂]	act: ϕ	<note:10 note:06 note:08 >
geser	1958	[?]			célib: □	act: [noma:ski] act: [noma:delta] act: [noma:danse] act: [noma:théâtre]	<note:13 note:13 note:13 >
dobey	1956	50 cr J. Jaurès	38100	Grenoble	célib: □	act: [noma:photo] act: [noma:ski]	<note:10 note:11 note:07 >
nay	1956	[?]			marié_e:[nom_jf: roy]	act: [noma:golf]	<note:12 note:14 note: ? >

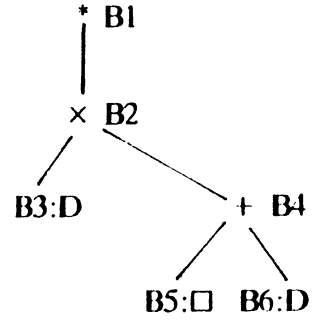
Figure 4.6 : **élaguer** (étudiant : **élaguer** (activités : **élaguer** (nban)))

Remarques sur les schémas obtenus après élagage

Considérons les schémas suivants :

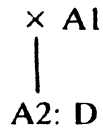


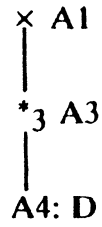
sch(Γ1)



sch(Γ2)

L'opération d'élagage conduit à des arbres "bizarres". Par exemple après élaguer (A1, A3) et élaguer (A1, A2) pour sch(Γ1), on obtient les arbres suivants :





On aimerait pouvoir transformer ces deux schémas pour obtenir des schémas plus simples : A2 : D et A3 : $<A4 : D>_n$ et réaliser de nouvelles opérations sur les FA ainsi obtenu. Pour cela, il faut des opérations de restructuration de schéma autorisant par exemple la transformation de A1 : [A2:D] en A2:D.

Si on enlève le noeud B5:□ à sch(Γ2), on obtient le schéma B1: { B2: [B3:D, B4: B6:D] } que l'on peut aimerait simplifier en B1: { B2: [B3:D, B6:D] }. Nous présentons en section IV.2.3, quelques opérations de transformation de schémas.

4- Regroupement de données

Cette opération correspond à l'opération "nest" de l'algèbre relationnelle étendue [FT 83, SCH 84]. Nous rappelons les effets d'une telle opération par un exemple :

R

A1	A2	A3	A4
1	3	4	4
1	3	5	3
1	3	6	5
2	3	6	7
2	3	4	4
2	4	7	5

nest
(A3, A4):A5

R'

A1	A2	A5	
		A3	A4
1	3	4	4
		5	3
		6	5
2	3	6	7
		4	4
2	4	7	5

On voit donc que l'opération nest(A3, A4):A5 transforme R(A1, A2, A3, A4) en R'(A1, A2, A5), relation N1FN. Pour chaque couple de valeur A1, A2 on regroupe les valeurs de A3, A4. On constate que (A1, A2) forme alors une clé pour R' et qu'à chaque paire (A1, A2) correspond un ensemble de paires (A3, A4).

Si l'on se réfère à notre formalisme, le schéma de FA correspondant à R est

R : [A1:*d1*, A2:*d2*, A3:*d3*, A4:*d4*]

et celui correspondant à R' est:

R' : [A1:*d1*, A2:*d2*, A5 : { X : [A3:*d3*, A4:*d4*] }]

On constate que pour notre formalisme, nous avons besoin d'un second nom (X) identifiant un élément de l'ensemble crée.

L'opération **grouper** que nous proposons s'applique à des FA Γ dont le schéma et de la forme A: [S1, S2, ..., Sn].

Définition

Soit Γ=(S, Δ) un FA et S = A: [S1, S2, ..., Sn], Si= Ai:*di*. Soit X1, X2 ∈ **A** et X1,X2 ≠ Ai pour tout Ai ∈ Elem(Γ).

grouper(Γ) ≡ **grouper** (A, X1 : LG)

où LG := Ai, Ai ∈ Elem(Γ) (1)

| LG (, LG)$^{+}$ [X2] (2)

- LG = Ai, grouper(Γ) ≡ grouper (A, X1 : Ai)

sch (grouper(Γ)) = A : [S1, S2, ..., Si-1, X1 : {Si}, ..., Sn]

On pose : Y = A1, A2, ..., Ai-1, Ai+1, ..., An
Γ'' = élaguer (A : Ai)

val (grouperΓ)) =
{ t | ∃ O ∈ val(Γ''))
et t.Aj = O.Aj, ∀ Aj ∈ Elem(Γ'')
et t.X1 = { r | ∃ O' ∈ élaguer (sélection (A, ES) : Y) et r.Ai = O'.Ai }
}

ES := et(=(A1,t.A1), =(A2,t.A2),..., =(Ai-1,t.Ai-1),...,
=(Ai+1,t.Ai+1),..., =(An,t.An))

- LG = Ai, Ai+1 [X2] - par exemple -

sch (grouper(Γ)) = A : [S1, S2, ..., Si-1, X1: { X2: [Si, Si+1] }, Si+2, ..., Sn]

On pose : Y = A1, A2, ..., Si-1, Si+2, ..., An
Γ'' = élaguer(A :Ai, Ai+1)

val (grouper(Γ, LG)) = { t | ∃ O ∈ val (Γ'')
et t.Aj = O.Aj, ∀ Aj ∈ Elem(Γ'')
et t.X1 = val (renommer (élaguer (sélection (A, ES) : Y) → X2)) }
}

ES := et (=(A1, t.A1), =(A2, t.A4), =(Ai-1, t.Ai-1),
=(Ai+2, t.Ai+2),..., =(An, t.An))

La Figure 4.8 donne le résultat du FA "étudiant" après application de l'opération : **grouper** (étudiant, X1 : nom, adresse, situation, activités, notes [X2])

Remarque :

L'écriture de l'opération **grouper** peut se simpifier. Dans la forme (2) de l'expression de renommage LG, on peut omettre le nom X2. Par défaut, on prendra le nom A du FA. **grouper** (A, X1: Ai, Ai+1 [X2]) s'écrit alors simplement **grouper** (A, X1: Ai, Ai+1) et dans la définition de val(grouper(A, LG) il n'est plus nécessaire d'effectuer une opération **renommer**.

× étudiant

année	* X1
1960	X2 : [nom: balou, adresse : [rue : 10 av couchetard, code : 38100, ville : Grenoble], situation : marié_e: [nom_jf: ∂], activités : { act : ϕ}, notes :<note:10 note:06 note:08 >]
1958	X2 : [nom : geser, adresse : [?], célib : □, activités : {act: [noma:ski, nban:15] act: [noma:delta, nban:3] act: [noma:danse, nban:15] act: [noma:théâtre, nban:3] } , notes : <note:13 note:13 note:13 >]
1956	X2 : [nom : dobey, adresse : [rue : 50 cr J.Jaurès, code : 38100, ville : Grenoble], célib : □, activités : { act: [noma:photo, nban:3] act: [noma:ski, nban:11] } , notes :<note:10 note:11 note:07 >] X2 : [nom : ney, adresse : [?], marié_e : [nom_jf : roy], activités : { act: [noma:golf, nban:5] }, notes : <note:12 note:14 note: ? >]

Figure 4.7 : **grouper** (étudiant, X1 : nom, adresse, situation, activités, notes [X2])

comportement des valeurs : ? , ∂ et ϕ

Les occurrences inapplicables, inconnues ou non-informatives posent un problème pour l'évaluation de l'opération **grouper** lorsqu'elles apparaissent comme valeur des éléments vis à vis desquels on regroupe (n'appartenant pas à la liste LG).

Considérons le FA Γ dont le schéma est A:[A1:entier, A2:entier, A3:entier]. Une valeur possible de ce FA est :

{ A: [A1:?, A2:1, A3:1], (a)
  A: [A1:?, A2:1, A3:2], (b)
  A: [A1:∂, A2:1, A3:1],
  A: [A1:∂, A2:1, A3:2],
  A: [A1:1, A2:ϕ, A3:1],
  A: [A1:1, A2:ϕ, A3:2]
}

Puisqu'un élément qui a la valeur nulle ∂ signifie " il ne doit pas exister de valeur pour cet élément", l'élément n'est pas modifiable et on ne peut donc pas remplacer la valeur ∂ par une autre valeur. On peut donc dire que $\partial = \partial$.

Par contre une valeur nulle inconnue ? à l'instant t peut très bien devenir connue à l'instant t + 1. Pour les occurrences (a) et (b) de I', on peut se retrouver dans la situation :

A: [A1:y, A2:1, A3:1] (a)
A: [A1:x, A2:1, A3:2] (b)

x, y ∈ dom(A:entier) et x = y ou bien x ≠ y

De manière générale, ? ≠ ?. De même pour la valeur ϕ, on admet que $\phi \neq \phi$ puisque qu'on peut passer d'une situation où il n'existe pas de valeur pour un élément à une situation où il existe une valeur.

L'opération **grouper** (A, X1: A3) pour I' conduit au FA de schéma
A: [A1:entier, A2:entier, X1:{A3:entier}] et dont la valeur est :

{ A: [A1:?, A2:1, X1:{A3:1}],
  A: [A1:?, A2:1, X1:{A3:2}],
  A: [A1:1, A2:∂, X1:{A3:1, A3:2}],
  A: [A1:1, A2:ϕ, X1:{A3:1}],
  A: [A1:ϕ, A2:1, X1:{A3:2}]
}

On constate que donc que dans le cas des valeurs ? et ϕ, le regroupement conduit à un ensemble possédant un seul élément.

5- Eclatement des données

On propose **éclater** l'opération inverse de **grouper**. Comme pour l'opération grouper, nous la définissons pour la racine du FA.

Définition

Soit I'=(S, Δ) un FA et S de la forme :

(1) A: {A1:*dl*}

(2) A: [S1, S2, ..., Sn] tel qu'il existe Si ∈ over(S) de la forme Ai : {Si'}

Une opération **éclater** sur Γ s'écrit :

éclater(Γ) ≡ **éclater** (A, LE)
où LE := A si S de la forme (1)
| Ai, si S de la forme (2),
Ai ∈ Elem(Γ) et Sch_élem(Ai) = Ai: {Si'}

- LE = A, S est de la forme A: { A1:*dl* }

  sch (éclater(Γ)) = A1:*dl*

  O = A: { I_1, I_2, ..., I_n } ∈ val(Γ) ⇒ I1, I2, ..., In ∈ val (éclater(Γ))

- LG = Ai, S est de la forme A: [S1, S2, ..., Sn]
  et Sch_élem (Ai) = Ai : {Si'} = Ai: {Ai': *di'* }

  sch (éclater(Γ)) = A : [S1, S2, ..., Si-1, Si', Si+1, ..., Sn]
  si et seulement si Ai' ≠ Aj, pour tout Aj ∈ Elem(Γ)

  On pose : Y = {A1, A2, ..., Ai-1, Ai+1, ..., An}
  val (éclater(Γ)) = { t | ∃ r ∈ val(Γ) et t.Aj = O.Aj, ∀ Aj ∈ Y et t.Ai' ∈ r.Ai }

La Figure 4.8 donne des exemples d'opérations **éclater** et montre en particulier que réaliser un éclatement de données sur un FA n'entraîne pas de perte d'information. Pour les besoins de la présentation, nous avons considéré dans ces FA, des valeurs d'éléments appartenant au domaine des Entiers augmenté des valeurs nulles ?, ∂ et ϕ (domaine formellement impossible pour un schéma).

Propriétés des séquences d'opérations **grouper** et **éclater** :

- **éclater (gouper(Γ))** = Γ si on regroupe par rapport à un seul élément.

  Soit sch(Γ) = A : [S1, S2, ..., Sn] et Si = Ai: *di*,
  sch(grouper(A, X1: Ai)) = A: [S1, S2, ..., Si-1, X1: {Si}, Si+1, ..., Sn] et
  sch(éclater(grouper(A, X1: Ai), X1))) = A: [S1, S2, ..., Si-1, Si, Si+1,
  ..., Sn]) = sch(Γ)

- **éclater (grouper (Γ))** ≠ Γ si l'on regroupe par rapport à plusieurs éléments.

  Soit sch(Γ) = A : [S1, S2, ..., Si, ..., Sn] et Si = Ai: *di*,

  sch(grouper(A, X1: Ai,Ai+1 [X2])) =
  A: [S1, S2, ..., Si-1, X1: {X2: [Si, Si+1]}, ..., An] et

sch(éclater(grouper(A, X1: Ai,Ai+1 [X2], X1))) =
A: [S1, S2, ..., Si-1, X2: [Si, Si+1] , ..., An] ≠ sch(Γ)

On remarque qu'il suffit de supprimer X2 pour obtenir l'égalité entre les deux schémas.

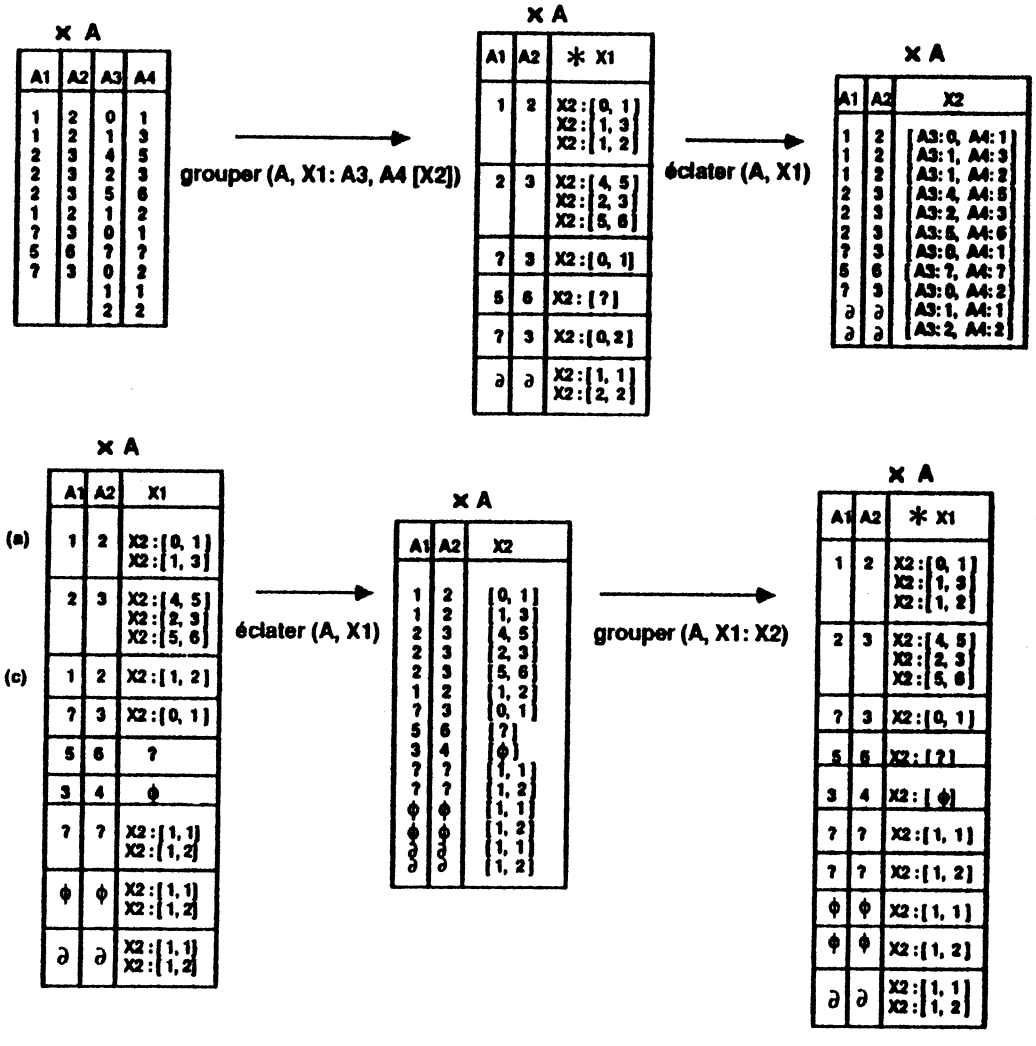


Figure 4.8 Enchaînement d'opérations **grouper** et **éclater**

- **grouper (éclater(Γ)) ≠ Γ.**

Sur la Figure 4.8, on constate que sch(grouper(éclater(Γ)) = Γ mais que val(grouper(éclater(Γ)) ≠ Γ. La présence dans notre algèbre des valeurs nulles ?, ∂ et de la non-information ϕ fait que l'on n'a pas de perte d'information. Par contre, on perd la séparation entre les occurrences (a) et (c) ayant des valeurs identiques pour les éléments A1 et A2. On perd également le regroupement qui existait vis à vis des valeurs ? et ϕ. En effet,

nous avons montré que regrouper par rapport à des éléments avec la valeur ? ou ϕ conduisait à un ensemble contenant une seule valeur. Une solution consiste à associer des indices aux valeurs ? et ϕ [ROT 85]. On a alors les propriétés :

$$\begin{cases} ?_i = ?_j \\ \\ \phi_i = \phi_j \end{cases} \qquad \text{si } i=j$$

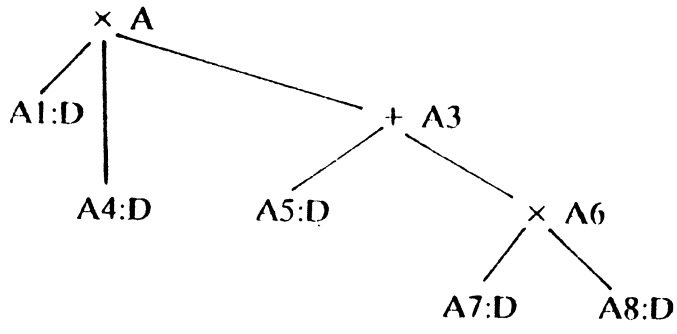
IV.2.3 Conclusion

Nous avons présenté un ensemble d'opérateurs pour l'expression de requêtes sur des FA. Pour les opérateurs sélection, élaguer, renommer et ordonner, nous avons donner leur syntaxe complète de manière à montrer comment il peuvent s'appliquer à un FA et aux FA définis par ses éléments. On a donc pu constater que l'on applique les opérateurs d'une manière récursive. Chaque nouvel appel de l'opérateur correspond en fait à descendre d'un niveau dans l'arbre du FA. Puisqu'on n'admet pas de schéma de FA cyclique, on garantit une profondeur de l'arbre fini et la "récursivité" se termine toujours.

1- Sélection étendue

L'opération de sélection que nous avons proposée permet de faire des comparaisons uniquement entre éléments d'un FA.

Soit le FA avec le schéma suivant :



On ne peut pas écrire par exemple : **sélection** (A, = (A1, A7) (1)

Pour cela, on introduit une nouvelle forme d'expression de sélection élémentaire. Pour le FA Γ, et A1, Ai ∈ Elem (Γ), on définit :

ESe := comp (A1, élaguer (Γ_{Ai})).

O ⊨ comp (A1, élaguer (Γ_{Ai})) ssi comp (O.A1, val(élaguer(Γ_{Ai}))).

L'opération **élaguer** est définie sur un FA correspondant à un élément de Γ donc pour une occurrence particulière O de Γ. On remarque qu'elle est appliquée directement à l'élément Ai sans avoir besoin de préciser qu'il s'agit en fait d'un élagage sur Γ. Ceci est dû au fait que le contexte Γ est défini par l'opération de plus haut niveau, c'est à dire ici la sélection.

Du fait de la présence des noms dans une occurrence, on peut être amener à effectuer un renommage du FA résultat de l'opération élaguer. Ce renommage sera réalisé implicitement.

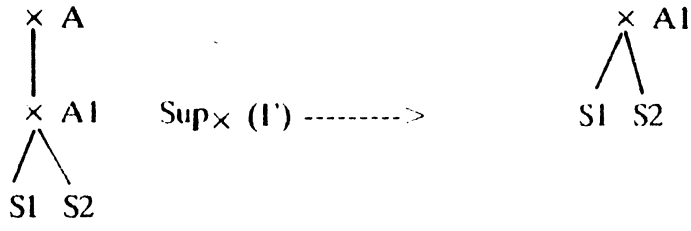
On écrira donc la sélection (1) ci-dessus par :

sélection (A, = (A1, **élaguer** (A3 : A5, **élaguer** (A6: A8)))).

2- Quelques opérations utiles

Les opérations que nous présentons ici sont des opérations de transformation de schémas qui permettre de supprimer un noeud de l'arbre d'un schéma de FA sans détruire l'élément (noeud) qui lui est rattaché. Elles s'appliquent donc à des cas de schémas de FA très particuliers que nous allons examiner.

2-1. opération Sup$_\times$



Cette opération s'applique un FA Γ dont le schéma est de la forme S = A: [A1: *dl*], *dl* = D ou *dl* = [S1, S2, ..., Sn]. Elle correspond à l'opération *cross-collapse* de [ABI 86].

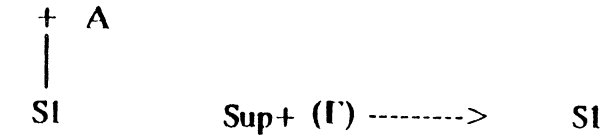
Définition

Soit Γ un FA et sch(Γ) = A: [A1: *dl*], *dl* = D ou *dl* = [S1, S2, ..., Sn].

sch($Sup_{\times}$) = A1: *dl*

val($Sup_{\times}$) = { t | ∃ O ∈ val(Γ) et t = O.A1 }

2-2. opération Sup_{+}



Cette opération s'applique un FA Γ dont le schéma est de la forme S = A: S1.

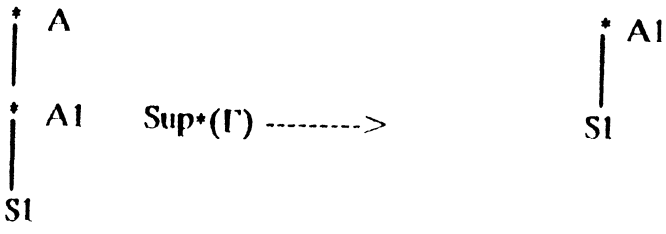
Définition

Soit Γ un FA et sch(Γ) = A: S1 et S1 de la forme A1: *dl*

sch(Sup_{+}) = S1

val(Sup_{+}) = { t | ∃ O ∈ val(Γ) et t = O.A1 }

2-3. opération Sup*



Cette opération s'applique un FA Γ dont le schéma est de la forme S = A:{A1:{S1}}. Elle correspond à l'opération *combinaison* de [ABI 87].

Définition

Soit Γ un FA et sch(Γ) = A: { A1: { S1} },

sch(Sup*) = A1 : {S1}

val(Sup*) = U { I | ∃ O ∈ val(Γ) et I ∈ O }

Les trois opérations que nous venont de décrire s'appliquent à la racine d'un schéma de FA. Elles pourront être utilisées sur un FA, ou bien sur les FA définis par ses éléments dans les opérations de l'algèbre.

Ces opérations sont utiles après une opération **élaguer**. Pour la sélection étendue que nous avons donnée ci-dessus, il serait plus juste d'écrire :

sélection (A, = (A1, **sup**$_{\times}$ (**sup**$_{+}$ (**élaguer**(A3 : A5, **élaguer**(A6: A8))))))

3- De la complétude des opérations.

Pour montrer que les opérations proposées définissent un langage d'interrogation pour FA complet, il faut mesurer le pouvoir d'expression de ce langage par rapport à celui de certains langages pour lesquels une notion de complétude (mesure de la puissance d'un langage) a été définie.

Dans notre cas, on pense qu'il serait possible de comparer nos opérations à celles du calcul relationnel pour lequel il existe une notion de complétude. Il faut alors interpréter un FA en termes de relations. C'est ce type d'approche qui a été choisie pour montrer la complétude du langage sur V-relations [ABI 84]. Dans [ABI 87], on dit qu'il *"semble préférable d'utiliser non pas le calcul relationnel mais directement un calcul pour objets structurés"*. La complétude du calcul proposé ici reste un problème ouvert.

4- Remarques sur les noms dans un schéma de FA

Dans la présentation des opérations, le lecteur a pu noter l'importance de la notion d'élément qui permet une définition récursive des opérations. Le système de nommage que nous avons choisi fait que l'on peut avoir des noms identiques à des niveaux différents de l'arbre représentant un schéma de FA. Ceci présente l'inconvénient d'avoir à réaliser un renommage pour effectuer certaines opérations et d'aboutir à des expressions un peu lourde. On aurait pu imposer d'avoir tous les noms présents dans un schéma distincts. Cette approche aurait simplifiée la syntaxe des opérations mais n'aurait rien changé à leur définition interne.

On doit voir les opérations présentées ici comme un noyau à partir duquel peut être développée une interface conversationnelle (du type langage ou graphique) destinée à l'utilisateur *informaticien* pour l'expression de ses requêtes.

IV.3 Mise à jour d'un FA

Nous présentons dans cette section les opérations pour la mise à jour des FA. Nous n'acceptons pas de mises à jour globales sur un FA. Les opérations : **créer, modifier, supprimer** s'appliquent donc à une occurrence et les opérations : **affecter, effacer** s'appliquent à un élément d'une occurrence. Toutes ces opérations sont des opérations de GIF. Elles sont encapsulées dans des opérations "utilisateurs" de plus haut niveau (opérations spécifiques) auxquelles sont associées la notion de transaction. La confirmation d'une opération spécifique (d'une transaction) provoque la validation des mises à jour réalisées durant cette opération. Ce qui correspond à translater les mises à jour en opérations sur la base de données.

IV.3.1 Création d'une occurrence

Pour créer une occurrence d'un FA Γ (S, Δ), S = A:*da*, on dispose de l'opération :

créer (nom(Γ) , <id_occ>)
< id_occ > est un identificateur interne d'occurrence.

On donne le nom du FA. GIF crée une nouvelle occurrence identifiée par <id_occ>. Il est clair que cela pourra se faire si et seulement si GIF dispose de la signification du FA (SCHEMA_FA).

On utilise l'opération **créer** lorsqu'on veut construire une nouvelle occurrence pour un FA. Elle correspond à enregister l'existence d'une nouvelle occurrence de FA et à créer son contexte de création. Les éléments du FA possède la valeur ϕ ou la valeur ? s'ils portent l'option nullallowed.

L'occurrence créée possède un schéma qui n'est pas toujours stable en particulier lorsqu'un des noeuds du schéma est défini avec le constructeur choix. Le schéma d'une occurrence évolue en fonction des choix réalisés. Le choix des schémas et la valuation des éléments terminaux se fait en utilisant les opérations de mise à jour des éléments d'une occurrence, présentés dans la section suivante. Il est clair que ces opérations vont permettent de donner à tout les éléments de l'occurrence un schéma et une valeur reconnue par GIF et conforme à la sémantique du FA. Bien entendu cette valeur peut être une valeur non-informative, inconnue ou inapplicable.

IV.3.2 Modification d'une occurrence.

On accepte pas de modifications globales de formulaires. On retrouve un ensemble de formulaires puis on modifie chaque formulaire. Pour un utilisateur la recherche de formulaires consiste à activer une opération recherche et à spécifier une expression de recherche à partir d'un formulaire type. Cette opération sera traduite en une FA_expression qui permet de retrouver toutes les occurrences de FA vérifiant l'expression de recherche, de les ramener dans GIF et de les faire apparaître comme des formulaires à l'utilisateur. La modification d'un formulaire correspond pour GIF à réaliser l'opération **modifier** sur l'occurrence de FA sous-jacente et à effectuer les mises à jour de ses éléments en fonction des actions de l'utilisateur.

Une modification sur un FA Γ, s'exprime par :

modifier (nom(Γ), <id_occ>)

Lorsqu'on émet cette opération, on donne l'identificateur <id_occ> de l'occurrence. Le contexte de manipulation de l'occurrence est alors rendu actif.

IV.3.3 Destruction d'une occurrence

détruire (nom(Γ), <id_occ>),

détruit l'occurence identifiée par <id_occ> du FA Γ.

Cette opération consiste à marquer l'occurrence <id_occ> comme détruite. Ainsi, on pourra valider une destruction d'occurrence au niveau de la base de données.

Supposons que les occurrences du FA "étudiant" soit identifiées par : *idO1, idO2, id03, id04*. Les opérations suivantes sur le FA "étudiant" conduisent au FA de la Figure 4.10.

détruire (id01);
détruire (id02);
créer (étudiant, id05)

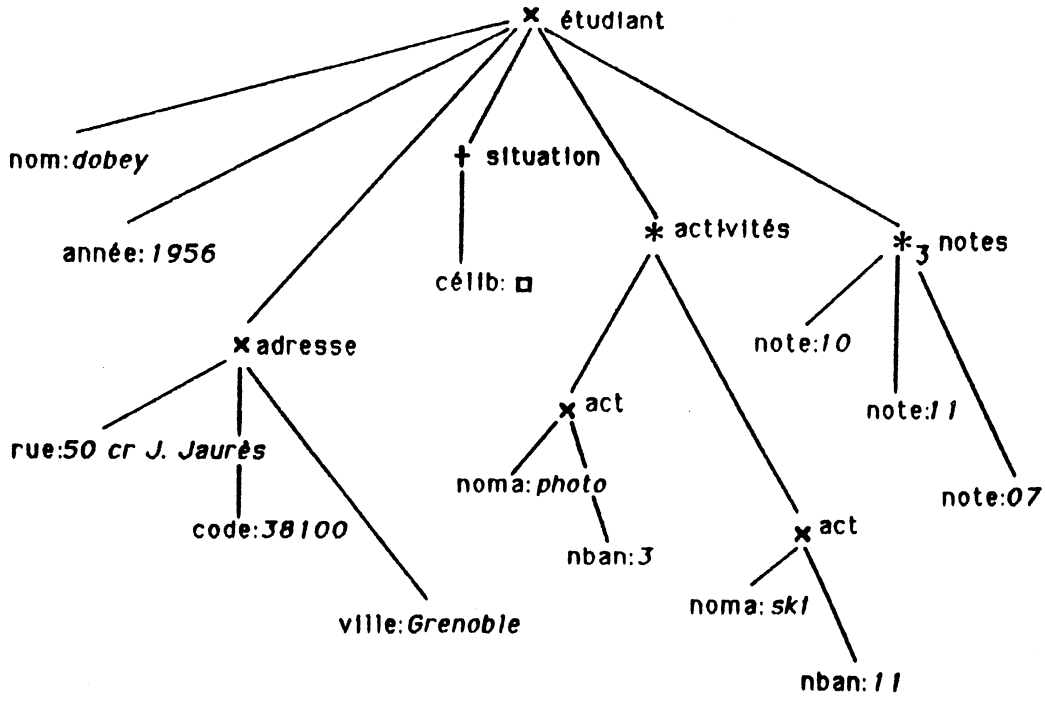
× étudiant

	nom	année	× adresse			+ situation	* activités	*3 notes
			rue	code	ville			
id03	dobey	1956	50 cr J. Jaurès	38100	Grenoble	célib: □	act: [noma:photo, nban:3] act: [noma:ski, nban:11]	<note:10 note:11 note:07 >
id04	nay	1956	[?]			marié_e:[nom_jf: roy]	act: [noma:golf, nban:5]	<note:12 note:14 note:15 >
id05	ϕ	ϕ	[?]			ϕ	ϕ	<?>

Figure 4.9 : FA "étudiant" après destruction et création d'occurrences.

IV.3.4 Mise à jour des éléments d'une occurrence

Considérons l'occurrence *id03* du FA "étudiant" (Figure 4.9). L'arbre suivant représente cette occurrence.



Occurrence *id03* **du FA "étudiant"**

Une mise à jour des éléments de cette occurrence (dans le cas d'une opération **créer** ou **modifier**) consiste à affecter une valeur à un noeud ou bien supprimer la valeur d'un noeud.

Dans un FA on peut avoir des noms identiques à des niveaux différents du schéma. Pour désigner un noeud de l'arbre, la notion d'élément (nom) n'est donc pas suffisante. Contrairement au schéma de FA, on remarque que dans une occurrence, on peut avoir plusieurs branches qui partent d'un noeud $*_n$ (liste) ou * (ensemble). Par contre on a une seule branche qui part d'un noeud + , indiquant le choix qui a été fait.

Avant de modifier la valeur d'un noeud il faut accéder à ce noeud, pour cela on utilise un trajet.

Trajet

Un trajet sert à désigner où se trouve la donnée que l'on va modifier. Ce trajet a la forme d'un chemin dans l'occurrence du FA.

Exemple : *id03*.adresse.ville retrouve la donnée "ville:*grenoble*".

Définition

Soit le FA I' (S, Δ), un trajet a la forme : **<id_occ> [. <chemin(S)>]** .[1]

où <id_occ> est un identificateur d'occurrence
<chemin(S)> est l'expression d'un chemin dans le schéma S.

On définit un chemin de manière récursive par :

- S = A:D ou S= A:□, chemin(S)=A
- S = A: |S1, S2, ..., Sn|, Si = Ai:*di*,

  chemin (S) = Ai | Ai.chemin(Si)
- S = A: S1 | S2 | ... | Sn et Si = Ai:*di*,

  chemin (S) = Ai | Ai.chemin (Si)

[1] Pour expliciter la syntaxe des opérations, nous utilisons dans la suite de cette thèse les notations suivantes : < I > indique que I doit être donnée, [I] indique que I est optionnel, $\{ I \}^*$ ($\{ I \}^+$) indique que I peut être répété un nombre de fois ≥ 0 (≥ 1).

Une occurrence pour S est de la forme A: **va** et *va* est construite selon l'un des schémas S1, S2, ..., Sn. Il est clair que chemin(S) ne sera pas défini si **va** n'est pas de la forme (Ai:**vi**).

- S = A: $<S1>_n$, S1 = A1:*d1*

  chemin(S) = '<' i '>'. chemin(S1)
  | '<' i '>'.A1 chemin(S1), si S1 = $<S1'>_n$ ou S1 = {S1'}
  | '<' i '>'
  i ∈ **N** et $0<i\leq n$

  On s'intéresse à un seul élément de la liste. On définit donc un chemin linéaire.

- S = {S1},

  chemin(S) = '(' i ')'. chemin(S1)
  | '(' i ')'.A1 chemin(S1), si S1 = $<S1'>_n$ ou S1 = {S1'}
  | '(' i ')'
  i ∈ **N**

  Lorsqu'on veut modifier un élément particulier d'un ensemble, on parcourt l'ensemble et on indique (pointe) cet élément. Du point de vue externe (écran) chaque élément de l'ensemble à une position qui correspond à une identification interne de l'élément. On utilise donc cette position pour décrire le chemin, sachant qu'elle n'a aucun équivalent du point de vue formel. Dans d'autres cas de figure, on pourra parcourir l'ensemble (voir section IV.4).

Exemples de trajets pour l'occurrence *id03* :

id03.notes<2> désigne "note:*11*"
id03.activités(1).nban désigne "nban:*3*"
id03.situation.marié_e est un chemin indéfini pour *id03*
id03.situation désigne *célib*:□.

On constate que chaque donnée désignée est une valeur d'élément. La donnée "note:*11*" est une valeur d'élément *note* du FA *notes*. Chaque trajet (non indéfini) désigne donc une valeur d'élément. Et on peut associer à tout trajet un schéma d'élément. Au trajet *id03*.notes<2> correspond le schéma : *note*:*D*. Au trajet *id03* correspond le schéma "étudiant".

La notion de trajet peut se simplifier si on impose des noms uniques dans un schémas de FA (ce qui est la cas pour le schéma "étudiant"). On peut alors écrire "id.03.rue" au lieu de "id03.adresse.rue" mais par contre, on doit toujours expliciter le trajet lorsqu'on veut mettre à jour les éléments d'un ensemble ou d'une liste.

1- Affecter

L'opération **affecter** s'applique à une valeur d'élément et modifie cette valeur. Sa syntaxe est :

affecter (<trajet>, <valeur>)
<trajet> désigne la valeur d'élément E à modifier
<valeur> ∈ dom(Sch_élem(E)) - {ϕ}

Dans l'expression de <valeur>, on omet le nom de l'élément car celui-ci est explicite par le contexte (trajet).

L'effet de cette opération dépend de Sch_élem(E).

Reprenons notre occurrence *id03* du FA "étudiant".

- **affecter** (*id03*.nom, 'toto') remplace la valeur nom:*dobey* par nom:*toto*
- **affecter**(*id03*.situation, 'marié_e')

  Le schéma de l'élément "situation" est de type **choix**, l'opération **affecter** donne une nouvelle valeur à cet élément, indiquant le choix qui a été fait. Si il existe déjà un choix pour l'élément alors cette opération a pour effet de à supprimer l'ancien choix et de le remplacer par le nouveau choix. On indique donc que pour l'occurrence *id03*, l'élément "situation" prend comme schéma : *situation: marié_e: [(nom_jf:DI)inapplicable]*.

  "nom_jf" est un élément portant l'option nonapplicable. Le système détermine s'il doit être appliqué ou non : comme nous l'avons dit au chapitre III, la description d'un élément avec l'option inapplicable nécessite de donner au FA de la connaissance définissant les conditions d'applicabilité de l'élément. Dans l'expression de cette connaissance, òn peut demander des informations à l'utilisateur. Par exemple, pour "étudiant", on peut avoir la connaissance :

  En cas de choix "marié_e" pour l'élément "situation", demander le sexe de l'étudiant". Si S = F alors affecter à l'élément "nom_jf" la valeur ϕ, sinon affecter à "nom_jf" la valeur ∂. "

Dans le cas où l'élément "nom_jf" est appliqué, on lui donne "momentanément" la valeur ϕ. Cette valeur ne peut pas apparaître dans l'occurrence stockée. En effet une occurrence d'étudiant est acceptée dans la base si et seulement si l'élément "nom_jf" portant l'option nonapplicable a une valeur qui appartient au domaine de son schéma (pas de non-information dans ce domaine).

affecter (*id03*.situation.marié_e.nom_jf, 'maya') sera acceptée si le "nom_jf" est appliqué.

- **affecter**(*id03*.notes, *14*)

  On ajoute la valeur : note:*14* à la liste de *notes*.

  Le schéma de l'élément est de type **list**. <valeur> est une liste de valeurs de la forme '<' v1, v2, vk '>', k ≥ 1. Dans le cas où on a une seule valeur à ajouter, on peut ommettre les signes "<" et ">". L'opération **affecter** ajoute les valeurs v1, v2, vk en fin de la liste de valeurs déjà existante pour l'élément, en respectant les contraintes de la liste. On ne peut ajouter des valeurs que dans la limite de la borne supérieure définie pour la liste.

- **affecter**(*id03*.notes<2>, *14*)

  note:*14* devient la valeur d'élément notes<2>.

  Avec une opération affecter de cette forme, on peut une nouvelle valeur au ième élément d'une liste.

- **affecter**(*id03*.activités, act : [noma: *golf*, nban:*1*])

  Le schéma de l'élément est de type **ensemble**. <valeur> est un ensemble de valeurs de la forme '{' v1, v2, ..., vn '}'. L'opération **affecter** ajoute ces valeurs à l'ensemble de valeurs déjà existant. Si une valeur à ajouter existe déjà dans l'ensemble, elle ne lui est pas affectée. Les parenthèses peuvent êtres omises lorsqu'on ajoute une seule valeur à l'ensemble.

- **affecter**(*id03*.activités(2).nban, *6*)

  donne à l'élément "nban" de la secònde activité ("act") de l'ensemble la valeur nban:*6*. Si cette affectation de valeur fait que l'activité prend une valeur déjà existante dans l'ensemble, on le signale et on demande confirmation de l'opération. S'il y a confirmation alors on réalise une suppression de l'activité en question.

2- Supprimer

Cette opération supprime une valeur d'élément d'occurrence. C'est l'opération duale d'**affecter**. Sa syntaxe est :

supprimer(<expression_sup>)

où <expression_sup> := <trajet>
| <trajet> , '<' v1, v2, ..., vk '>' (1)
| <trajet> , '{' v1, v2, ..., vk '}' (2)
| <trajet>, vi (3)

<expression_sup> peut prendre une des formes (1), (2) ou (3) si le schéma S associé à <trajet> est de de la forme $<S1>_n$ ou {S1}. vi ∈ dom(S1) et vj ∈ dom(S1) pour j ∈ [1...k]

Dans le cas d'un ensemble, l'opération retire la ou les valeurs de l'ensemble. Dans le cas d'une liste, l'opération donne la valeur ϕ ou la valeur ? à l'élément ou aux éléments ayant la ou les valeurs v1, v2, vk. On peut alors se retrouver avec des listes contenant des "trous" (des valeurs ϕ) et on peut vouloir réorganiser la liste. Nous montrons en section IV.4.2 comment cela peut être réalisé. Si une valeur vi n'est pas présente dans l'ensemble ou la liste, l'opération le signale et est sans effet sur l'occurrence.

Exemples :

supprimer (*id03*.adresse.ville) donne à l'élément "ville" de loccurrence *id03* la valeur "ville: *?*".

supprimer (*id03*.année) donne à l'élément "année" de l'occurrence *id03*, la valeur "année:ϕ".

supprimer (*id03*.notes<1>) donne à l'élément "notes" de l'occurrence *id03*, la valeur "notes: <note:*?*, note: *11*, note:*07*>".

supprimer (*id03*.situation.marié_e) supprime le choix qui a été fait. Potentiellement "situation" a trois valeurs possibles dans l'occurrence, momentanément, l'élément "situation" reçoit la valeur ϕ.

Remarque :

Un trajet désigne une valeur d'élément et peut être utilisé comme <valeur> dans une opération **affecter** ou **supprimer**. Par exemple, on peut écrire :

affecter (*id03.notes*<2>, *id03*.notes<1>)

IV.4 Extension des FA_opérations : les FA_expressions

Pour augmenter la puissance des FA_opérations on introduit :
1. La notion de FA_variable et une opération pour parcourir un ensemble de valeurs.
2. La possibilité d'utiliser des fonctions particulières à la place de valeurs dans les FA_opérations. Les arguments de ces fonctions pouvant être des constantes ou des valeurs d'éléments (trajets).

IV.4.1 FA_variable

La notion de FA_variable est introduite pour permettre la conservation temporaire d'un FA qui peut être le résultat d'une requête ou bien une valeur d'élément (pour une occurrence). Une FA_variable est définie par :

FA (V, [UNIQUE] <expression>), V ∈ A

<expression> est (1) l'expression d'une requête ou (2) un trajet désignant la valeur d'un élément pour une occurrence particulière.

(1) <expression> := R (requête)

Nous avons vu que l'expression d'une requête produit un FA dont le schéma et la valeur sont connus.

Considérons la requête R dont le résultat est le FA Γ (S, Δ), S = A:*da*. L'opération **FA (B, R)** permet de donner un nom à la valeur Δ. La FA_variable B est un ensemble contenant une ou plusieurs occurrences construites selon le schéma S. On peut donc associer à chaque FA_variable un schéma qui est le schéma du FA résultat de la requête.

Exemple :

Pour le FA "étudiant" de la Figure 4.3, l'opération suivante :

FA (ETU1, élaguer(sélection(étudiant, >(année, 1958))
: adresse, situation, activités))

construit la FA_variable :

ETU1 = { étudiant: [nom: *balou*, année: *1960*,
notes:<note: *10*,note: *06*,note: *08*> }

ETU1 est défini sur le schéma
étudiant : [nom:D1, année:D2, notes:<(note:D2)$_{\text{nullallowed}}$>3]

Dans certains cas, on a besoin de s'assurer que la valeur de la FA_variable est un ensemble contenant une seule occurrence. Dans ce cas, on utilise l'option UNIQUE avant <expression>. La FA_variable est alors un ensemble contenant une valeur qui est la première occurrence du FA résultat de <expression>.

(2) <expression> := T (un trajet)

A tout trajet correspond à une valeur d'élément et un schéma d'élément. Ce schéma et la valeur désignée par le trajet forme un FA. A tout trajet T correspond donc un FA Γ. L'opération **FA** se définit alors de la même manière que lorsque <expression> est une requête.

Exemple :

Considérons l'occurrence *id03* du FA "étudiant". **FA** (NT, *id03*.activités) construit la FA_variable :

NT = { activités: {act: [noma: *photo*, nban: *3*]
act: [noma: *ski*, nban: *11*]} }

Le schéma de NT est activités: {act: [noma:D1, nban:D2]}

Pour cette forme d'expression de l'opération FA, il est inutile d'utiliser l'option UNIQUE puisqu'une valeur d'élément est toujours une sous-occurrence d'une occurrence de FA.

Manipulation de FA_variable

Une FA_variable est définie comme un FA. On peut donc lui appliquer toutes les opérations proposées pour les FA. Par exemple, on peut écrire :

FA (ETU2, élaguer(sélection(étudiant, < (année, 1958))
: adresse, situation, activités))

FA (ETU, union(ETU1, ETU2))

IV.4.2 L'opération : pour

L'opération **pour** permet de parcourir la valeur d'un FA ou la valeur d'une FA_variable. Comme nous l'avons vu, le schéma et la valeur d'un élément d'une occurrence de FA définissent un FA. On pourra donc utiliser la fonction pour sur des valeurs d'éléments et en particulier sur une valeur d'élément qui est un ensemble de valeurs (sous-occurrences).

L'opération **pour** a la syntaxe suivante :

pour (<nom>, <variable>)
<FA_expression> { ';' <FA_expression> }*
fin

<nom> := <nom_FA> | <nom_FA_variable> | <trajet>
<variable> := "?."<identificateur>.

Lors de l'exécution de l'opération **pour**, la variable prend comme valeur : l'identificateur interne d'une occurrence de FA. Cette variable pourra donc être utilisée à la place de <id_occ> dans un trajet.

Exemples :
```
pour (ETU1, ?e )
   supprimer (?e.notes)
fin

pour (étudiant, ?e)
   pour (?e.activités, ?a)
      affecter ( ?a.act.nban, 10)
   fin
fin
```

Ce dernier exemple montre l'utilisation de la fonction **pour**, pour parcourir un ensemble de sous-occurrences. En effet pour une occurrence ?e de "étudiant", ? e.activités est un trajet auquel correspond :

le schéma activités : { act: [noma:D1, nban:D2]}

et une valeur qui est un ensemble de la forme { activités : { I1, I2, ..., Ik} }, I1, I2, ..., Ik sont des sous-occurrences. Ces sous-occurrences définissent la valeur d'un FA de schéma act:[noma:D1, nban:D2] et ?a désigne une occurrence de cette valeur.

IV.4.3 Les fonctions

Les FA_opérations manipulent des données de FA. Un type du langage utilisé pour la mise en oeuvre des FA_opérations doit correspondre à un type de données de FA. On peut supposer que ce langage fournit un ensemble de fonctions de base pour la manipulation de ses types et qu'on dispose d'un mécanisme de définition de nouvelles fonctions (fonctions utilisateurs) pour la manipulation des types FA n'ayant pas de correspondant dans le langage.

Les fonctions les plus courantes sont :

- des **fonctions arithmétiques** sur les entiers, les réels : addition (+), soustraction (-), multiplication (*), division (/) ...

- des **fonctions sur le texte** : longueur d'un texte (lgtexte), concaténation de deux textes (concat), recherche d'une chaîne dans un texte ...

_ des **fonctions sur le temps** qui s'appliquent à des données de type temps simple, restreint, intervalle, période. Elles correspondent aux opérations et fonctions proposées dans [BUI 86, ADI 87]. Pour la manipulation de données temps d'un formulaire multimédia, nous avons besoin également d'une fonction interval_temps(t1,t2) qui construit une valeur de type temps-intervalle "t2-t1" à partir de deux valeurs de type temps t1 et t2 tel que t1<t2. Nous pensons également qu'il doit être possible d'appliquer certaines fonctions arithmétiques (min, max, moy) sur des listes de types temps simples ou restreints et temps durée.

- des **fonctions sur des listes** : somme, multiplication des valeurs d'une liste, maximum, minimum et moyenne des valeurs d'une liste, nombre d'éléments dans une liste , réorganisation d'une liste,...

- des **fonctions logiques** : et, ou, non, egal (=), inférieur (<), inclus (⊂),

appartient ($\in$), ... pour la comparaison de données de même type. Ces fonctions délivrent une valeur de type booléen. On dispose également de deux fonctions logiques particulières : VRAI et FAUX qui délivrent respectivement les valeurs boolénnes "vrai" et "faux".
Nous avons défini également la fonction **existe** qui vérifie l'existence d'une valeur pour un élément. Si la valeur de l'élément est ϕ, la fonction retourne la valeur booléenne "faux". Par contre si l'élément a une valeur v (pouvant être composé de valeurs ϕ) ou bien une valeur inconnue ou inapplicable, alors la fonction retourne "vrai". Cette fonction peut être appliquée à une FA_variable. Elle permet de vérifier que cette variable est un ensemble non vide.
Nous avons également la fonction **null** qui vérifie l'existence d'une valeur inconnue (?) pour un élément (inconnu).
La fonction **nonap** vérifie l'existence d'une valeur inapplicable (∂) pour un élément.

- la **fonction conditionnelle si** permet d'introduire une structure de contrôle "si <condition> alors <actions> sinon <actions>" sur les données d'un FA.
Sa syntaxe est : si (<condition>, { <actions-alors> } [, { <actions-sinon > }]).
La fonction **si** évalue la partie <condition> qui est une expression logique construite en utilisant les fonctions logiques. Si le résultat de cette évaluation est "vrai", la fonction éxécute la valeur de l'argument <actions-alors> sinon elle éxécute la valeur de l'argument <actions-sinon>. Les actions sont des FA_expressions séparés par un point-virgule ";".
On pourra trouver également la fonction VRAI dans la partie <condition>.

- des **fonctions de conversion** : réel - entier, argent - chaîne de caractères, ...

- des **fonctions sur les ensembles** : count, max, min, sum, vide qui délivre la valeur "vrai" si l'ensemble ne contient pas d'éléments.

- la fonction **changeacces** qui modifie le mode d'accès (lecture, écriture forte ou faible) d'un ou plusieurs éléments.

La plupart des fonctions ci_dessus s'appliquent à des valeurs d'éléments. Le résultat d'une fonction et une valeur. Elles peuvent être combinées avec les FA_opérations pour lesquelles la syntaxe autorise une valeur : **sélection, affecter, supprimer**. Dans le chapitre V, nous donnons des exemples d'utilisation de ces fonctions et nous détaillons les fonctions particulières que nous avons introduit.

Les fonctions utilisateurs

Une fonction utilisateur offre la possibilité de représenter un traitement particulier sur un FA. Par le biais de fonctions utilisateurs, on pourra intégrer à GIF, des outils particuliers développés par ailleurs sous forme de programmes. Ces programmes sont prédéfinis : traitement d'images, de texte, d'ensembles de formules, etc, ou ont été développés tout particulièrement pour un ou plusieurs FA.

La définition d'une fonction se fait en utilisant la commande suivante :

DEF_FONC <nom de fonction>
entrée <entrée>
sortie <sortie>
programme <nom de programme>

On donne à chaque fonction un nom unique, différent des noms de fonctions existantes pour GIF (FA_opérations, fonctions de base, ou fonctions utilisateur). Toute définition fonction fait référence à un programme dont le nom est donné dans la partie **programme**.

Dans la partie **entrée** de la définition d'une fonction, on décrit les occurrences de FA nécessaires à l'évaluation de la fonction. On définit <entrée> avec la syntaxe :

entrée := ent { ';' <entrée> }*

avec ent := nom_FA | **list** <min>, <max> **of** <nom_FA>

Une entrée "**FA1**; **FA2**" pour une fonction F indique que l'évaluation de la fonction nécessite une occurrence de FA1 et une occurrence de FA2.

La forme **list** <min>, <max> **of** <nom_FA> pour une entrée est utilisée lorsqu'on veut indiquer que plusieurs occurrences du même FA sont nécessaires à l'évaluation de la fonction. "**list** 2,4 **of** FA1" pour une fonction F indique que son évaluation nécessite la donnée d'au moins deux occurrences de FA1 et d'au plus quatre.

La partie **sortie** de la définition d'une fonction indique le FA qui pourra recevoir le résultat de son évaluation.

Exemple :

Supposons que l'on dispose du FA **carte : image;** et d'un programme : supercarte qui réalise la superposition d'au moins et d'au plus deux cartes. La commande :

DEF_FONC supercarte
entrée list(2, 2) of carte
sortie carte
programme superposition

définit la fonction supercarte qui pourra alors être utilisée dans les FA_expressions.

Une fonction pourra être définie même si les FA auxquels elle fait référence sont inconnus de GIF. La fonction est marquée comme non évaluable. Elle deviendra évaluable dès que les FA en question auront été définis. Dès que l'on modifie le schéma d'un FA, utilisé dans une fonction, la fonction est alors marquée comme non évaluable.

Il est clair que l'évaluation d'une fonction nécessite des transformations de données. GIF devra transformer toute occurrence en entrée sous une forme compréhensible par le programme à exécuter et reconstruire à partir du résultat de la fonction, une occurrence de FA.

IV.4.4 Exemples de FA_expressions

Avec les FA_opérations, les FA_variables, l'opération pour et l'ensemble des fonctions présentées ci-avant, on peut construire ce que l'on appelle des FA_expressions. Nous avons déjà donné des exemples de FA_expressions, en introduisant certaines opérations.

Exemples :

FA (MOYENNE, **renommer** ((**élaguer**(étudiant : adresse, situation, activités)), année → moy);

pour (MOYENNE, ?m)
  affecter (?m.moy, moylist(?m.notes));
fin

Ces expressions permettent à partir du FA "étudiant", de calculer la moyenne des notes de chaque étudiant. Le FA "MOYENNE" contient pour chaque étudiant : son nom, la moyenne de ses notes et les notes.

pour (étudiant, ?e)
    compress (?e.notes);
fin

Ces expressions permettent de réorganiser la liste de notes de chaque étudiant. Une rérganisation consiste à éliminer les trous de la liste. Par exemple pour une liste de valeurs L = "notes: <note: *10*, note:ϕ, note: *12*>", compress(L) conduit à la liste : "notes: <note: *10*, note: *12*>".

Remarque : Un FA peut avoir des éléments qui ont une valeur non-informative, inapplicable ou bien inconnue. Une FA_expression et une fonction doivent pouvoir s'exécuter en tenant compte de ces valeurs particulières.

IV.5 Les transactions GIF et leur validation

Nous nous intéressons ici à la notion de transaction associée à une opération spécifique : opération externe correspondant à une manipulation de formulaire par un utilisateur. Nous soulignons la nécessité d'avoir un mécanisme de transaction à ce niveau dans GIF et discutons de la sémantique que peuvent prendre les opérations **début_T**, **fin_T**, **annuler_T** utilisées pour décrire une transaction.

La mise à jour des formulaires donc des FA sous-jacents, nécessite un mécanisme de transactions. Considérons la tâche "remplir" un formulaire. Sur le FA sous-jacent, elle correspond à effectuer : (1) une création d'occurrence (créer), (2) des affectations de valeurs (affecter) à cette occurrence. La phase de valuation (remplissage) des champs est dirigée par la description du FA (structure + règles). On sait que certains champs (ne pouvant pas prendre une valeur non-informative ou bien une valeur inconnue) doivent obligatoirement être valués. Durant toute la phase d'affectation cette contrainte peut être violée. Elle ne doit pas être vérifiée en cours d'affectation, mais seulement à la fin. Il faut donc pouvoir considérer les actions d'affectation de valeurs aux champs comme une seule opération. A la fin de celle-ci, l'exécution de la règle de vérification d'existence de valeurs pour certains champs a un sens. On pourra ainsi rejeter ou accepter la création de l'occurrence.

La notion de transaction nous proposons est utilisée pour encapsuler des FA_opérations ou des FA_expressions. C'est donc une suite finie d'actions sur des occurrences de FA qui les font passer d'un état cohérent à un autre état cohérent [ADI 82] (cohérence faible ou forte).

Les opérations **début_T** et **fin_T** seront utilisées pour délimiter une transaction. Pour faire échouer volontairement une transaction, on propose l'opération couramment utilisée : **annuler_T**.

Par exemple lorsqu'un active l'opération "remplir", une transaction est créée (début_T). Si une opération "annuler_T" apparaît (suicide), alors la transaction échoue et toutes les actions faites sur le FA sont ignorées. Si aucune opération "annuler_T" n'apparaît (suicide) et qu'il se produit "fin_T" (l'usager indique explicitement qu'il a fini de remplir ou bien il active une autre opération spécifique) alors deux stratégies sont envisageables :

- considérer "fin_T" comme toute autre action sur un FA qui sera acceptée ou bien rejetée. Ce rejet correspond à "suicider" la transaction par l'opération "annuler_T".

- considérer "fin_T" comme un essai d'action (stratégie essai-erreur) : "je désire faire fin_T, que va-t-il se passer ?". Il faut déterminer l'état dans lequel risque de se trouver l'occurrence du FA. Deux cas de figure : - incohérent, - cohérent.

  Disposant de cette information, l'usager confirme ou ne confirme pas fin_T. Une confirmation de "fin_T" dans le cas cohérent ne pose pas de problèmes : les actions des règles sont confirmées et la transaction se termine. Par contre dans le cas incohérent, la transaction est "annulée". Quelque soit l'état, une non-confirmation de "fin_T" restaure le contexte de manipulation (courant) de l'occurrence dans l'état qui était le sien juste avant la demande de "fin_T". On revient au point de manipulation précédent "fin_T".

Le choix d'une stratégie n'est pas évident. Il existe peu de littérature sur le sujet. Dans la première approche, on adopte la solution du "tout ou rien" qui est mal adaptée avec le concept de FA multimédia. La valuation d'un champ multimédia peut nécessiter un temps de manipulation important et le fait d'ignorer cette action simplement parce qu'un autre élément du formulaire n'est pas valué est peut être un peu "dur" pour l'usager.

Dans la seconde approche, on offre la possibilité de ne pas perdre les manipulations effectuées jusqu'au point d'essai "fin_T". Cette stratégie est mieux adaptée à l'usager mais elle coûte chère. Chaque fois que "fin_T" est demandée, il faut exécuter des règles puis confirmer ou ne pas confirmer les actions qu'elles ont réalisées. De plus, on risque de voir se dérouler des transactions de longue durée. Une solution pour minimiser ce risque est de donner à chaque opération spécifique définie sur un formulaire une durée de vie (d'exécution) maximale. Ce qui nécessite le contrôle du temps d'exécution d'une transaction.

Nous avons choisi dans un premier temps d'adopter la première stratégie. Donc toute action "fin_T" qui est confirmée implique une confimation de la transaction dans GIF. Dans le cas de notre exemple "remplir", toutes les opérations réalisées sur le FA sous-jacent (créer, affecter, supprimer ...) sont validées. Le contexte de manipulation de l'occurrence est mis à jour.

Bien que confirmée, on ne voit pas les effets d'une transaction GIF (opération spécifique) sur la base. Il faut que l'utilisateur **valide** son opération. Pour GIF cela correspond à translater un ensemble d'opérations de mises à jour de FA en opérations de mise à jour reconnues par le SGBD. Pour réaliser une validation, GIF utilise le contexte d'exécution de l'opération : le journal des FA_opérations réalisées durant l'opération spécifique. Pour l'instant nous n'acceptons pas de validation globale d'opérations. Chaque fois qu'une opération spécifique se termine correctement (fin_T est confimée), GIF valide cette opération.

Avec l'exemple de l'opération "remplir", nous avons considéré les transactions encapsulant des ordres de mise à jour de FA. Néanmoins avant toutes mises à jour, il faut disposer dans GIF des données à manipuler. L'utilisateur précise quelles occurrences de FA il veut manipuler en activant une opération spécifique du type "recherche". Il a alors la possibilité de spécifier sa requête au travers du formulaire. La confirmation de la transaction associée à la recherche a pour effet de construire la FA_expression correspondante. La validation de l'opération a pour effet de réaliser la recherche proprement dite dans la base et de ramener dans GIF, les occurrences retrouvées.

Remarque :

1. Dans une transaction, on ne peut pas trouver les opérations **début_T** et **fin_T**. On n'accepte pas des transactions imbriquées

2. Nous n'avons pas discuté du cas d'annulation d'une transaction par panne. Dans la plupart des SGBD, le mécanisme de reprise efface de la base toute trace de l'exécution des transactions en cours au moment de la panne. Pour des cas de panne dans GIF, ce type de mécanisme est mal adapté : on risque de perdre l'exécution de nombreuses actions longues et coûteuses (de par les contrôles qui sont réalisés pour chaque action ou groupe d'actions). Après une panne, il faudrait pouvoir réactiver l'opération que l'on était en train de réaliser au moment de la panne et se retrouver au point de manipulation précédent la panne. Cela suppose que l'on soit capable de maintenir, dans le "journal" d'une opération spécifique s'exécutant, suffisament d'informations pour restaurer le contexte d'exécution de l'opération et non pas défaire ce qui a été fait.

CHAPITRE V

LES REGLES

La méthode ne peut plus se séparer de son objet

W. Heisenberg

CHAPITRE V

LES REGLES

Définir un FA c'est décrire :

- un schéma représentant les propriétés structurelles du formulaire
- de la connaissance permettant à GIF de s'assurer que les données du FA (occurrences) sont correctes du point de vue de leur sémantique.

L'expression de la connaissance attachée à un schéma de FA se fait en utilisant le concept de **règle**. Chaque règle définie est propre à un schéma et appartient à la partie <règles> de la commande DEF_FA (chapitre III).

Une règle est utilisée pour décrire :

- les propriétés que doivent vérifier les occurrences de formulaire abstrait pour exister (contraintes). Certaines de ces propriétés sont inhérentes au schéma et constituent la sémantique relative au concept de FA (conformité d'une occurrence au schéma, existence de valeur pour les éléments clés, etc).
- les politiques d'opérations pour chaque donnée modifiable (élément ou occurrence de FA).
- les liens entre les éléments du schéma et des éléments avec d'autres objets base de données au sens large.

Nous avertissons le lecteur que nous n'avons pas cherché à résoudre ici tous les problèmes liés à la gestion de la dynamique des données. Le concept de règle est défini dans un cadre précis qui est celui de l'interaction du formulaire avec l'utilisateur. Il est donc introduit pour modéliser et gérer le comportement d'un formulaire en fonction des actions de l'utilisateur, qui correspondent à des opérations de mise à jour de FA. On se place donc dans la situation où l'utilisateur active une opération spécifique pour la création ou la modification d'un formulaire : une transaction débute, une opération **créer** ou **mofifier** est demandée puis des opérations **affecter** et **supprimer** se produisent.

Dans la suite de ce chapitre, nous donnons (section V.1) la commande proposée pour la définition d'une règle en détaillant ses différents composants et en montrant comment sera traitée une règle. Dans la section V.2, nous présentons des exemples de règles de manière à introduire une classification en : règles de contrainte, règles de statut, règles de valeur, règles de report, règles d'exception. Finalement en section V.3, nous discutons du problème de la cohérence des règles en montrant plus particulièrement comment on peut éviter les cycles dans les définitions.

V.1 Définition d'une règle

Elle se fait avec la commande suivante :

DEF_R <nom_règle> :

WHEN <liste_événements>
BEFORE <liste_actions_avant>
AFTER <liste_actions_après>

Plusieurs commande DEF_R peuvent être émises pour le même FA. Chaque règle définie pour un FA a un nom unique. Une définition de règle doit contenir obligatoirement une clause WHEN et au moins une des clauses BEFORE et AFTER.

V.1.1 La clause WHEN

Dans la section III.2.3, nous avons discuté de l'état d'une occurrence de FA. Nous avons montré que cet état est déterminé en fonction du contenu de l'occurrence et des propriétés sémantiques qu'elle vérifie. Dans un environnement interactif, la vérification d'une propriété sémantique se fait au plus tôt, c'est à dire dès que les actions de mise à jour sur les données intervenant dans la propriété se sont produits.

Si l'on dispose dans la définition d'une propriété, d'une partie décrivant quand (WHEN) celle-ci doit être vérifiée, on privilégie l'aspect "contrôle immédiat des actions utilisateurs". Le formulaire est alors un objet intelligent qui détecte et rejete au plus tôt les actions le faisant évoluer dans un état non conforme à sa raison d'être (support structuré d'informations non ambiguës).

Dans la clause WHEN on trouve une liste d'**événements** conditionnant le déclenchement de la règle pour son traitement.

1- Qu'est-ce qu'un événement ?

Dans la littérature sont proposées différentes définitions de la notion d'événement. Ce peut être une décision, un changement du monde réel à reflèter dans la base, une observation ou décision liée au temps, une action ou activité dans laquelle participe les objets de la base, le début ou la fin d'un changement d'état de la base, une condition pour l'activation d'une transaction, une création ou suppression d'entités dans la base [ANT 81].

Dans le cadre que nous nous sommes fixés, un événement est une notion duale de celle d'une FA_opération de mise à jour. L'activation d'une règle est donc conditionnée par les opérations : **créer, modifier, détruire, affecter, supprimer, fin_T, annuler_T**. Les opérations pour l'expression de requêtes : *sélection, union, intersection, différence, renommer, produit, élaguer, grouper et éclater* et les opérations : *FA, début_t, si, etc...* ne peuvent pas être utilisées pour activer une règle.

On dit qu'un événement Evt se **produit** lorsque l'opération qui lui correspond est demandée. Si le traitement de cette opération se passe bien, alors on dit que l'événement Evt est **confirmé**.

Dans une opération représentant un événement, on peut utiliser une variable (chaîne de caractères précédée d'un point d'interrogation). Lorsque l'opération se produit, la variable est valuée. La variable pourra être référencée dans les FA_expressions décrivant les clauses BEFORE et AFTER de la règle. Par exemple, pour le FA "demande_mission", on peut décrire les événements :

créer (demande_mission, ?m);

Lorsqu'une opération de création d'occurrence pour le FA "demande_mission" se produit, la variable "?m" prend comme valeur un identificateur d'occurrence.

affecter (?d.nom_prénom, ?n);

Lorsqu'une affectation de valeur *V* pour l'élément "nom_prénom" d'une occurrence *idocc* du FA "demande_mission" se produit, la variable "?d" prend pour la valeur *idocc* et la variable "?n" prend la valeur *V*.

Dans nos exemples de règles, on pourra constater que l'on introduit très souvent dans la clause WHEN un événement **créer**. Cet événement est présent uniquement pour rappeler à quel FA se rattache la règle.

2- Plusieurs événements dans la clause WHEN : Pourquoi ?

Prenons l'exemple de la propriété **P** ≡ "début_mission" **precede** "retour_mission" devant être vérifiée par toute occurrence du FA "demande_mission".

Supposons que dans la clause **WHEN**, on ne puisse spécifier qu'un seul événement. Pour décrire la propriété **P**, il faut alors écrire deux règles :

Règle1 : WHEN affecter (?m.date_mision.début_mission, ?d)
s'il existe une valeur r pour "retour_mission"
alors vérifier la condition **C** ≡ ?d ≤ r

Règle2 : WHEN affecter (?m.date_mission.retour_mission, ?r)
s'il existe une valeur d pour "début_mission"
alors vérifier la condition **C** ≡ **d** ≤ ?r

Dans ces deux règles, on retrouve finalement la même condition (**C**) à vérifier. On remarque qu'avant de *vérifier*, on s'assure de disposer des deux valeurs à comparer. Ce qui correspond :

- pour Règle1 à s'assurer qu'un événement "affecter (?m.date_mission.retour_mission, ?r)" s'est produit.
- pour Règle2 à s'assurer qu'un événement "affecter (?m.date_mission.début_mission, ?d)" s'est produit.

On peut donc dire que la condition C doit être vérifiée lorsque :

- l'événement "affecter (?m.date_mision.début_mission, ?d)" se produit **ET** un événement "affecter (?m.date_mission.retour_mission, ?r)" s'est produit.
- l'événement "affecter (?m.date_mission.retour_mission, ?r)" se produit **ET** un événement "affecter (?m.date_mision.début_mission, ?d)" s'est produit.

Ceci montre que l'ordre dans lequel a lieu les événements n'est pas important mais qu'il faut obligatoirement que les deux opérations qui leur correspondent aient été demandées (l'une des opérations n'a toutefois pas encore été traitée).

Bon nombre de propriétés sémantiques et de contraintes, portent sur plusieurs valeurs. En autorisant l'écriture de plusieurs événements dans la clause WHEN, il suffit de définir une seule règle pour contrôler ce type de propriétés.

La propriété P sera donc décrite par une règle dont la clause WHEN a la forme suivante :

WHEN affecter(?m.date_mission.début_mission, ?d);
affecter(?m.date-mission.retour_mission, ?r);

vérifier la condition : ?d $\leq$?r

3- Définition

La clause WHEN d'une règle a la forme générale :

$< Evt_1; Evt_2; ....; Evt_{n-1}; Evt_n;>$ où Evt_i est un événement

L'apparition d'un événement (opération) intervient toujours durant une transaction GIF (≡ à une opération spécifique pour l'utilisateur). Au cours d'une transaction, on peut se trouver dans le cas de figure suivant :

(1) L'événement $Evt_i \in \{ Evt_1, Evt_2, ..., Evt_{n-1}, Evt_n\}$ se produit et n-1 événements Evt_j, $j \in \{1,...,n\}$ et $j \neq i$ se sont produits (dans n'importe quel ordre au cours de la transaction ou bien dans une autre transaction) alors la règle s'exécute. On dit qu'elle devient **active**. On exécute alors les actions de la clause BEFORE. Si une opération d'annulation d'événement (voir section V.1.2) est réalisée dans la clause BEFORE, l'événement Evt_i est rejeté. Le résultat de l'exécution de la règle est un **échec**. Si aucune annulation d'événement n'est réalisée alors on éxécute les actions de la clause AFTER. Si une de ces actions ne peut pas être exécutée alors on rejète Evt_i. Si par contre toutes les actions sont réalisées, alors Evt_i est **confirmé** et l'exécution de la règle est un **succès**. (voir section V.1.4).

(2) Il existe des événements de la clause WHEN qui ne sont pas produits. La règle est dite en **en attente** (d'activation). Elle deviendra active au cours de la transaction, ou bien dans une autre transaction.

Cas particulier de l'événement "fin_T"

L'événement "fin_T" ne peut pas se trouver en début ou au milieu de la liste des événements de la clause WHEN. Il est soit le seul événement de la liste ou bien il se trouve en fin de liste. Une clause WHEN avec l'événement fin_T a donc la forme générale :

< Evt_1; Evt_2; ....; Evt_{n-1}; Evt_n; **fin_T**;>

L' exécution de la règle se fait lorsque **fin_T** se produit et que tous les autres événements se sont produits. Dans la section IV.5 (chapitre IV), nous avons discuté des sémantiques possibles pour l'opération **fin_T**. Nous avons choisi de donner à cette opération une sémantique traditionnelle qui consiste à annuler la transaction si **fin_T** n'est pas confirmée. Si l'exécution de la règle est un échec cela correspond à ne pas confirmer **fin_T** donc à réaliser **annuler_T**. Par contre si **fin_T** est confirmée alors la transaction GIF est **validée** pour la base de données.

Exemples de clauses WHEN

Exemple 5.1 :

WHEN créer(remboursement_frais, ?r)

La règle introduite par cette clause devient active losrqu'on crée une occurrence du FA "remboursement_frais".

Exemple 5.2 :

WHEN affecter(?r.entête, ?t); affecter(?r.envers, ?v);
affecter(?r,voyage.aller_retour); fin_T;

Cette règle devient active lorsque **fin_T** se produit ET que les événements "affecter(?r,voyage.aller_retour)", "affecter(?r.entête, ?t)", "affecter(?r.envers, ?v)" se sont produits.

V.1.2 La clause BEFORE

Cette clause contient un ensemble d'actions devant être réalisées avant la confirmation de l'événement ayant rendu la règle active. Supposons une règle possédant une clause BEFORE. Si cette règle devient active, les actions de la clause BEFORE sont alors exécutées.

Quelles sont les actions rencontrées dans une clause BEFORE?

L'exécution des actions d'une clause BEFORE conditionnent la confirmation de l'opération liée à l'événement ayant rendu la règle active; Il paraît clair que ces actions modélisent des **contraintes** d'intégrité sémantique liées à l'événement. Comment exprimer ces contraintes ?

Dans le chapitre II (section II.1.1), nous avons présenté la notion de *trigger* proposé dans [GIB 84]. Dans cette approche :

- la clause *condition* spécifie la contrainte à vérifier (condition portant sur les variables introduites dans la clause *pattern* ou sur des variables globales)
- la clause *erreur* donne l'action à exécuter lorsque la condition n'est pas vérifiée (en général un message d'erreur). L'exécution de cette clause provoque l'arrêt de l'exécution du *trigger* et la non validation de l'opération du *pattern*.

Mais nous avons pu constater qu'avec cette forme de *trigger*, il n'est pas possible de :

- spécifier plusieurs conditions : par exemple, pouvoir décrire que lorsqu'on affecte une valeur à un champ clé il faut (1) vérifier que ce champ n'a pas déjà une valeur (champ non modifiable). Si tel est le cas, il faut (2) vérifier que la valeur donnée identifie de manière unique l'occurrence crée (pas d'occurrence existante avec cette valeur).
- forcer un utilisateur à réaliser un événement : "avant de pouvoir confirmer votre opération, on veut la réalisation de telle opération". L'instruction demandant la réalisation de cette opération doit pouvoir être interprétée par l'interface Utilisateur-GIF comme "en attente de tel événement".

La notion de règle est à rapprocher de celle de *trigger* dans [GIB 84] mais la clause BEFORE ne peut pas être définie comme une simple clause "condition - erreur" puisqu'on veut spécifier différentes contraintes à vérifier et demander la réalisation d'actions. Pour ne pas "alourdir" la syntaxe d'une règle par de nouvelles sous-clauses (condition, action) et par souci d'homogénéité des concepts, nous utiliserons certaines fonctions pour décrire les actions de la clause BEFORE.

1- La fonction si

La fonction si sera utilisée pour exprimer bon nombre de contraintes. Avec cette approche, il faut alors décrire explicitement quand l'événement doit être annulé. Nous avons donc besoin de la fonction : **annuler_evt** dont l'effet est de rejeter l'événement ayant rendu la règle active et de stopper l'exécution de la règle.

annuler_evt correspond à une action corrective vis à vis d'une violation de contraintes. Avant d'annuler, on peut donner un message explicitant pourquoi l'événement est rejeté (fonction **message**).

2- La fonction **wait_evt**

Elle est utilisée pour forcer un utilisateur à réaliser un <événement>. Sa syntaxe est : **wait_evt** (<événement> [,<temps>]). L'argument <temps> est une durée (par exemple : *2h30:00*) qui permet de limiter la durée d'attente d'un événement. La fonction retourne un code CWE qui indique à la règle - si l'événement demandé s'est produit et a été confirmé (CWE= 1) ou - si l'événement ne s'est pas produit dans le temps prévu (CWE=0). Lorsque l'interface n'a pas connaissance de l'élément concerné par l'événement, la fonction n'est pas exécutable et CWE = 0. Cette fonction fait un appel bloquant à l'interface "Utilisateur-GIF" : tant que l'interface n'a pas retourné une réponse (code CWE) à GIF, l'exécution de la règle reste bloquée.

3- La fonction **FA** qui permet de construire une FA_variable.

IMPORTANT : Dans la clause BEFORE, on ne peut pas trouver d'actions décrites par une opération correspondant à un événement.

Exemples de clauses BEFORE

Exemple 5.3 :

```
WHEN affecter(?d.date-mission.début-mission, ?m);
        affecter(?d.date-mission.retour-mission, ?r);
BEFORE
si( precede(?r, ?m),
   { message('date retour-mission doit être supérieure à date début-mission');
     annuler_evt;
   } );
```

Dans la clause BEFORE de cette règle on exprime une contrainte d'intégrité sur les éléments "début-mission" et "retour-mission" du FA "demande_mission". La valeur donnée au premier doit "précéder" dans le TEMPS celle donnée au second.

Exemple 5.4 :

```
WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s);
BEFORE
si( et ( existe(?d.imputation.type, ?t), existe(?d.imputation.ident_type, ?i),
      existe(?d.imputation.code_type, ?c) ),
   { FA( RES, sélection(listeresponsable, et ( =(type, ?t), =(ident_type, ?i),
                  =(code_type, ?c),
                  sélection( responsables, ∃ (responsable, =(nom, ?USER)) ));
     si( non (existe (RES)),
        {  message('Vous ne disposez pas des droits pour signer une
                     demande de mission ');
           annuler_evt;
        } );
   },
   { message('Avant de signer, veuillez préciser sur quel contrat ou laboratoire
              les frais occasionnés par la mission doivent être imputés ');
     annuler_evt;
   } );                                        /* fin */
```

Lorsqu'un utilisateur signe une demande de mission, on doit vérifier qu'il est le responsable du contrat ou du laboratoire sur lequel a été imputée la mission. Pour cela on consulte l'objet "listeresponsable" (FA) de schéma :

```
listeresponsable: [type:D, ident_type:D, code_type:D,
                   responsables: {responsable:D}]
```

Ce FA donne pour chaque contrat ou laboratoire ("type"), son nom ("ident_type"), son code ("code_type") et l'ensemble ("responsables") des noms d'utilisateurs responsables. On recherche donc dans cet objet s'il existe une occurrence pour laquelle : l'élément "type" a une valeur égale à celle donnée pour ? d.imputation.type ET l'élément "ident_type" = ?d.imputation.ident_type ET l'élément "code_type" = ?d.imputation.code_type ET il existe un élément (responsable) de l'ensemble "responsables" ayant pour valeur ?USER.

Cette règle montre l'utilisation de la fonction :

- **existe**(<trajet>, [V|) où <trajet> désigne un élément de FA, et V une variable optionnelle. Cette variable contient la valeur de l'élément. La fonction délivre la valeur "vrai" s'il existe une valeur pour l'élément : sa valeur est donc $\neq \phi$. Dans le cas où l'élément (ou un des composants) a pour valeur ϕ, la fonction retourne la valeur "faux". On remarque que la fonction peut être appliquée à une FA_variable qui désigne la racine (élément) de son schéma : existe(RES) délivre la valeur "vrai" si RES $\neq \{\phi\}$ (si RES n'est pas un ensemble vide).

On utilise également une variable réservée de GIF : **?USER**. Elle a pour valeur l'identifiant de l'utilisateur courant du formulaire (celui qui a activé le formulaire). Nous disposons également de la variable ?TIME qui délivre le temps du système.

La règle de l'exemple 5.4 aurait pu s'écrire :

Exemple 5.5 :

```
WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s);
BEFORE
si( et ( existe(?d.imputation.type, ?t), existe(?d.imputation.ident_type, ?i),
      existe(?d.imputation.code_type, ?c) ),
   {  message('Avant de signer, veuillez préciser sur quel contrat ou laboratoire
              les frais occasionnés par la mission doivent être imputés ');
      wait_evt( affecter(?d.imputation.type, ?t), 30:00);
      wait_evt( affecter(?d.imputation.ident_type, ?i), 30:00);
      wait_evt( affecter(?d.imputation.code_type, ?c), 30:00);
      si( ≠(CWE, 1), annuler_evt);
   },
   { FA( RES, sélection( listeresponsable, et( =(type, ?t), =(ident_type, ?i),
            =(code_type, ?c),
            sélection( responsables, ∃ (responsable, =(nom, ?USER))) ) ) );
     si( non(existe(RES)),
        { message('Vous ne disposez pas des droits pour signer une
                   demande de mission ');
          annuler_evt;
        } );
   } );
```

On "force" l'utilisateur à donner une valeur aux éléments "type", "ident_type" et "code_type" en utilisant la fonction **wait_evt**.

L'action **wait_evt (affecter(?d.imputation.type, ?t), 30:00)** a une signification particulière pour l'interface Utilisateur-GIF. Elle signifie :

** conserver le contrôle du curseur sur l'élément "type" (pendant une durée maximale de 30:00), tant que l'événement "affecter (?d.imputation.type, ?t)" ne s'est pas produit.**

Si l'événement s'est produit et a été confirmé, l'interface retrourne à GIF un code et le traitement de la règle se poursuit : on demande d'affecter l'élément "ident_type".

Une telle action dans la clause BEFORE ne modélise pas une contrainte sémantique au sens courant (sur des valeurs d'éléments) mais une contrainte sur un événement. La règle décrit un enchaînement d'actions **arrière**.

V.1.3 La clause AFTER

La confirmation d'un événement peut entraîner l'initialisation d'une ou plusieurs actions à exécuter sur le formulaire ou sur d'autres objets. Ces actions sont décrites dans la clause AFTER. Chaque action est une FA_expression. On trouvera également des actions pour forcer la réalisation de certaines opérations par l'utilisateur (fonction **wait_evt**). Dans ce cas la règle décrit un enchaînement d'actions **avant** . Dans une clause AFTER, on interdit l'utilisation des opérations **début_T** et **fin_T** (pas de transactions imbriquées) et de l'opération **annuler_evt**.

Exemple 5.6

```
WHEN créer(remboursement_frais, ?r);
        affecter(?.r.envers.groupe.type-groupe, ?t);
        affecter(?r.envers.indice, ?i); affecter(?r.envers.fonction, ?f);
BEFORE
FA( GR, UNIQUE élaguer( sélection(tablegroupe, =(indice, ?i)) : indice ) ;
si( non(existe(GR)), annuler_evt);
AFTER
si( =(?t, france),
   { si( =(?f, "enseignant"), affecter(?r.envers.groupe.valeur_groupe, 1),
         /* sinon */
         pour (GR, ?g)
           affecter(?r,envers.groupe.valeur_groupe, ?g.grfrance);
         fin;
      );
   },
   /* t = étranger */
   pour (GR, ?g)
     affecter( ?r.envers.groupe.valeur_groupe, ?g.grétranger)
   fin;
);
```

Cette règle concerne le FA "remboursement_frais". Lorsqu'on crée une occurrence de ce FA, l'exécution de la règle value son élément "valeur-groupe" (si les éléments "indice", "fonction" et "type_groupe" ont une valeur). La valeur donnée est fonction de l'"indice" et de la "fonction" de la personne ayant effectuée la mission et du type de la mission ("type-groupe" = *france* ou *étranger*). Si la personne est un enseignant et que la mision a lieu en *france* alors la valeur du groupe est 1. Dans les autres cas de figure, la valeur à donner à "valeur_groupe" est extraite (clause BEFORE) de l'objet **tablegroupe** (indice, grfrance, grétranger) ≡

tablegroupe: [indice:D, grfrance:D, grétranger:D] qui donne pour chaque indice, la valeur des groupes "grfrance" et grétanger".

Les valeurs des éléments "prix_découcher", "prix_repas", "prix_jour" seront calculées par rapport à la valeur de "valeur_groupe".

Exemple 5.7 :

WHEN créer(demande_mission, ?m);
AFTER
{ wait_evt(affecter (?m.id_mission, ?i));
wait_evt(affecter (?m.no_secu, ?n));
});

Cette règle pour le FA "demande_mission" permet de s'assurer que dès la création d'une nouvelle occurrence, l'utilisateur donne une valeur à l'élément clé "id_mision" et à l'élément "no_secu". Cette règle décrit un enchaînement d'actions **avant**.

V.1.4 Traitement d'une règle

Après avoir montré en quoi consiste la définition d'une règle, nous discutons dans cette partie du traitement d'une règle.

Supposons l'existence d'un interpréteur pour le traitement des FA_expressions. Cette interpréteur traite les opérations dans l'ordre dans lequel il les reçoit. Supposons qu'il doive traiter une opération **OP** sur un FA. Une étape particulière du traitement à réaliser consiste à déterminer les règles dont la clause WHEN contient un événement qui correspond à **OP**. S'il existe des règles, il faut alors déterminer celles qui deviennent actives par OP et les exécuter.

1) Comment déterminer les règles à exécuter ?

Une règle devient active si tous les événements de sa clause WHEN (autre que OP) se sont produits. Pour déterminer les règles à exécuter pour **OP**, il faut disposer d'informations indiquant les divers événements qui se sont produits sur l'occurrence du FA concernée par **OP**. Pour chaque occurrence de FA, il faut maintenir un contexte d'activation (des règles) qui est la liste des **derniers** événements qui se sont produits sur l'occurrence. Dans le contexte de manipulation d'une occurrence (chapitre III), on conserve pour chacun de ses éléments l'historique des opérations qu'elle a subi. Chaque fois qu'une

occurrence est manipulée, il est donc possible de construire dynamiquement le **contexte d'activation** pour les règles du FA. Ce contexte évolue chaque fois qu'un événement se produit et est confirmé.

2) Dans quel ordre exécuter ces règles ?

Soit *Règle1* avec la clause **WHEN** E_1; EVT; E_3; et *Règle2* avec la clause : **WHEN** E_4; E_5; EVT;.

On suppose que E_1, E_3, E_4, E_5, se sont produits et que EVT se produit. *Règle1* et *Règle2* sont à exécuter. En fait c'est comme si on avait à exécuter une seule règle avec la clause **WHEN** E_1; EVT; E_3; E_4; E_5;. Sa clause BEFORE (AFTER) est la conjonction des clauses BEFORE (AFTER) de *Règle1* et *Règle2*. Cette approche est généralisable à **n** règles. En section V.3.2, on énonce la propriété que doivent vérifiées les règles pouvant devenir actives en même temps.

On constate donc que pour une opération (OP) rendant n règles actives, tout se passe comme si on traitait une seule règle R. Si R possède une clause **BEFORE** alors l'interpréteur traite les opérations de cette clause. Si par contre R ne possède pas de clause **BEFORE** alors l'interpréteur termine le traitement de OP : réalise OP et traite les actions de la clause **AFTER**.

Traitement de la clause BEFORE

Si une opération d'annulation se produit alors l'interpréteur stoppe le traitement de **OP**. L'opération est refusée. Si aucune opération annuler_evt ne se produit alors l'interpréteur poursuit le traitement de **OP** : réalise OP et traite ensuite les opérations de la clause **AFTER** (si celle-ci existe pour la règle).

Traitement de la clause AFTER : enchaînement des règles

OP a été réalisée. Supposons que la clause AFTER de la règle (Règle1) à exécuter pour **OP** contiennent les opérations OP1, OP2, ..., OPn. L'interpréteur doit traiter : OP1 puis OP2, ..., puis OPn (voir Figure 5.1). Supposons que l'interpréteur traite OPi pour laquelle une règle (Règle2) doit être exécutée :

1. Si l'opération OPi est refusée, l'interpréteur passe alors au traitement de l'opération OPi + 1. On peut donc se trouver avec des actions de la clause **AFTER** qui ne sont pas réalisées. Ceci peut conduire à des situations non

prévues pour l'occurrence traitée. Dans le cas de règles spécifiant des reports de mises à jour d'éléments virtuels sur les objets sources, cela peut être avoir pour conséquence la non réalisation de ces reports.

2. L'opération OPi est réalisée et l'interpréteur doit traiter les opérations OP1', OP2', ...,OPn'. Ces opérations peuvent conduirent à la réalisation d'autres règles et à la réalisation (partielle ou totale) d'autres actions.

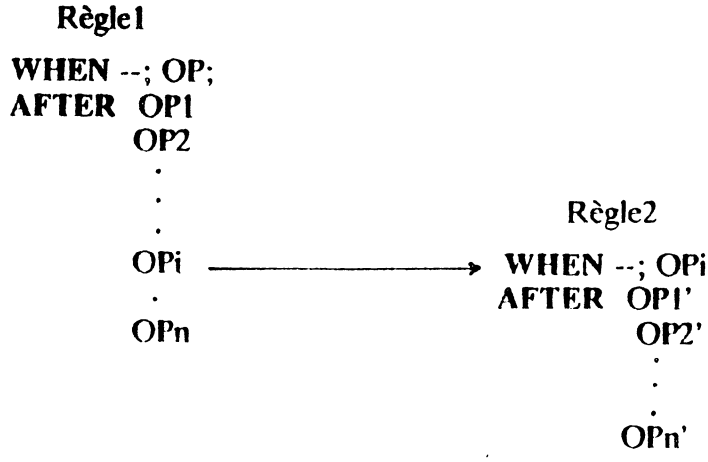


Figure 5.1 : Cascades de règles

En conclusion, on peut avoir une réalisation partielle des actions prévues dans la clause AFTER et une cascade d'opérations à traiter par enchaînement de règles. L'enchaînement des règles ne doit pas aboutir à un cycle. Dans la section suivante, nous discutons de ce problème et montrons qu'il est utile de proposer une méthodologie pour la définition des règles interdisant les cycles dans leurs définitions.

Pour limiter les effets de bord de la non réalisation d'une action (de la clause AFTER), nous adoptons la stratégie suivante :

> *Lorsqu'une action de la clause AFTER n'est pas réalisée, l'interpréteur stoppe l'exécution de la règle et annule les opérations qu'il a réalisé depuis le début de l'exécution de la règle. OP est donc refusée.*

Une règle se comporte donc comme **une mini-transaction**. On parle alors :

- de l'échec d'une règle si une opération annuler_evt se produit ou si une des actions de la clause AFTER n'est pas réalisée.

- du **succès** d'une règle si toutes les opérations de la clause AFTER ont été réalisées. Le succès d'une règle **confirme** l'opération l'ayant déclenchée.

V.2 Exemples de règles : Classification

Par les exemples présentés dans cette section, nous montrons que la notion de règle est adaptée à la représentation des diverses propriétés sémantiques des formulaires complexes.

V.2.1 Règles de contrainte

En utilisant le concept de règle, il est possible de décrire bon nombre des contraintes couramment admises sur les objets des bases de données. Ces règles sont qualifiées de **règles de contrainte**.

En faisant un parallèle entre le concept de relation et celui de FA, nous montrons dans les exemples suivants que les règles WHEN-BEFORE correspondent à des contraintes d'intégrité (CI) appartenant aux classes admises pour les SGBD relationnels [DEL 83, OLI 86] :

- CI monorelationnelles ou multirelationnelles
- CI individuelles ou d'ensemble
- CI statique ou dynamique
- CI immédiate ou différée

La plupart des règles que nous avons présentées dans les sections précédentes sont des règles de contrainte. La règle de l'exemple 5.4 décrit une contrainte du type monorelationnelle, individuelle, statique et immédiate. La règle de l'exemple 5.5 décrit une contrainte du type multirelationnelle, individuelle, statique et immédiate.

Exemple 5.8 :

Dans cette règle, on exprime que tout code ("code_type") surlequel sera imputé les frais occasionnés par une mission (?m) doit appartenir à la liste des codes, de contrats ou de laboratoires, repertoriés dans l'objet BD "tablecode (code, type, ident)" ≡ tablecode: [code:D, type:D, ident:D]

```
WHEN créer(demande_mission, ?m);
        affecter(?m.imputation.type, ?t);
        affecter(?m.imputation.ident_type, ?i);
        affecter(?m.imputation.code_type, ?c); fin_T;
BEFORE
FA( COD, élaguer( sélection(tablecode, et ( =(type, ?i), =(ident, ?i)) : type,
ident);
si( ∈(?c, COD),
    { message('code de contrat ou de laboratoire inconnu');
      annuler_evt;
    } );
```

La contrainte exprimée ici est du type multirelationnelle, individuelle, statique, et différée (en fin de transaction). Cette contrainte appartient à la classe particulière des contraintes d'intégrité **référentielle** : le "code_type" référence une occurrence de l'objet "tablecode".

Exemple 5.9 :

```
WHEN créer(professeur, ?p); affecter(?p.car_person.salaire, ?s)
BEFORE
si( <=( ?s, ?p.car_person.salaire) ,
   { message('le salaire ne peut pas diminuer');
     annuler_evt } ) ;
```

Dans la clause BEFORE, on décrit le fait que le salaire d'un professeur ne peut pas diminuer. Puisque l'action "affecter(?e.car-personne.salaire, ?s)" n'a pas été confirmée, la donnée "?e.car-personne.salaire" est la valeur de l'ancien salaire. On peut donc comparer l'ancienne et la nouvelle valeur du salaire sans avoir à écrire explicitement leur état (OLD et NEW).

La contrainte spécifiée est de type monorelationnelle, individuelle, dynamique et immédiate.

Exemple 5.10 :

Lorsqu'on donne une valeur à l'élément "car_person" d'une occurrence de FA "professeur", on veut s'assurer que la propriété : *la moyenne des salaires des professeurs est > 12000 F* est toujours vérifiée.

```
WHEN créer(professeur, ?p); affecter(?p.car_person.salaire, ?s);
BEFORE
FA( SAL, sup×( élaguer( sélection(personne, et ( =(fonction, professeur),
                                    ≠(no_ss, ?p.car_person.no_ss)) )
                  : nom, prénom, adresse, indice, fonction, mode_paiement)
               )
   );
si ( ≤(  /( +( moy (SAL), ?s), 2) , 12000),
   { message('Avec cette valeur pour le salaire, la moyenne des salaires des
              professeurs devient supérieure à 12000 F');
     annuler_evt; } );
```

Cette règle décrit une contrainte multirelationnelle, d'ensemble, statique et immédiate.

V.2.2 Règles de statut

Les règles qualifiées de **statut** décrivent ce qui est particulier à un élément de FA. Elle donne à l'élément un statut : obligatoire, modifiable, non modifiable, personnalisé, verrouillé, confidentiel, etc.

Certaines règles de statut sont directement issues du schéma de FA : tout élément clé est non modifiable et doit en plus avoir une valeur unique. Tout élément portant l'option *nonapplicable* et qui a été appliqué doit avoir une valeur...

Exemple 5.11 :

```
WHEN affecter(?d.id_mission, ?i);
BEFORE
si( existe(?d.id_mission), annuler_evt;);
FA( DEM, UNIQUE sélection(demande_mission d, =(?d.id_mission, ?i)));
si( existe(DEM), annuler_evt);
```

Cette règle exprime que la valeur de l'élément "id_mission" est unique et non modifiable ("id_mission" est décrit comme clé). La règle de l'exemple 5.7 donne à l'élément "id_mission" le statut obligatoire. Ces deux règles expriment la sémantique inhérente au schéma du FA. En particulier, elles décrivent que l'élément "id_mission" est clé et doit obligatoirement avoir une valeur unique et non modifiable. Elles peuvent être définies automatiquement à la création d'un schéma de FA "demande_mission". Pour cela il faut définir des méta_règles décrivant la sémantique du concept de schéma de FA. Nous laissons ce problème ouvert.

Dans de nombreux formulaires, lorsqu'un champ reçoit une valeur, celle-ci n'est plus modifiable.

Exemple 5.12 :

```
WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s)
BEFORE
si( existe(?d.signature_responsable),
   { si ( non(null(?d.signature_responsable)),
        { message('ce champ ne peut pas être modifié'); annuler_evt; }
   } );
```

Cette règle permet une affectation de valeur à l'élément "signature_responsable" si celui-ci a une valeur nulle inonnue. Dès qu'il a reçu une valeur (non nulle), il devient impossible de lui affecter une nouvelle valeur. Cette règle décrit le fait que l'élément a le statut : non modifiable.

Une opération courante sur un formulaire est d'interdire la modification d'un ou plusieurs de ses éléments dès qu'un élément (signature, tampon, date, ...) est valué. Supposons que l'on veuille interdire toute modification des éléments d'une demande de mission signée. Pour chaque élément, on doit écrire une règle (du type de celle de l'exemple 5.12) dont l'effet est de le rendre non modifiable si "signature_responsable" a une valeur non nulle. En fait ce que l'on veut exprimer, c'est que l'utilisateur n'a plus le droit d'écrire sur les éléments d'une demande de mission signée. Il serait donc intéressant de pouvoir décrire en une seule règle que ces éléments ne disposent plus d'un mode d'accès en écriture. Cela laisse supposer que l'on associe à chaque élément de FA, un **mode d'accès** par utilisateur, pouvant être modifié. Le contrôle de ce mode sera fait par GIF dès qu'une opération est demandée sur l'élément.

On dispose de la fonction :

> **changemode (<list_élément> , <mode> [, <utilisateur>])**
> qui modifie les modes d'accès aux éléments d'une occurrence pour certains utilisateurs.

1- L'argument **<list_élément>** précise le(s) élément(s) dont on modifie les droits. Les règles syntaxiques pour cet argument sont :

<list_élément> ::= <trajet> | '(' <trajet> {';' <trajet>}$^{+}$ ')'
| **allof** <trajet> **except** <list_élément>

<trajet> désigne un élément E. <élément> est un nom d'élément pour le FA

défini par E. On rappelle qu'un trajet comportant seulement un identificateur d'occurrence désigne l'occurrence elle-même et correspond à l'élément racine de son schéma.

2- L'argument <**mode**> est le nouveau droit pour les éléments, désignés par <list_élément>.

Un FA est un objet hiérarchique dont les éléments simples (feuilles) sont complexes et ont leur propre structure (texte, graphique, image). On se trouve confronté à deux niveaux de structure et de sémantique : pour le FA ou les éléments complexes et pour les éléments simples (feuilles). Comme, il a été montré dans [BAL 84], on a deux types de modification possible pour un élément : écriture FORTE (la sémantique de l'élément est changée) et écriture FAIBLE (la sémantique n'est pas modifiée : correction de fautes dans un texte par exemple). Il est clair que pour ce dernier type de modification, les règles sémantiques ne seront pas validées. Pour la lecture d'un élément, on dispose de deux modes de lecture : lecture EXACTE d'un élément ne supportant aucune modification concurrente de l'élément, lecture APPROCHEE autorisant une écriture FAIBLE simultanée. Dans [BAL 84] ces modes opération ont été identifiés pour la spécification d'un mécanisme de verrous utiles à la gestion des accès concurrents à un document.

Nous ne nous préoccupons pas pour l'instant de la gestion de tels verrous puisque dans le cadre notre étude nous n'avons pas d'accès concurrents à une occurrence de FA (contexte mon-utilisateur, mono-formulaire). Lorsqu'on veut manipuler une occurrence, on active une opération qui correspond à lancer une transaction dans laquelle on demande une Lecture EXACTE de l'occurrence. Ensuite pour effectuer les manipulations sur l'occurrence, on demande des écritures FORTE. Les demandes seront acceptées si les modes d'accès associés aux éléments les autorisent. Par exemple une écriture ne sera pas accordée sur un élément si son mode d'accès indique "une lecture seulement".

Dans l'optique d'un contexte plus ouvert pour GIF, nous proposons les modes d'accès : LA (lecture approchée), LE (lecture exacte), EFA (écriture faible) et EFO (écriture forte) et les combinaisons LE-EFA, LE-EFO, LA-EFO, LA-EFO. Pour l'instant dans le cadre de notre étude, nous interpréterons les modes LA et LE comme le mode Lecture, et les modes EFA et EFO comme le mode Ecriture.

L'opération de création (initiale) d'une occurrence de FA donne à tous les éléments le mode : **LE-EFO**

3- L'argument **user** donne les utilisateurs concernés par la modification.

<utilisateur> ::= <ident_utilisateur>
| '{' <ident_utilisateur> (',' <ident_utilisateur>)$^{+}$ '}'
| all except <utilisateur>

Lorsque cet argument n'est pas spécifié, cela signifie que la modification des droits concernent tous les utilisateurs (all) pouvant utiliser le formulaire.

Exemple 5.13 :

WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s);
AFTER
changemode(?d, LE-EFA, all except ?s);
changemode(?d, except ?d.date_mission.retour_mission, LE-EFA, ?s);

La première action de la clause AFTER réalise un changement de mode d'accès sur la racine de l'occurrence ?d (élément "demande_mission") pour tous les utilisateurs sauf l'utilisateur ayant signé la mission. L'élément "demande_mision" est complexe. Le changement est reporté sur ses éléments. Dans la seconde action, on donne le nouveau mode d'accès à l'occurrence ?d (excepté un de ses éléments : "retour_mission") pour l'utilisateur ayant signé la mission.

Cette règle exprime qu'une fois signée, une occurrence de "demande_mission" peut être lue et modifiée *faiblement* par l'ensemble des utilisateurs. Seul la personne qui l'a signé conserve le droit de pouvoir modifier *fortement* l'élément "retour_mission". Cet exemple montre bien qu'un élément ("retour_mission") peut avoir des modes d'accès différents selon les utilisateurs.

Portabilité des modes

Il est clair que les opérations de lecture et écriture sur un élément complexe peuvent toujours être considérées comme des lectures et écritures sur ses composants. Un changement de mode sur un élément complexe est automatiquement reporté sur ses composants. Un nouveau mode pour un élément composant sera accepté si et seulement si il n'est pas en conflit avec le mode d'accès attribué à son élément "père".

Le tableau de la Figure 5.2 montre la compatibilité des modes entre un élément père est un élément fils. On constate que l'on peut avoir par exemple, un élément E (agrégat) avec le mode LE-EFO donc les éléments composants sont : E1

avec le mode LE, E2 avec le mode EFO (on ne peut pas manipuler cet élément), E3 avec le mode LE-EFA et l'élément E4 avec le mode LE-EFO.

Père / Fils	LA	LE	EFA	EFO	LE-EFO	LE-EFA	LA-EFO	LA-EFA
LA	1	0	0	0	0	0	1	1
LE	0	1	0	0	0	1	0	0
EFA	0	0	1	0	0	1	0	1
EFO	0	0	0	1	1	0	1	0
LE-EFO	0	0	0	0	1	0	0	0
LE-EFA	0	0	0	0	1	1	0	0
LA-EFO	0	0	0	0	0	0	1	0
LA-EFA	0	0	0	0	0	0	1	1

1 = Compatible - 0 = Incompatible

Figure 5.2 : Compatibilité des modes pour deux éléments père-fils

V.2.3 Règles de valeur

Une règle est dite de **valeur** parce sa validation entraîne la valuation d'un ou plusieurs éléments de FA. Ces éléments sont dits **calculés**. La ou les actions permettant de les valuer appartiennent à la clause AFTER.

La valeur d'un élément calculé est **stockée** dans l'occurrence du FA. On évite ainsi la validation de la règle chaque fois que l'occurrence est manipulée. Ce qui permet une exécution plus performante des requêtes portant sur les éléments calculés. Cette valeur dépend des valeurs des objets utilisés pour son calcul. Ces objets sont nommés objets **sources**. Nous distinguons :

1- les règles de valeur locale pour lesquelles les objets sources sont locaux : ce sont des éléments du FA pour lequel on spécifie la règle.

Exemple 5.14 :

WHEN affecter(?r.frais_mission, ?m); affecter(?r.frais_inscription, ?i);
AFTER affecter(?r.total, +(?m, ?i));

"total" est un élément calculé du FA "remboursement_frais". L'expression de calcul pour cet élément fait intervenir d'autres éléments (sources) du FA : "frais_mission" et "frais_inscription". Par cette règle, l'élément "total" prendra une valeur si "frais_mission" **ET** "frais_inscription" ont tous les deux une valeur. Si on avait voulu une valuation de "total" pour une valuation de "frais_inscription" **OU** "frais_mission", il aurait fallu écrire les deux règles :

1) **WHEN** affecter(?r.frais_mission. ?m);
   AFTER affecter(?r.total, +(?m, ?r.frais_inscription));
2) **WHEN** affecter(?r.frais_inscription. ?i);
   AFTER affecter(?r.total, +(?r.frais_mission, ?i));

Exemple 5.15 :

```
WHEN affecter(?r.voyage.supplements, ?p)
AFTER
affecter(?r.voyage.montant_supplément,
        +(?r.voyage.montant_supplément, ?p.prix_sup));
```

"?p" est une variable qui peut contenir une valeur de supplément, par exemple (TGV, 70). "?p.prix_sup" est la valeur du prix de ce supplément. Le montant total des suppléments est recalculé pour chaque nouvelle affectation de valeur à "suppléments".

2- Les règles de valeur globale pour lesquelles les objets sources sont globaux : ce sont des objets BD (base de données) au sens large. L'expression de calcul de l'élément peut porter sur des éléments du FA et sur des éléments d'autres FA et/ou sur des propriétés d'objets (non FA).

Exemple 5.16 :

```
WHEN créer(demande_mission, ?d); affecter(?d.no_secu, ?n);
BEFORE
FA (PERS, UNIQUE élaguer( sélection( personne p, = (p.no_ss, ?n)) :
                                    échelon, salaire) );
si( non(existe(PERS)),
   { message('personne inexistante');
     annuler_evt;}
   );
AFTER
pour (PERS, ?p)
  affecter(?d.nom_prénom, concat(?p.nom, ?p.prénom);
  affecter(?d.adresse, ?p.adresse);
```

```
   affecter(?d.indice, ?p.indice);
   affecter(?d.fonction, ?p.fonction);
   affecter(?d.mode_paiement, ?p.mode_paiement);
fin;
```

Les actions de la clause AFTER montrent que les éléments "nom_prénom", "adresse", "indice", "fonction", "mode_paiement" sont calculés globalement à partir des propriétés de l'objet source "personne" stocké dans la base de données.

Exemple 5.17 :

```
WHEN créer(remboursement_frais, ?r); affecter(?r.id_frais, ?i);
BEFORE
FA( MISSION, UNIQUE sélection(demande_mission, = (id_mission, ?i)));
si( non(existe(MISSION)),
   { message('pas de mission correspondante');
     annuler_evt;
   });
AFTER
pour (MISSION, ?m)
   affecter(?r.envers.nom, ?m.nom_prénom);
   affecter(?r.envers_paiement, ?m.mode_paiement);
   affecter(?r.envers.fonction, ?m.fonction);
   affecter(?r.envers.indice, ?m.indice);
   affecter(?r.envers.lieu_mission,
            concat(m.lieu_déplacement.ville, ?m.lieu_déplacement.pays));
   affecter(?r.envers.date_mission,
    interval_temps(?m.date_mission.début_mission,  ?m.date_mission.retour_mis-
    sion));
fin;
```

Les éléments "nom", "paiement", "fonction", "indice", "lieu_mission", et "date_mission" sont calculés globalement à partir des valeurs de l'occurrence de "demande_mission" (dans la base) pour laquelle on veut le remboursement.

LIEN élément calculé - objets sources

1. ***Elément calculé localement***

Supposons pour l'exemple 5.14, qu'une première validation de la règle ait permis de valuer l'élément "total" à 1000F (200f de "frais_mission" et 800F de "frais_inscription"). Si on modifie "frais_inscription" (= 700F), la règle est validée une nouvelle fois et l'élément "total" = 900F. Si on modifie ensuite "frais_mission" = 300F, la règle validée une nouvelle fois donne la valeur 1000F à l'élément "total".

On voit que la règle de valeur fonctionne comme la contrainte "total" = "frais_mission" + "frais_inscription". La validation de la règle permet de s'assurer que la contrainte est toujours vérifiée. Plus généralement une règle de valeur locale correspond à une contrainte :

"élément calculé = expression de calcul".

Il existe un **lien de contrainte** entre l'élément calculé et ses éléments sources qui induit une dépendance de valeur forte. Si l'on se refaire à la terminologie des bases de données relationnelles, l'élément calculé localement est un **photographie** dont le rafraîchissement est automatique au moment où se produit une mise à jour d'un de ses éléments sources.

2. *Elément calculé globalement*

Les objets sources d'un élément calculé globalement sont : (1) des éléments du FA et des objets BD, (2) des objets BD.

L'exemple 5.16 correspond à ce dernier cas. les éléments calculés "nom_prénom", adresse", "fonction", ... sont des photographies sur l'objet personne prises au moment où l'événement "affecter (?d.no_secu, ?n) se produit. Si cet événement se reproduit, on réexécute la règle et on rafraîchit les éléments calculés.

Par contre une modification de "la personne" source est sans effet sur les éléments sauf si il existe une règle (de report) associée à l'objet "personne" décrivant comment l'opération doit être reportée. Le rafraîchissement n'est donc pas automatique au moment d'une mise à jour des objets sources. On peut donc avoir une valeur d'élément calculé en retard par rapport à ses valeurs sources.

Nous dirons qu'il existe un **lien de relation** entre un élément calculé globalement est ses objets sources. Il induit une dépendance de valeur faible. Pour les règles de valeur globale avec des objets sources locaux (éléments du FA), le rafraîchissement par rapport à une mise à jour M de ces objets (éléments) sera automatique si l'événement correspondant à M appartient à la clause WHEN.

Manipulation d'éléments calculés.

Dans la section précédente, nous avons montré les liens existants entre un élément calculé et ses objets sources. Dans les deux cas de figure, l'élément est considéré comme une photographie. On sait qu'une modification de photographie est interdite.

Pour interdire cette modification il faut que le système dispose pour chaque FA de la liste de ses éléments calculés. Il faudrait donc au moment de la définition d'une règle indiquer sa qualification "de valeur" et interpréter cette définition pour obtenir l'identification du ou des éléments calculés. Cette dernière phase n'est pas triviale. De plus on introduirait un "typage" des règles qui enleverait au concept de règle sa généralité.

Par ailleurs, on sait que toute manipulation faite sur un formulaire correspond à effectuer une opération spécifique qui définit pour l'utilisateur l'ayant activée les actions qui lui est possible de réaliser. Par le biais des opérations spécifiques, on peut donc facilement interdire la manipulation d'éléments calculés : pas d'actions de mise à jour de ces éléments dans la spécification des opérations. On peut également autoriser la manipulation d'un élément calculé. Si tel est le cas, l'élément calculé n'est plus une photographie mais un **objet dérivé matérialisé**, pouvant évoluer indépendement de ses objets sources.

1. *Elément calculé localement*

Dans le cas d'un élément calculé localement, nous avons montré que la règle de valeur fonctionne comme une contrainte : "élément calculé = expression de calcul" vérifiée pour toute manipulation de sa partie droite. Si l'on veut conserver une lien de contrainte entre les objets, il faut que cette contrainte reste vérifiée pour toute manipulation de la partie gauche.

Considérons que l'on autorise l'utilisateur à affecter une valeur à l'élément "total". Si frais_mission" est de 200F et "frais_inscription" de 800F, "total" a été valué à 1000F (règle de l'exemple 5.14). L'utilisateur modifie "total" (= 1500F). Pour maintenir un lien de contrainte entre "total" et ses éléments sources, il faut trouver une solution à l'équation :

1500 = frais_mission + frais_inscription.

Comment ? En définissant explicitement une règle de report qui décrit le processus de réalisation de cette équation pour aboutir à une solution. Dans les cas

d'équations complexes, il n'est pas possible de donner une règle de report. L'élément n'est pas manipulable.

Si l'on veut manipuler un élément calculé localement et maintenir une cohérence forte entre sa valeur et celles de ses objets sources, il faut **obligatoirement** spécifier comment une mise à jour doit être reportée sur les éléments sources.

2. ***Elément calculé globalement***

Pour les éléments calculés globalement, la dépendance de valeurs avec ses éléments sources est faible. La manipulation d'un élément calculé globalement est donc possible sans avoir obligatoirement un report à faire sur ses objets sources. Si un report de manipulation est désiré, il suffit de décrire comment il doit être fait dans une règle de report associé à l'élément.

V.2.4 Règles de report

Une règle de report définit comment certaines mises à jour effectuées sur un formulaire doivent être reportées sur ses objets sources. Dans "objets sources", nous englobons les objets sources pour les éléments calculés et les éléments virtuels.

Règle de report pour élément calculé

Considérons les éléments calculés globalement par la règle de l'exemple 5.16. Supposons que l'on autorise une modification de l'adresse d'une personne à partir du FA "demande_mission". Il faut écrire le règle de report suivante :

Exemple 5.18 :

```
WHEN créer(demande_mission, ?d);affecter(?d.adresse, ?a);
BEFORE
FA( PERS, UNIQUE sélection( personne, = (no_ss, ?d.no_secu));
si( non(existe(PERS)), annuler_evt);
AFTER
pour (PERS, ?p)
  affecter(?p.adresse, ?a);
fin;
```

Il n'est pas nécessaire de vérifier l'existence d'une valeur pour "no_secu" puisque par la règle 5.7 cet élément est valué de suite après la création d'une nouvelle occurrence de FA.

Règle de report pour élément virtuel

Dans la section III.3 nous avons défini ce qu'est un élément virtuel : vue sur des objets BD au sens large. La manipulation d'un élément virtuel sera propagée sur ses objets sources. Cette propagation se fait par l'exécution de règles de report.

Considérons l'élément virtuel "car_person" du FA "professeur" défini en III.3. Supposons que l'on autorise une modification des propriétés "adresse" et "indice" d'une personne (avec la fonction professeur) via les éléments "car_person.adresse" et "car_person.indice" du FA "professeur". Toute modification de ces éléments est reportée dans l'objet personne, source de l'élément virtuel "car_person".

Exemple 5.19 :

```
WHEN créer(professeur, ?pf);
        affecter(?e.car_person.adresse, ?a); affecter(?e.car_person.indice, ?i);
AFTER
FA( PERS, UNIQUE sélection( personne, et ( =(fonction, 'professeur'),
                                           =(no_ss, ?pf.car_person.no_ss) ) ) );
pour (PERS, ?p)
   affecter(?p.adresse, ?a);
   affecter(?p.indice, ?i);
fin;
```

Exemple 5.20 :

```
WHEN supprimer(?pf.car_person); fin_T;
AFTER détruire(professeur, ?pf);
```

Cette règle exprime que toute suppression de valeur pour l'élément "car_person" d'une occurrence **pf** (de professeur) entraîne sa destruction de la base.

Ceci traduit bien le fait que le FA "professeur" est un schéma externe sur l'objet "personne", objet source pour l'élément virtuel "car_person". L'existence d'une occurrence pour ce FA est fortement liée à l'existence d'une occurrence dans "personne". Lorsqu'une personne **P** est supprimée, on supprime la valeur de

l'élément "car_person" dans l'occurrence **pf** de "professeur" liée à **P**. les valeurs restant dans cette occurrence n'ont plus de sens. Il faut détruire **pf**.

V.2.5 Règles d'exception

Nous présentons ici des exemples de règles autorisant de la "tolérance" dans la connaissance attachée à un FA. Nous montrons ainsi qu'il est possible d'avoir des données exceptionnelles dans une occurrence de FA. Dans [ESC 87], a été introduit le concept de tolérance sémantique pour la spécification de diverses contraintes (de tolérance) permettant la prise en compte des types distints d'exceptionnalité. Ont été définies :

1- des contraintes de **tolérance nulle** : Les règles de contrainte présentées dans les sections précédentes représentent des contraintes rigides avec une tolérance nulle.

2- des contraintes de **tolérance totale** : Une règle de contrainte avec une clause BEFORE contenant seulement des actions qui produisent un "message" correspond à une contrainte-commentaire avec une tolérance totale. Par exemple, on peut borner le montant total d'une occurrence du FA "rembousement_frais".

Exemple 5.21 :

```
WHEN affecter(?r.récapitulatif.total, ?t);
BEFORE
si( >(?t, 50000),
   { message('Montant du remboursement supérieur à 50000F
              Veuillez confirmez qu'il s'agit d'une exception');
     wait_evt( affecter( ?t.EXCEPTION ) );
     si( = (CWE, 0), annuler_evt,
                     si( =(?t.EXCEPTION, 'non'), annuler_evt )
       );
   });
```

On signale que l'on outrepasse la contrainte ("total" > 50000F") et on demande confirmation à l'utilisateur. On utilise l'élément particulier EXCEPTION. Toute affectation de valeur à cette ELEMENT correspond à une mise à jour du contexte de manipulation de l'occurrence(?r) et de la donnée (?t).

3- des contraintes de **tolérance selon utilisateur** : on autorise certains utilisateurs à outrepasser les contraintes définies. Par exemple, exceptionnellement et pour des raisons de confidentialité, on autorise une occurrence de mission a ne pas contenir de valeur pour ses éléments "début_mission" et "fin_mission". Ces éléments doivent normalement avoir une valeur pour que l'occurrence soit acceptée. Si l'un des éléments ou les deux éléments n'ont pas de valeur, l'occurrence est alors considérée comme exceptionnelle. La manipulation d'une telle occurrence sera réservée à l'utilisateur "toto". On écrira la règle suivante :

Exemple 5.22 :

WHEN créer(demande_mission); fin_T;
BEFORE
si(et (non (existe(?d.date_mission)), non (=(?USER, 'toto'))),
{ message('Il faut donner une valeur aux éléments de *date_mission*');
annuler_evt;
});
AFTER
si(non(existe(?d.date_mission)),
affecter(?d.EXCEPTION, oui)
);

4- des contraintes de **tolérance selon le nombre d'exceptions** : qui autorisent le contrôle des exceptions tolérées. On peut ainsi représenter le fait que l'on autorise des exceptions comme des cas rares. Par exemple, on peut indiquer que l'on accepte pas plus de 10 occurrences du FA "demande_mission" n'ayant pas de valeur pour "date_mision". On écrit alors la règle :

Exemple 5.23 :

WHEN créer(demande_mission, ?d); fin_T;
BEFORE
si(non(existe(?d.date_mission),
{ FA(MIS_EX, sélection(demande_mission, = (date_mission, ϕ)));
si(>(count (MIS_EX), 10), annuler_evt);
})

5- des contraintes de **tolérance temporelle** : pour la prise en compte de données exceptionnelles non persistantes. Par exemple autoriser une demande de mission sans valeur pour son élément "motif" pendant au plus 10 jours à compter de la date de sa dernière utilisation.

Exemple 5.24 :

WHEN créer(JOUR, ?j);
BEFORE
FA(MIS, sélection(demande_mission, et (= (motif, ϕ),
=(DATE, subtime (?j, 10days))));
AFTER
pour (MIS, ?m)
  créer(REJET, ?m); détruire(demande_mission, ?m);
fin;

On utilise ici le FA particulier JOUR qui représente les différents jours "du système". Chaque fois que l'on change de jour, une occurrence est crée dans ce FA. L'élément particulier DATE représente la date de mise à jour de l'occurrence. Elle est stockée dans son contexte de manipulation. Les occurrences exceptionnelles ne pouvant plus persister dans la base sont sauvegardées dans l'objet REJET.

Conclusion :

Par ces exemples de règles d'exception, nous avons voulu montrer que le concept de règle permet de représenter les cas divers d'exceptionnalité proposés dans [ESC 87]. Nous constatons que pour décrire ces règles, il nous faut utiliser des informations nouvelles autres que celles présentes dans l'occurrence. En particulier, il est nécessaire de pouvoir utiliser explicitement le contexte de manipulation d'une occurrence.

V.3 Cohérence des règles

Une phase importante dans la définition des règles d'un FA est la vérification de leur cohérence. Cela consiste à s'assurer :

- que la règle n'est pas contradictoire ou incohérente.
- qu'il n'y a pas de conflits entre les règles.
- qu'il n'y a pas de risque de cycles dans les définitions, ce qui introduirait une exécution sans fin des règles.

V.3.1 Règle contradictoire

Une règle est contradictoire lorsque :

(1) un événement apparaît plus d'une fois dans la clause WHEN.

(2) un événement porte sur un objet qui n'a pas été défini.

(3) les actions des clauses BEFORE et AFTER sont contradictoires. Elles sont définies sur des objets non existant ou bien les actions conduisent *toujours* à une non confirmation de l'événement ayant rendu la règle active.

Exemple de contradiction dans la clause BEFORE

Soit la règle suivante :

WHEN créer (professeur, ?p); affecter(?p.car_person.salaire, ?s);
BEFORE si (non (et (>(?s, 9000), <(?s, 3000F))), annuler_evt)

qui exprime que la valeur donnée au salaire d'un professeur doit être supérieure à 9000F et inférieure à 3000F. La partie condition de la fonction **si** est toujours évaluée à "VRAI" quelque soit la valeur du salaire. Donc une affectation de valeur à l'élément "salaire" du FA "professeur" sera toujours rejetée.

Cet exemple montre que l'incohérence d'une action n'est pas facile à détecter. Ici, la sémantique de la fonction **si** est respectée : les parties *condition* et *actions-alors* sont correctes du point de vue syntaxique et de plus évaluables. Dans le cas présent, l'action serait cohérente si au moins une de ses exécutions (avec une valeur pour l'élément "salaire") ne réalise pas "annuler_evt". Pour tester la cohérence, il faut disposer de valeurs de salaire **significatives**. Le problème rencontré est de savoir comment déterminer ces valeurs.

On peut également parler de contradiction lorsque dans une clause BEFORE ou AFTER deux actions s'annulent mutuellement. Par exemple, une affectation de valeur suivie d'une suppression de valeur sur le même élément. Ces cas de contradiction sont voulus ou bien correspondent à des erreurs de spécification. Est-il utile de les détecter ?.

V.3.2 Conflits entre règles

Il y a conflit entre deux règles lorsqu'elles doivent s'exécuter en même temps. Elles sont définies avec des clauses WHEN identiques (même liste d'événements) ou avec des clauses WHEN possédant des événements en commun. Une première solution consiste à interdire de telles définitions. Une seconde solution consiste à les autoriser. Supposons les deux règles suivantes :

Règle1 : **WHEN** E_1; E_2;
BEFORE action1;
AFTER action2;

Règle2 : **WHEN** E_1; E_3;
BEFORE action3;
AFTER action4;

Si E_1, E_3, (E_2) se sont produits et que E_2 se produit, ces deux règles deviennent actives. En section V.1.1, nous avons dit que cela correspond à exécuter :

(1)
WHEN E_1; E_2; E_3;
BEFORE action1;
action3;
AFTER action2;
action4;

OU

(2)
WHEN E_1; E_2; E_3;
BEFORE action3;
action1;
AFTER action4;
action2;

Supposons que action2 est "affecter (T1, + (T2, 3))" et action4 est "affecter (T1, + (T2, 5))", l'exécution des règles : Règle1 et Règle2 selon l'approche (1) donne à T1 la valeur $T2+3$ alors qu'avec l'approche (2) T1 = $T2+5$.

La propriété proposée dans [GIB 84] peut être adoptée ici :

Des règles en conflit sont exécutables dans un ordre arbitraire si et seulement si les actions de leurs clauses BEFORE et AFTER (principalement les actions de mise à jour) portent sur des éléments de FA en mutuelle exclusion.

Pour Règle1 et Règle2, action1 et action2 ne doivent pas porter sur le même élément de FA.

V.3.3 Cycles dans les définitions

Une règle devient active lorsque tous les événements de sa clause WHEN se sont produits (un des événements n'est toutefois pas encore confirmé). La réalisation de l'opération liée à l'événement ayant rendu la règle active et la continuation du traitement de la règle (exécution de la clause AFTER) est

fonction de la réalisation des actions de la clause BEFORE. Dans cette clause, on peut avoir des actions demandant la réalisation d'événements. Parmi ces événements, on peut trouver un événement de la clause WHEN. Dans la clause AFTER, on peut avoir également des actions correspondant aux événements de la clause WHEN.

En schématisant, on peut donc se retrouver dans les cas de figure suivants :

(1) **WHEN** E_1; E_2; ...; E_n;
BEFORE E_1; E_2; ...; E_n;
AFTER E_1; E_2; ...; E_n;

Le fait de réaliser à nouveau un événement de la clause WHEN dans les clauses BEFORE ou AFTER est problématique. Il provoque une réactivation de la règle. Lorsque dans les clauses BEFORE et AFTER d'une règle, se rencontre une action correspondant un événement de la clause WHEN, on se trouve dans un cas de définition **récursive** aboutissant à un cycle. Une telle définition n'est pas autorisisée.

(2)

Règle1	Règle2
WHEN E_1; E_2;	**WHEN** E_3; E_4;
BEFORE E_3;	**BEFORE** E_1;
AFTER E_4;	**AFTER** E_2;

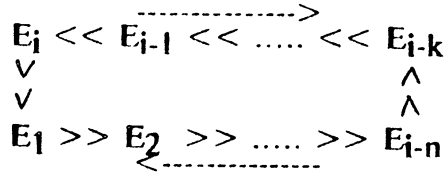
Supposons que Règle1 devient active alors on demande la réalisation de l'événement E_2. Si E_4 s'est produit alors Règle2 devient active. On demande alors la réalisation de E_1 qui va provoquer la réactivation de Règle1. Si E_4 ne s'est produit et E_3 est confirmé alors on passe à l'éxécution de l'action E_4 de la clause AFTER. Règle2 devient active et la demande de réalisation de E_1 peut provoquer une nouvelle activation de Règle1.

Quelque soit les étapes de validation successives, on aboutit toujours à un **cycle**. Il est dû à des définitions de règles qui se référencent mutuellement via leurs événements.

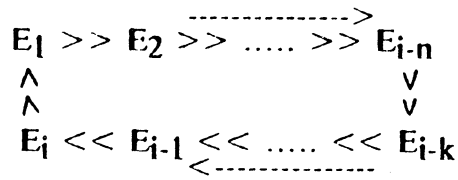
Soit :

- E_i un événement
- la relation << entre deux événements $E_i << E_{i-1}$ qui a pour signification : *l'événement E_i référence* **avant** *l'événement E_{i-1}*.
  E_{i-1} appartient à la clause BEFORE de la règle dont la clause WHEN contient E_i.
- la relation >> entre deux événements $E_i >> E_{i+1}$ qui a pour signification : *l'événement Evt_i référence* **après** *l'événement Evt_{i+1}*.
  Evt_{i+1} appartient à la clause AFTER de la règle dont la clause WHEN contient Evt_i.

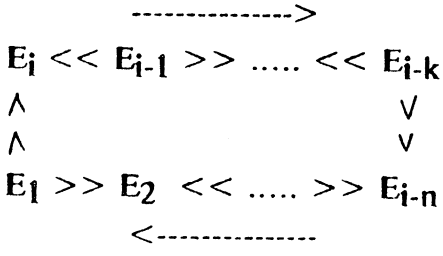
Par analyse des règles à leur définition, on interdit les cas de définitions cycliques suivants:

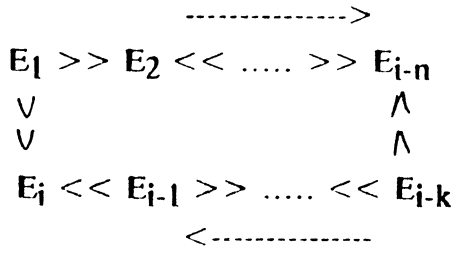


cycle par référence **avant**



cycle par référence **après**





cycle par référence **avant** et **après**

V.3.4 Propriétés particulières des règles pour élément virtuel

Toute règle définie sur un élément virtuel de FA doit être cohérente. De plus, elle doit vérifier les conditions suivantes :

- ne pas contenir dans sa clause BEFORE des actions en contradiction avec celles des règles définies pour ses objets sources

- ne pas contenir dans sa clause AFTER des actions sur d'autres objets que le FA. Cette propriété permet de limiter les effets de bord d'une mise à jour d'objet source aux occurrences avec lesquelles il est lié. Elle pourra être vérifiée lors de l'analyse de la définition de la règle.

Cas de contradiction :

Il peut exister une contradiction entre une action d'une clause BEFORE d'une règle sur élément virtuel et une action d'une règle pour un de ses objets sources. Par exemple :

Règle1 définie pour "personne" :

WHEN affecter (?p. salaire, ?s)
BEFORE si (<(?s, 15000), annuler_evt);

Règle2 définie pour "professeur" :

WHEN affecter (?pf. salaire, ?s)
BEFORE si (<(?s, 10000), annuler_evt);

On pourra donner à un professeur **pf** un salaire de 11000F. Mais le report de mise à jour de salaire ne sera pas acceptée pour la personne **p** correspondante à **pf**.

Pour détecter de telle contradiction, il faut :
- disposer de Règle1 au moment de la définition de Règle2
- disposer de valeurs significatives pour salaire dans "professeur" et vérifier Règle1 avec ces valeurs. Un échec de Règle1 détermine une contradiction.

Compte tenu de la compléxité du problème, on peut adopter la solution suivante :

INTERDIRE toute règle de la forme WHEN BEFORE pour un élément virtuel dont la donnée source est concernée par une règle WHEN BEFORE.

V.3.5 Conclusion

La vérification de la cohérence des règles n'est pas triviale. Certains problèmes identifiés dans les sections précédentes peuvent être évités en proposant un environnement de définition des règles pour un FA qui vérifie :

- pour la clause WHEN :

  * sa description n'est pas identique à celle d'une autre règle.

  * si un de ses événements appartient à la clause WHEN d'une autre règle alors on doit vérifier que les actions de sa clause BEFORE (respectivement sa clause AFTER) ne concernent pas les mêmes objets que ceux des actions de la clause BEFORE (AFTER) de la règle en question. Pour réaliser cette vérification, il faut disposer, pour chaque Règle définie, de la liste des objets qu'elle utilise (ce qui n'est pas simple).

- pour les clauses BEFORE et AFTER :

  * une règle ne peut référencer avant ou après que des événements dont les règles associées ont déjà été définies. C'est une définition "ascendante" des règles. De cette manière, on peut détecter les cycles. Cela nécessite de disposer pour chaque règle de la liste des événements correspondant aux actions de ses clauses BEFORE et AFTER.

Ces vérifications nécessitent la gestion d'une définition de règle en termes d'événements et de données (manipulées par la règle). Elles vont permettre de limiter les risques d'interblocage et de définitions cycliques. Mais le problème des actions contradictoires subsiste.

Pour chaque action définie et plus généralement pour une règle, il faudrait disposer d'un mécanisme permettant de valider la règle à sa définition avec des valeurs représentatives des données qu'elle utilise. On trouve dans [OLI 86] et [NGU 86] une approche pour la vérification sémantique d'une base de Connaissances d'une application (description *logique* des données et de leur sémantique) basée sur la notion d'"échantillon". Pour GIF, cet échantillon correspondrait à une ou plusieurs occurrences de FA. La définition d'une nouvelle règle conduirait à "affiner" cet échantillon pour qu'il reste un *modèle* d'occurrences cohérentes. Le problème soulevé ici reste du domaine de la recherche. La méthode suivie pour la définition des règles ne détecte pas les risques d'incohérence dans une action.

CHAPITRE VI

DYNAMICITE DES SCHEMA_FA

CHAPITRE VI

DYNAMICITE DES SCHEMA_FA

Dans ce chapitre, nous présentons les commandes de GIF autorisant la dynamicité des SCHEMA_FA. On rappelle qu'un SCHEMA_FA possède deux composantes : le schéma du FA (chapitre III) et les règles attachées à ce schéma (chapitre V). Un formulaire est stocké dans une base de données. La commande DEF_FA (chapitre III) correspond à une insertion dans le catalogue de FA : objets de la base stockant les SCHEMA_FA. Pour manipuler un FA, on utilise les opérations présentées au chapitre IV. Avec les commandes proposées dans les sections suivantes, le concepteur pourra :

- supprimer un SCHEMA_FA.
- modifier un SCHEMA_FA c'est à dire
  * modifier le schéma du FA en supprimant et/ou en ajoutant des éléments.
  * modifier les règles en supprimant des règles existantes et/ou ajoutant de nouvelles règles.

Ces commandes ne sont pas sans conséquences sur la valeur du FA (ensemble d'occurrences). Pour chacune d'elles nous précisons ses effets sur le catalogue de FA et sur le FA lui même (sa valeur).

Notre modèle de formulaires ne se limite pas au concept de FA. A chaque FA sont associées une ou plusieurs Présentations. Le concept de Présentation est présenté plus en détail dans le chapitre VII. Il permet de décrire un format "externe" pour un FA et les opérations spécifiques autorisées sur le FA au travers du format. Une Présentation est liée à un FA et toutes mises à jour d'un SCHEMA_FA aura également des effets sur ses Présentations.

VI.1 Suppression de SCHEMA_FA

La destruction d'un FA se fait par la commande suivante :

DEL_FA <nom_FA>.

<nom_FA> est le nom désignant un FA. Cette opération implique la destruction du schéma de FA, de ses règles et de toutes ses occurrences. Elle a pour effet :

- les règles référençant le FA ne peuvent plus être évaluées. Ces règles seront supprimées.
- le(s) Présentation(s) définies pour le FA ne sont plus utilisables. Elles seront détruites.

VI.2 Modification de SCHEMA_FA

VI.2.1 Modification de schéma

La modification d'un schéma de FA consiste à ajouter ou supprimer un élément de FA. Ceci correspond à rajouter ou supprimer un noeud dans l'arbre représentant le schéma. Pour désigner un noeud, on utilise un **chemin** dans l'arbre. Nous ne donnons pas la définition formelle de ce chemin. C'est une suite de noms séparés par des points. Par exemple pour le schéma du FA "étudiant" : adresse.ville désigne le noeud (l'élément) "ville".

1- Suppression d'éléments de FA

La suppression d'éléments de FA se fait par la commande :

DEL_ELEM <nom_FA> ':' <chemin> ';' { <chemin> ';' }*

La suppression d'un élément entraîne la suppression de sa description dans le catalogue de FA. On obtient un nouveau schéma de FA qui traduit une nouvelle représentation (pour les occurrences) dans la base.

Une telle commande a de l'effet sur :

- les occurrences existantes avant la suppression. Celles-ci doivent donc être stockées selon le nouveau SCHEMA_FA. Pour réaliser cette transformation des occurrences, on peut utiliser l'opération **élaguer**. Mais on perd de l'information !.

- les règles du FA où les éléments sont référencés. En effet ces règles ne sont plus évaluables, elles seront détruites.
- les formats dans lesquels les éléments ou une partie de ces éléments apparaissent. On détruit la description de leurs formats.
- les opérations spécifiques dans lesquelles les éléments sont référencés. On demande au concepteur de modifier ces opérations sinon elles seront détruites.

La suppression d'éléments doit être faite avec précaution car on perd de l'information. De nombreux formulaires ont une valeur juridique et la suppression de valeurs de champs liées à une modification de schéma n'est pas envisageable. On préférera construire un nouveau SCHEMA_FA. Une autre solution consiste à conserver les anciennes occurrences et leurs significations (SCHEMA_FA) comme une ancienne version du formulaire.

2- Ajout d'éléments de FA

L'ajout d'éléments à un schéma de FA correspond à rajouter des noeuds dans l'arbre correspondant au schéma. Il faut préciser (1) à quel niveau de la hiérarchie on veut ajouter les éléments, (2) leurs schémas. Selon le constructeur définissant le noeud auquel on veut rattacher les éléments, l'ajout n'est pas possible. On autorise un ajout d'élément dans un schéma obtenu avec les constructeurs agrégat et choix. On peut donc agréger de nouveaux éléments à un noeud agrégat ou bien proposer de nouvelles alternatives à un noeud choix.

ADD_ELEM <nom_FA> AFTER <chemin>

'set' <déf_ajout> ';' { <déf_ajout> ';' }*

<déf_ajout> := <définition_schéma>
[VALUE <expression_valeur>]

Dans la clause AFTER, on donne un <chemin> dans le schéma de nom <nom_FA> ou le mot clé TOP. On utilise TOP lorsqu'on veut ajouter des éléments après la racine de l'arbre représentant le schéma du FA (agrégat ou choix). <chemin> désigne un élément de FA dont le schéma doit bien sûr être de type agrégat ou choix. On donne ensuite les définitions des éléments que l'on veut

rajouter en utilisant la syntaxe proposée dans la commande DEF_FA : on donne son nom et sa description. On peut également donner des options aux éléments.

La clause VALUE est utilisée lorsqu'on désire affecter une valeur à l'élément ajouté. <expression_valeur> est :
- une valeur d'élément non liée à une occurrence particulière : elle est décrite par la partie <chemin> d'un trajet (voir section IV.2.4),
- une fonction définie sur des valeurs d'éléments non liées,
- une FA_expression d'intérrogation.

L'exécution d'une commande ADD_ELEM transforme le schéma du FA : on ajoute au catalogue les nouveaux schémas en tenant compte de leurs liens avec le FA qu'ils complètent. De même que pour la suppression d'un élément, cette opération d'ajout implique une restructuration des occurrences existantes du FA. Chacune des occurrences est transformée en ajoutant à son schéma le ou les éléments et en leur donnant la valeur ϕ ou une valeur qui dépend des options données aux éléments et du contenu de la clause VALUE :

* si un élément a été défini avec une option *default*, on lui affecte la valeur par défaut.
* si un élément a été défini, avec les options *nullallowed* et *nonapplicable*, on lui affecte la valeur nulle correspondante.
* si un élément a été défini avec une clause VALUE, on lui donne une valeur qui est le résultat de l'application d'un <chemin> à l'occurrence traitée, le résultat d'une fonction ou d'une FA_expression appliquée à l'occurrence traitée. Cette valuation peut nécessiter des renommages d'éléments qui seront réalisés implicitement.

Remarque : Dans le cas d'un ajout d'élément E à un FA choix F, on n'accepte pas de clause VALUE. Les occurences existances ne sont pas modifiées. Si pour une occurrence particulière on veut donner à F le schéma de l'élément E, on devra modifier explicitement l'occurrence.

Exemple :

ADD_ELEM étudiant **AFTER** TOP
set moyenne_note : entier VALUE moyliste (notes);

"notes" est l'expression d'un chemin dans le FA "étudiant". L'exécution de la commande transforme le schéma "étudiant" : le noeud racine de ce schéma a un nouveau noeud fils "moyenne_note". Ensuite, on transforme de la même manière

chaque occurrence d'"étudiant" identifiée par <id_occ>, et on affecte au nouvel élément une valeur qui est le résultat de "moyenliste(<id_occ>.notes)".

VI.2.2 Modification des règles

Nous avons vu que dans la définition d'une règle, on utilise essentiellement des événements et des FA_expressions. Ces expressions sont construites avec des FA_opérations et des opérations particulières sur les types de données (texte, chaîne, image), des fonctions de programmation, des fonctions utilisateurs. Ces fonctions sont définies par le concepteur du formulaire et doivent pouvoir être détruites ou modifiées.

1- suppression et modification de fonctions utilisateurs

La destruction d'une fonction se fait par **DEL_FONC <nom_fonction>**. Une telle opération a pour effet de rendre non évaluables les règles dont l'expression utilise la fonction. Ces règles seront supprimées.

La modification d'une fonction consiste à modifier le programme la réalisant. Elle se fait par la commande :

MOD_FONC <nom_fonction> WITH <nom de nouveau programme>.

Compte tenu de la sémantique d'un formulaire (moyen de transmettre de l'information à un instant donné d'une manière non ambiguë et difficilement modifiable par la suite), nous laissons au concepteur le soin de décider de l'effet de la modification d'une fonction. On peut :

- réexécuter toutes les règles utilisant la fonction modifiée, pour les occurrences existantes de FA. Cette opération revient à effectuer une modification d'occurrences qui peut influer sur leur état de cohérence et de complétude.
- ne pas réexécuter les règles. La création ou la modification d'occurrences pour un FA possèdant des règles utilisant la fonction modifiée se fera en appliquant le nouveau programme. Dans certains cas, pour les anciennes occurrences on veut pouvoir les modifier en continuant à appliquer l'ancien code de la fonction. Ceci sous-entend qu'il faut laisser au concepteur la possibilité de lier une occurrence à une fonction.

2- Suppression d'une règle

Chaque règle définie pour un FA porte un nom unique (nom_règle). Pour supprimer une ou plusieurs règles, on utilise la commande :

DEL_R <nom_FA> : <nom_règle> ';' { <nom_règle> ';'}*

Les effets :

- La destruction d'une règle de valeur pour un élément indique que sa valeur initiale ne sera plus obtenue par calcul mais sera donnée par l'utilisateur.
- La suppression d'une règle de contrainte (pour l'expression d'une contrainte devant être vérifiée par une occurrence de FA pour pouvoir exister) peut avoir pour conséquence de rendre cohérente une occurrence qui ne l'était pas.
- La suppression d'une règle d'exception peut avoir pour conséquence de rendre incohérente une occurrence jusqu'alors considérée comme exceptionnelle. Cet état d'incohérence n'est pas admis dans GIF.

3- Ajout d'une règle

Ajouter une règle correspond à définir une nouvelle règle pour un FA. Dans le processus de définition d'un SCHEMA_FA, la description de sa composante dynamique consiste à réaliser des commandes DEF_R (définition d'une règle). Dans cette commande, on n'a pas à préciser pour quel schéma de FA on définit la règle. La commande ADD_R pour l'ajout d'une règle est une variante de la commande DEF_R. Elle a la forme suivante :

ADD_R <nom_FA> ':' <définition_règle> ';' {<définition_règle>}*

où <nom_FA> est le nom d'un schéma de FA
<définition_règle> est une commande DEF_R de définition de règle (chapitre V)

Pour traiter cette commande, on fait comme si on était dans le processus de définition du SCHEMA_FA. De cette manière, on peut vérifier la cohérence d'une nouvelle règle en tenant compte du schéma de FA et des règles déjà existantes pour ce schéma.

Après ajout d'une nouvelle règle, on peut :
- ne rien faire
- exécuter la règle pour toutes les occurrences de FA existantes.

Ce qui a pour effet :

- s'il s'agit d'une règle de valeur (pour valuer un ou des éléments), on peut avoir alors avoir des modifications de valeurs pour les occurrences. Ceci peut avoir des effets sur l'état de cohérence et de complétude des occurrences modifiées.

- s'il s'agit d'une règle de contrainte, on peut avoir des occurrences jusque là cohérentes qui deviennent incohérentes. Il n'est pas possible de maintenir ces occurrences comme formulaires.

- s'il s'agit d'une règle d'exception, on peut alors voir des occurrences jusque là dans un état non exceptionnel devenir exceptionnelles.

VI.3 Conclusion

Nous avons vu que la modification d'un SCHEMA_FA entraîne des effets pas toujours souhaitables (incohérence d'occurrences, perte d'informations, etc). De manière générale, on peut dire que : (1) une modification de schéma de FA reflète une évolution structurelle d'un formulaire (2) une modification de règles de FA reflète une évolution de la sémantique d'un formulaire. On peut voir une modification de SCHEMA_FA comme le reflet d'une nouvelle version d'un formulaire papier se traduisant par une nouvelle version de ses composantes : schéma et règles. Partant de cette idée de version, on peut envisager de voir l'objet (appelons le CAT_SCH) utilisé pour stocker les SCHEMA_FA comme un historique [BUI 86, ADI 87]. Ce qui permettrait de maintenir les valeurs successives d'un SCHEMA_FA au cours du temps. Les valeurs de CAT_SCH sont des couples (vi, ti) signifiant qu'à l'instant ti un SCHEMA_FA a pris la valeur vi (description d'un schéma et des règles). Cette valeur vi est appelée version temporelle de CAT_SCH.

L'objet CAT_SCH peut être un historique :

(1) à versions manuel : le concepteur au vue des effets des modifications d'un SCHEMA_FA décide de l'historiser.

(2) à versions successives : tous les valeurs succesives d'un SCHEMA_FA sont maintenues.

Considérons ce deuxième cas. A chaque SCHEMA_FA $\underline{F}$ = <S, R> est associé un FA **F** (ensemble d'occurrence). $\underline{F}$ a été défini à l'instant t. On modifie $\underline{F}$ à l'instant t' (> t). Le nouveau SCHEMA_FA, $\underline{F'}$ est stocké dans CAT_SCH sous la forme ($\underline{F'}$, t') et l'ancien SCHEMA_FA ($\underline{F}$, t) devient valeur de la zone historique pour CAT_SCH. Il est alors clair que la manipulation du FA **F** ne pourra se faire que si l'on maintient un lien avec le SCHEMA_FA $\underline{F}$. Ceci sous-entend qu'il faut également mettre **F** dans la zone historique.

Nous avons montré également les effets d'une modification de SCHEMA_FA sur les Présentations associées au FA. Là aussi, il faut songer à "historiser". De même les modifications de fonctions utilisateurs peuvent être vues comme des reflets de versions successives d'une fonction.

En conclusion, si l'on veut offrir la possibilité de modifier un SCHEMA_FA, sans perdre de l'information et sans effet de bord indésirable, on peut aborder le problème en disant que l'on crée une nouvelle version du SCHEMA_FA. Pour maintenir et gérer ces versions, on peut s'appuyer sur la notion d'historique [BUI 86]. Il faut alors offrir des mécanismes dans GIF permettant de gérer les différentes versions d'un formulaire. Par exemple à l'utilisation d'un formulaire (objet sur écran) on peut demander à l'utilisateur quel version il veut utiliser (par défaut, la plus récente). La persistance des versions doit pouvoir être contrôler : au bout d'une certaine période, on peut ne plus vouloir considérer une certaine version. Il faut offrir une commande pour "désactiver" une version. Pour certaines requêtes sur un FA, on peut avoir besoin de retrouver des occurrences dans différentes versions. Supposons que l'on soit en train d'utiliser la version N du formulaire Form1. Toute requête émise depuis Form1 sera d'abord traitée sur le FA *F1* sous-jacent. Si par contre à l'émission de la requête, l'utilisateur demande à ce que celle-ci s'applique à la version I (ou ALL) alors on recherchera les occurrences correspondantes dans I (ou toutes les versions). La présentation de ces occurrences ne se fera pas avec Form1. Il faut alors prévoir pour chaque formulaire des opérations spécifiques permettant de passer d'une version à l'autre du formulaire ("réel").

CHAPITRE VII

LA PRESENTATION

What you sketch is what you get

CHAPITRE VII

LA PRESENTATION

Dans ce chapitre, nous présentons les éléments pour la **Présentation** d'un FA. Un FA peut avoir différentes Présentations. Une Présentation admet deux composantes : un format et des opérations spécifiques.

Le format est la spécification de la représentation externe d'un formulaire abstrait pour un outil de restitution particulier. En effet, on peut définir un format pour présenter une occurrence de FA comme un message parlé et un second format avec des informations complémentaires pour voir cette occurrence comme une image sur écran. Dans cette thèse nous nous limitons aux formats pour affichage sur écran. Nous donnons dans la section VII.1 la définition d'un format d'affichage.

Les opérations spécifiques codent la connaissance attachée au format. Elle décrivent les actions "réelles" faites sur le format (remplir, valider, rehercher, etc). Dans la section VII.2, nous précisons la notion d'opérations spécifiques. Un FA avec une certaine présentation, matérialisé sur un écran constitue le formulaire "réel" : ce que l'utilisateur va voir. La section VII.3 donne les opérations qui ont été définies pour manipuler une présentation. Finalement dans la section VII.4, nous donnons quelques exemples de formulaires tels qu'ils se présenterons sur un écran.

VII.1 Le format (FT)

Le format donne des règles générales pour la représentation externe d'un FA sur un écran. C'est un mécanisme pour la matérialisation d'occurrences qui "réconcilie" le concept de FA avec les caractéristiques des différents outils d'édition et les limites de perception d'un utilisateur.

L'avantage d'un tel constructeur est de permettre la définition de différentes représentations externes pour un FA. Chaque utilisateur a sa propre "vue" d'un formulaire abstrait. On doit donc pouvoir définir différents formats pour un même FA. Un format reflète le schéma du FA ou une partie seulement. Parfois entre deux catégories d'utilisateurs, la présentation varie très peu (question de visibilité des champs). Dans ce cas, il n'est pas nécessaire de décrire deux formats, il faut offrir la

possibilité de préciser la visibilité d'un élément de FA pour une catégorie d'utilisateurs.

VII.1.1 Eléments de définition

La définition d'un format se fait avec la commande suivante :

DEF_FT <schéma> FOR <FA> WITH { <usager> }

Un format possède un schéma qui est une description de boîtes. L'agencement de ces boîtes reflète le schéma du formulaire abstrait pour lequel il est défini. Dans la clause FOR, on donne le nom de ce FA ou une FA_expression. Finalement, on indique dans la clause WITH la liste des utilisateurs pouvant utiliser ce format. On peut utiliser le mot clé "all" qui signifie que le format est défini pour tous les utilisateurs.

1- Schéma

Dans un schéma, on retrouve les trois types d'informations décrites pour les "templates" de OFS [TSI 85] : (1) l'aspect image des boîtes, (2) les associations boîtes-champs, (3) les informations sur le positionnement des boîtes. L'approche choisie ici est proche de celle proposée dans le système JANUS [CHAM 82] mais nous n'utilisons pas de langage de programmation pour spécifier un format. Nous n'avons pas non plus réfléchi au problème de l'association dynamique FA - format.

Un schéma est un ensemble de boîtes. Chaque boîte est décrite par un ensemble d'attributs. La table de la Figure 7.1 donne la liste des attributs proposés.

Une boîte est définie par :

- son nom
- son titre : information textuelle ou graphique affichée pour donner une idée du contenu de la boîte.
- la position du titre : indique où le titre doit être placé par rapport au contenu de la boîte (au dessus, à gauche - option standard -, ...).
- son type :

1 simple. Une boîte de type simple permet de présenter une donnée. Les contrôles associés à cette boîte autorisent un déplacement horizontal et vertical ainsi qu'un "zoom" de son contenu.

2 multiple. Une boîte multiple permet de présenter un ensemble ou une liste de données. Les contrôles associés à une telle boîte autorisent un parcours de l'ensemble ou de la liste (premier, suivant, précédent, dernier, "scroll").

- son apparence : on indique quel "fond", on veut donner à la boîte (blanc, noir, grisé, ...). Pour une boîte devant contenir du texte, il faut préciser, la fonte à utiliser, l'espacement des lignes et la taille des caractères.

- sa dimension : hauteur, largeur indépendante ou proportionnelle à la boîte englobante.

- sa position : par rapport aux autres boîtes (au-dessus, au-dessous, dans, ...),

- son contenu. On donne ici une valeur constante (chaîne de caractères, texte, logo, graphique) ou un élément de formulaire abstrait (dont la valeur sera affiché) ou bien une fonction système (date courante). Si le contenu est une liste d'éléments, on peut préciser dans quel ordre les éléments seront affichés (ASC ascendant ou DES descendant par rapport à un composant de l'élément).

- justification du contenu (à droite par défaut)

- son contour. Par cet attribut, on indique comment doit apparaître le contour de la boîte (trait plein pointillé, avec couleur, de telle épaisseur ou bien rien).

- sa description. On donne ici un texte explicatif qui indiquera à l'utilisateur(sur sa demande) le type de l'élément, comment l'utiliser, comment sa valeur a été définie si celle-ci existe (utilisateur, FA_expression, règle), ...

- affichable. On indique ici les conditions d'affichage de la boîte. La valeur de cet attribut est l'expression d'une condition devant être vérifiée pour que la boîte s'affiche. La syntaxe utilisée pour exprimer cette condition est celle de la partie "condition" de la fonction Si. On peut écrire par exemple :

ET (= (month ('now'), 12), = (?USER, 'crisco'))

La boîte décrite avec une telle valeur pour son attribut "affichable", sera affichée si le mois en cours est le mois de décembre et l'utilisateur ayant demandé l'affichage est 'crisco'. Si l'expression de la condition est le mot clé **always** (pas de condition), l'élément est toujours affiché.

- extensible. La présence de cet attribut indique que la boîte est extensible. Sa dimension s'adapte à son contenu dans une certaine limite liée à l'outil de restitution.

ATTRIBUT BOITE	POUR
nom	nom de boîte
titre	titre de boîte
position titre	au-dessus, à côté
type	simple, multiple
apparence	fond de la boîte
dimension	hauteur, largeur
position	au-dessus, au-dessous, dans une boîte
contenu et justification	désignation élément/constante
contour	aspect du contour de la boîte : plein, pontillé
description	texte pour l'aide
affichable	toujours (always) / condition
extensible	oui / non

Figure 7.1 Liste des attributs d'une boîte

Les boîtes pouvant contenir des listes de données, des ensembles de données, des données multimédia pourront être représentées sur l'écran comme des boîtes "popup" (de dimension standard). Le fait de cliquer sur une des ces boîtes permettra sa matérialisation sur écran dans un format adapté à la manipulation de son contenu.

2- Occurrence de format

Dans la réalité, on rencontre des formulaires qui véhiculent des informations qui ne correspondent pas à des informations fixes ou explicatives et qui ne se trouvent pas dans les zones (champs) réservées. Ces informations représentent des annotations ou bien apparaissent lorsqu'on a un cas non prevu. Elles peuvent être vues comme un complément de description de boîtes. Dans d'autres cas de figures, l'utilisateur veut "personnaliser" la présentation d'un occurrence (par exemple "griser" une valeur). Il faut donc lui donner la possibilité de modifier la définition du schéma de format et de créer ainsi une occurrence particulière de ce format liée à l'occurrence de FA. Cela pourra se faire en donnant de nouvelles valeurs à certains attributs (apparence, justification, contour, affichable, description). Il faut proposer des opérations de mise à jour de format. Par exemple **affecter_boîte** (<tra-

jet>.apparence, 'grisé') qui donne à l'attribut "apparence" de la boîte associé à l'élément désigné par <trajet> la valeur *grisé*.

La définition de ces opérations et leurs utilisations possibles au cours d'une transaction GIF reste un problème ouvert. Pour l'instant dans GIF, une occurrence de format est identique à un schéma de format.

VII.1.2 Format standard

Décrire un format est une opération longue et fastidieuse. En disposant d'un outil graphique d'édition de format la chose serait plus aisée mais néanmoins nécessaire. Nous proposons un mécanisme pour la construction d'un format standard pour un FA. Ce mécanisme utilise le schéma du FA.

1- Principe de construction

Un format standard sera défini en donnant :

- son nom,
- le nom du formulaire abstrait,
- les utilisateurs autorisés
- un type de boîte :
  * simple (S) pour la présentation d'une occurrence à la fois sur l'écran.
  * multiple (M) pour la présentation de plusieurs occurrences sur l'écran.
  * menu (MN) pour la présentation d'un menu (liste d'opérations).

  Avec ce type, il est possible de déterminer les fonctions de contrôle à associer au format.

Le mécanisme de construction de format standard doit déterminer les spécifications d'un ensemble de boîtes en utilisant le schéma du FA précisé. Il effectue un parcours du schéma de manière à associer une boîte à chaque élément (noeud) du schéma. Il débute avec la racine du schéma qui est le premier élément auquel doit être associé une boîte. Pour cette boîte, on connaît déjà son type. Il faut compléter la description de la boîte. Cette description, comme pour tout autre élément du schéma correspond à donner un ensemble d'attributs à la boîte.

Pour chaque élément de schéma, est définie une boîte possédant :

- un *nom* interne,
- un *titre* qui est le nom du schéma de l'élément. Sa position par défaut est "à gauche" de la boîte. Elle variera en fonction du format de la boîte englobante. A chaque boîte correspond une boîte dite *logique* qui est une extension de la boîte englobant son titre.
- une *justification* à droite du contenu
- une *apparence* standard
- une *description* qui est la définition de l'élément (son schéma). Cette information est extraite du catalogue stockant les SCHEMA_FA.
- un attribut *affichable* (always) sans condition

Les autres attributs sont définis en utilisant les règles de correspondance schéma d'élément - boîte ci-dessous.

Correspondance schéma - boîte

Pour un schéma **simple** (A:D, A:□), on détermine les attributs :

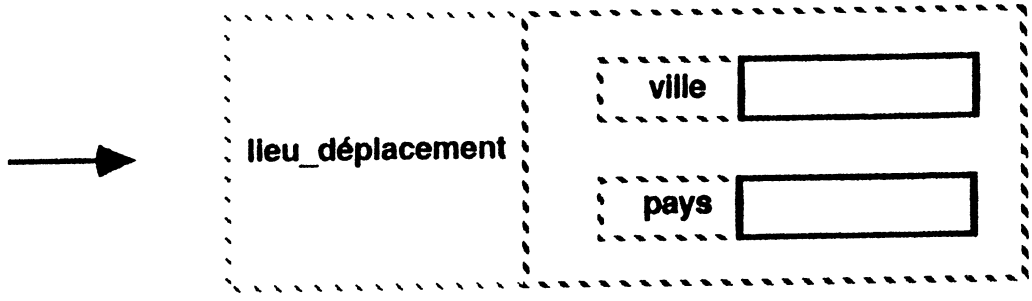
- position titre : au desssus de la boîte si la boîte englobante a le format vertical.
- dimension qui est calculée en fonction du domaine spécifié. S'il s'agit d'un domaine multimédia, la boîte est une boîte "popup".
- contour de la boîte : trait plein.
- le type de fonte (romain), la taille de la fonte (10) et l'espacement des lignes (1,5) dans la boîte.

élément ≡ nom_prénom : string(30);

Pour un schéma **agrégat**, on détermine les attributs suivants :

- position titre : au desssus de la boîte si la boîte englobante a le format vertical.
- dimension : calculée en fonction de la taille respective des N boîtes composantes.
- contour : rien
- format :
  * horizontal si la boîte engobante a le format horizontal ou s'il s'agit de la boîte de plus haut niveau (correspondant à la racine du schéma de FA). Les boîtes composantes seront arrangées dans K zones horizontales de la boîte agrégat. La détermination de K est fonction du nombre N de composants et de la taille de la boîte associée à chacun de ces composants. La hauteur d'une zone est fonction de la hauteur des boîtes qu'elle contient.
  * vertical si la boîte englobante a le format vertical. Les boîtes composantes sont arrangées en N zones verticales.

schéma ≡ lieu_déplacement : begin
ville : string (20);
pays : string (10);
end;



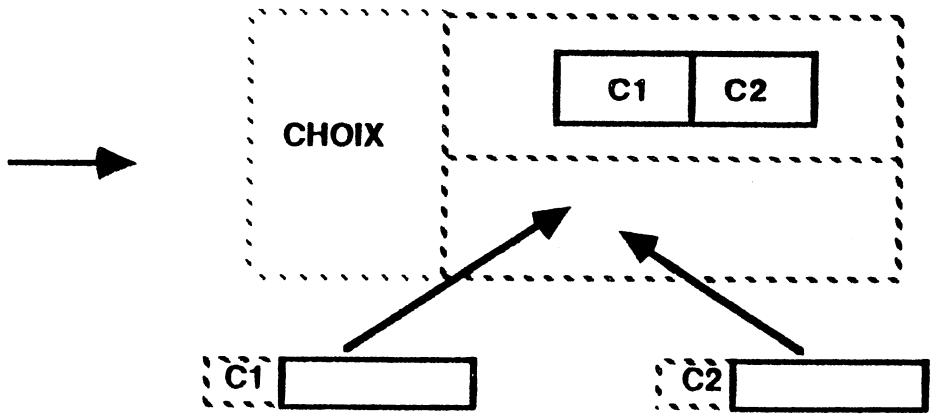
Pour un schéma **choix**, on donne les attributs :

- type : simple
- dimension calculée en fonction de celle de la plus grande des boîtes associées aux schémas du choix et de la boîte menu pour effectuer le choix.
- contour : rien

- format : horizontal. On découpe la boîte en deux zones horizontales. La première zone contient une boîte de type menu avec pour contenu : les noms des schémas composant le choix. La seconde zone est réservée pour l'affichage de la boîte correspondant au schéma qui sera choisi. La taille de cette zone est égale à celle de la plus grande des boîtes associées aux schémas décrivant les choix. Lorsqu'un de ces schémas est une constante (domaine □), on ne définit pas de boîte associée.

  Chaque boîte associée à un schéma (Si) composant le schéma choix (S) aura un attribut affichable défini par la condition : = *(nom(S), nom(Si))*

schéma ≡ CHOIX case of C1: string (10);
C2 : integer;
end;



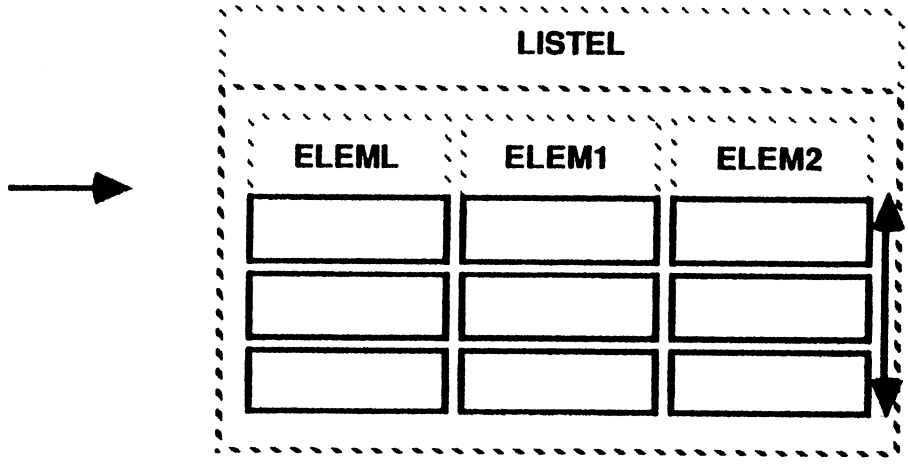
Pour un schéma **liste** (d'ordre n), la boîte correspondante est définie avec les attribut supplémentaires suivants :

- position titre : au dessus de la boîte L (Liste).
- type : multiple
- dimension : La dimension de la boîte est calculée de manière à pouvoir contenir au moins K boîtes, chacune correspondant au schéma composant le schéma liste (K dépend de la taille de la fenêtre utilisée pour matérialiser le formulaire). Pour la boîte englobante, la boîte liste pourra être une boîte "popup".

- contour : trait plein
- ordre : ascendant sur le premier composant de l'élément de la liste.
- extensible : la boîte doit pouvoir s'étendre selon le nombre de boîtes (valeurs) de la liste ("scroll" associé)
- format : vertical. La boîte est divisée en K zones horizontales. Chaque zone contient la boîte (de format vertical) associée au schéma S décrivant un élément de la liste. Cette boîte ne possédera pas de titre. Par contre elle sera étendue par une boîte verticale supplémentaire (V), de titre nom(S), dont le contenu sera la valeur de l'indice de l'élément dans la liste.
  Si le schéma liste est de la forme :
  * list (min, max) of (A :D) alors on associe à la boîte V (de titre A), une boîte de de titre "VALEUR", pouvant contenir une valeur du domaine D.

  Pour ne pas alourdir la présentation d'une boîte liste, on donnera une seule fois les titres des boîtes associées à ses éléments.

schéma ≡ LISTEL : list (1, *) of ELEML : begin
ELEM1 : integer;
ELEM2 : texte;
end;



Pour un schéma **ensemble**, on procède de la même manière que pour un schéma list. La boîte contenant une valeur "indice" permet de désigner un élément de l'ensemble.

La Figure 7.2 est le résultat de l'affichage du FA *demande_mission* avec son format standard.

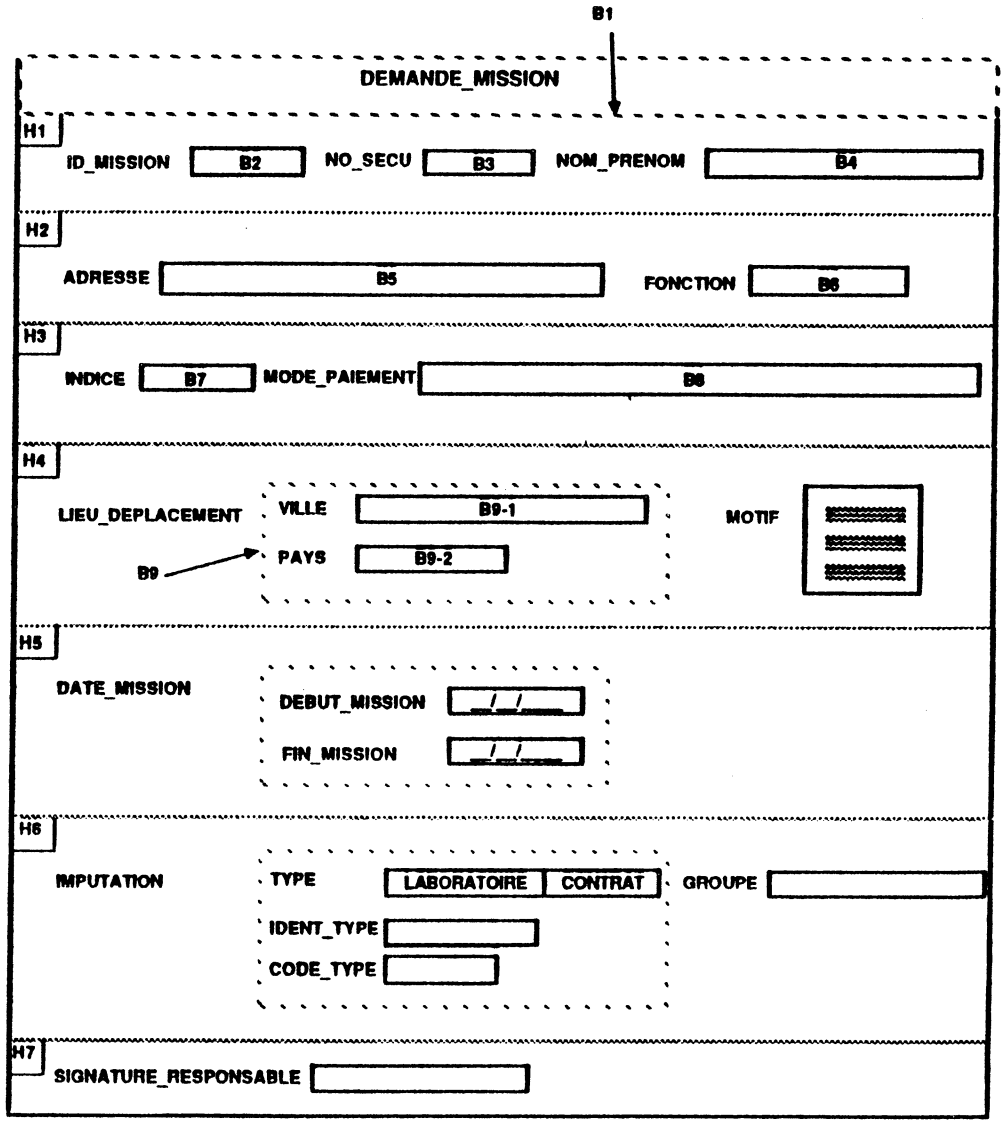


Figure 7.2 : affichage du FA *demande_mission* avec son format standard

En conclusion, un FA possédera au moins un format dit "standard" dont le schéma reflète l'organisation logique du FA. Si on dispose d'un mécanisme de réduction d'images et de gestion d'icônes sur le poste de travail, on pourra construire un format "icône", image réduite du résultat de l'association format standard et FA.

VII.2 Les opérations spécifiques

Les opérations spécifiques constituent la seconde composante d'une présentation pour un FA. Elles symbolisent les actions "réelles" qu'un utilisateur va pouvoir réaliser sur l'image, résultat de l'association FA - format.

Dans le chapitre V, nous avons montré comment il est possible d'attacher de la connaissance au FA en définissant des règles. Cette connaissance détermine le comportement du FA vis à vis d'événements possibles qui correspondent à des FA_opérations de mise à jour.

Lorsqu'on manipule un formulaire, toute action externe (clic de la souris sur un champ, remplissage d'un champ, etc) est traduite en opération(s) sur le FA. Le format détermine un environnement pour l'utilisation du FA sous-jacent : outils d'édition pour la manipulation des données "visisibles" dans le format, utilisateurs autorisés à le manipuler. Certains formats cachent une partie importante des données, et de ce fait la traduction d'actions externes en FA_opérations n'est pas facilitée voir même impossible. Par exemple au travers du format de la Figure 7.3, on ne pourra pas réaliser la création d'une nouvelle mission. Ce format permet de visualiser une liste d'occurrences du FA "demande_mission" (pour une personne) et présente pour chaque occurrence une partie seulement de ses informations.

De plus toute utilisation d'un formulaire est contrôlée. Les actions pouvant être réalisées par chaque intervenant possible sont parfaitement connues. Par exemple, pour le FA *demande_mission* avec le format de la Figure 7.2, l'utilisateur ***balou*** n'est pas autorisé à manipuler les éléments "imputation", "groupe" et "signature_responsable".

On voit donc qu'au niveau externe, il faut ajouter à un format de la connaissance qui code le comportement du FA sous-jacent vis à vis de ce format. Cette connaissance s'exprimera par les opérations **spécifiques** associées au format. Elles constituent le seul moyen d'action des utilisateurs sur le formulaire.

Une opération spécifique "remplir" pour le format de la Figure 7.2 et l'utilisateur *balou* précise les opérations qui seront reconnues par le FA *demande_mission* lorsque cet utilisateur va réaliser des actions. En autorisant une description explicite des événements qui peuvent être réalisés sur un FA (au travers d'un certain format), une opération spécifique apparaît comme un **superviseur** de règles. Lorsqu'un utilisateur activera une opération, GIF disposera alors d'un ensemble de renseignements sur ce qui peut être fait pour le formulaire. On a ainsi

un premier niveau de contrôle des actions utilisateurs sans avoir à exploiter des règles.

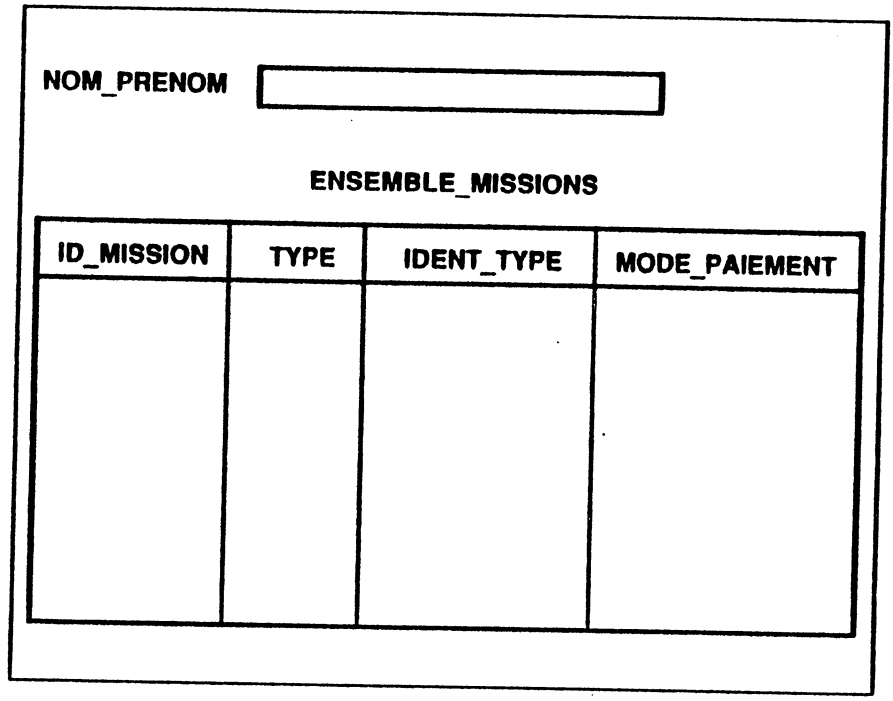


Figure 7.3 : Format Liste_demande_mission

VII.2.1 Spécification informelle

Une opération spécifique c'est un programme utilisant les services du poste de travail : SGBD, gestionnnaire du dialogue Homme-Machine (interface), outils logiciel (messagerie, traitement de texte, graphisme, ...). Si l'on prend l'exemple du FA *demande_de_mission* présenté avec le format affichage de la Figure 7.2, on peut définir :

- une opération spécifique REMPLIR qui reçoit et envoie des ordres à l'interface, demande l'utilisation d'outils d'édition, échange des informations avec le gestionnaire de règles.
- une opération RECHERCHE qui autorise l'utilisateur à exprimer une requête et construit la FA_expression correspondante pour l'extraction des occurrences recherchées.
- une opération LIST qui permet de faire afficher les occurrences du FA (présentes dans l'environnement) avec le format de la Figure 7.3. Activée après une

recherche, elle permet de visualiser le résultat de cette recherche avec une présentation mieux adaptée à la manipulation d'un ensemble d'occurrences.

Le langage utilisé pour définir une opération spécifique doit intégrer des fonctions de programmation, des opérations sur des formats, des FA_expressions, des commandes particulières de l'interface, et du poste de travail (éditer, envoyer, ...). Il est clair que la programmation d'une opération spécifique sera facilitée si l'on dispose sur le poste d'un "intégrateur" pour les outils logiciels (messagerie, tableur, graphiques, traitement de texte, ...).

Une opération sera définie par :

DEF_OP <nom_opération> FOR <nom_format> AND { <utilisateur> }
begin
parameters: { <paramètre> };
objects : { <objet_local> };
body : <corps_opération>;
end;

Dans la partie "parameters", on décrit les données devant être envoyées à l'opération au moment de son activation (identifiant de l'utilisateur, du format, du FA, ...).

Dans la partie "objects", on trouve la description des objets locaux, c'est à dire les structures de données propres à l'opération, "images" du FA manipulé.

Le corps de l'opération est un ensemble d'actions à faire liées par des structures de contrôles. Celles_ci peuvent être déterminées par la description du schéma de FA concerné et sa "vue" à travers le format. Avec l'approche que nous avons choisi (dans un premier temps) de construire un format standard pour un FA, reflétant sa structure, nous pourrons également générer le cadre de certaines opérations spécifiques de base telles que : remplir, modifier, supprimer, rechercher.

Ce cadre n'est qu'une spécification informelle de la définition de l'opération. Nous le nommons **schéma de comportement**. Il est une aide pour la construction du programme correspondant à l'opération. La traduction de cette spécification en un programme reste un problème ouvert.

VII.2.2 Schéma de comportement

La connaissance attachée à un format est complètement définie par ses différents schémas de comportement. Un format standard possède au moins un schéma retrouver, un schéma remplir, un schéma modifier et un schéma supprimer. Un schéma de comportement est généré pour un format (indépendamment des utilisateurs). Il est défini en utilisant les informations données par le SCHEMA_FA (schéma et règles) du FA sous-jacent au format considéré.

Pour la construction d'un schéma de comportement, nous disposons de correspondances entre constructeurs de FA et structures de contrôle. Ces structures sont utilisées pour lier les événements autorisés sur les éléments de FA.

Au constructeur agrégat correspond la séquence (**begin .. end**). Une opération sur un élément défini avec ce constructeur est composée d'une séquence d'opérations, une pour chaque composant de l'élément. Par exemple une opération remplir sur le FA "demande_mission" sera composé d'une séquence d'opérations pour donner une valeur aux éléments "id_mission", "nom_prénom", "adresse", ...

Au constructeur list correspond une répétition (**repeat**). Une opération remplir sur l'élément supplément du FA "remboursement_frais" est composée d'une opération qui s'applique à chaque composant de la liste supplément.

Finalement, pour le constructeur choix nous avons la structure de contrôle choix (**if then else**). Un remplissage pour l'élément "réduction" du FA "remboursement_frais" correspond à choisir une valeur parmi les valeurs ('oui', 'non') du sélecteur ("réponse") puis à donner une valeur à l'élément correspondant au choix effectué.

Pour le format standard "remboursement" (annexe I) du FA "remboursement_frais", on peut générer le schéma de comportement REMPLIR de la Figure 7.4.

Cet exemple montre bien que le schéma de comportement indique quels éléments du formulaire doivent être remplis. Il constitue le cadre du dialogue entre le formulaire et l'usager. On peut remarquer qu'il n'y a pas d'actions "affecter" attendues pour les éléments calculés, les éléments déclarés avec une valeur par défaut ou pouvant avoir une valeur nulle inapplicable.

SCHEMA_COMP REMPLIR **FOR** remboursement

```
begin
  affecter (ident_form);
  entete : begin
           affecter (exercice);
           affecter (no_uer);
           affecter (mandat);
           end
  voyage : begin
           réduction : begin
                       if réponse = 'oui'
                       then affecter (pourcentage_reduc);
                       if réponse = 'non' then;
                       end
           suppléments : repeat (0,5)
                       affecter (ident_sup);
                       affecter (prix_sup);
                       end
           affecter (départ_grenoble);
           affecter (retour_grenoble);
           affecter (arrivée_étranger);
           affecter (départ_étranger);
           affecter (signature_responsable);
           affecter (signature_intéressé);
           affecter (signature_ordonnateur);
  end
```

Figure 7.4 Schéma de comportement REMPLIR
pour le format "remboursement"

VII.3 Opérations pour une présentation

Comme nous l'avons vu, une présentation est définie par deux composantes indissociables. Lorsqu'un usager va demander un formulaire, il le fera par le nom sous lequel il connaît ce formulaire, c'est à dire le nom d'un format. Le système construit alors le formulaire "réel" : associe le format avec le FA correspondant et les opérations spécifiques pour l'utilisateur. Ensuite, l'utilisateur peut activer une opération.

1- Les opérations pour le format

On propose quatre opérations pour les formats : **FT, associer, présenter, effacer**. Elles peuvent être utilisées au même titre que les FA_expressions dans n'importe quelle transaction. Dans une opération spécifique (d'un formulaire), on pourra alors décrire les actions pour présenter un nouveau formulaire.

FT

L'opération **FT** (<variable>, <nom_format>) est utilisée pour définir une FT_variable désignant un format. On pourra utiliser cette variable dans les opérations sur un format ou pour donner une valeur à un élément de FA défini sur le domaine graphique ou image. De plus, on peut associer à cette variable des occurrences d'un FA.

Exemple : **FT** (D, demande);

associer

L'opération **associer** (FT, F) associe les occurrences d'un FA identifié par F (nom de FA, ou nom d'un FA temporaire) avec le format FT (nom de format, ou variable désignant un format).

Exemple : FA (F, sélection (demande_mission, = (id_mission, 87/07/M10)))
associer (D, F);

présenter

L'opération **présenter** (FT, (x,y), demandeur) matérialise le format identifié par FT, contenant ou non des occurrences. x et y donne les coordonnées pour l'affichage sur l'écran. On suppose que l'origine est le coin en haut à gauche de l'écran. L'axe des x se trouve être parallèle au bord haut de l'écran et se dirige vers la droite. L'axe des y descend sur le côté gauche de l'écran. <demandeur> désigne un utilisateur U du système. L'opération vérifie que le format FT demandé est autorisé à l'utilisateur U et associe à FT les opérations spécifiques réservées à U.

Exemple : **présenter** (F, (0,0), 'balou');

effacer

Pour effacer un format de l'écran, on utilise **effacer** (FT).

2- Les opérations pour les Opérations Spécifiques

Pour les opérations spécifiques, On propose deux opérations :

(1) **activer** (OP), pour activer une opération de nom OP

(2) **annuler** (OP), pour annuler une opération de nom OP

3- Dynamicité des présentations

Dans la section VII.1, nous avons montré comment on peut envisager un mécanisme de construction de format standard. Proposer un tel mécanisme réduit considérablement la tâche du concepteur. Néanmoins, le format standard peut très bien ne pas lui convenir et on doit pouvoir lui fournir les outils pour le modifier. On peut voir cet outil comme un éditeur de formulaire graphique On peut envisager d'avoir un outil qui permette d'avoir une image graphique d'un format standard, de manipuler (créer, déplacer, réduire, étendre, couper, coller, ...) des boîtes et modifier certains des attributs des boîtes. Ces manipulations seront acceptées si elles correspondent à un schéma appartenant au FA associé au format.

Pour les opérations spécifiques, on pourra les détruire, les modifier de la même manière que les fonctions (chapitre VI).

VII.4. Quelques formulaires

Dans cette section, nous présentons certains formulaires de l'outil de description de GIF. Cet outil est une application formulaire particulière. Nous voulons montrer que le modèle de présentation envisagée pour GIF (formulaire) répond aux critères demandés aux interfaces postes de travail : naturel , simple, pédagogique, cohérent, souple.

Le processus de définition dans GIF débute par la description d'un schéma de FA que l'on enrichit au fur et à mesure des étapes de définition (règles, format, opérations).

La Figure 7.5 donne l'apparence d'un écran GIF, contenant une fenêtre pour formulaire. Cette fenêtre "formulaire" est découpée en trois zones : 1) une zone affichage (A) destiné à l'affichage de l'association FA-format 2) une zone

(D) qui sert à l'affichage des messages et au dialogue avec l'utilisateur 3) une entête (E) pour l'affichage des noms d'opérations spécifiques.

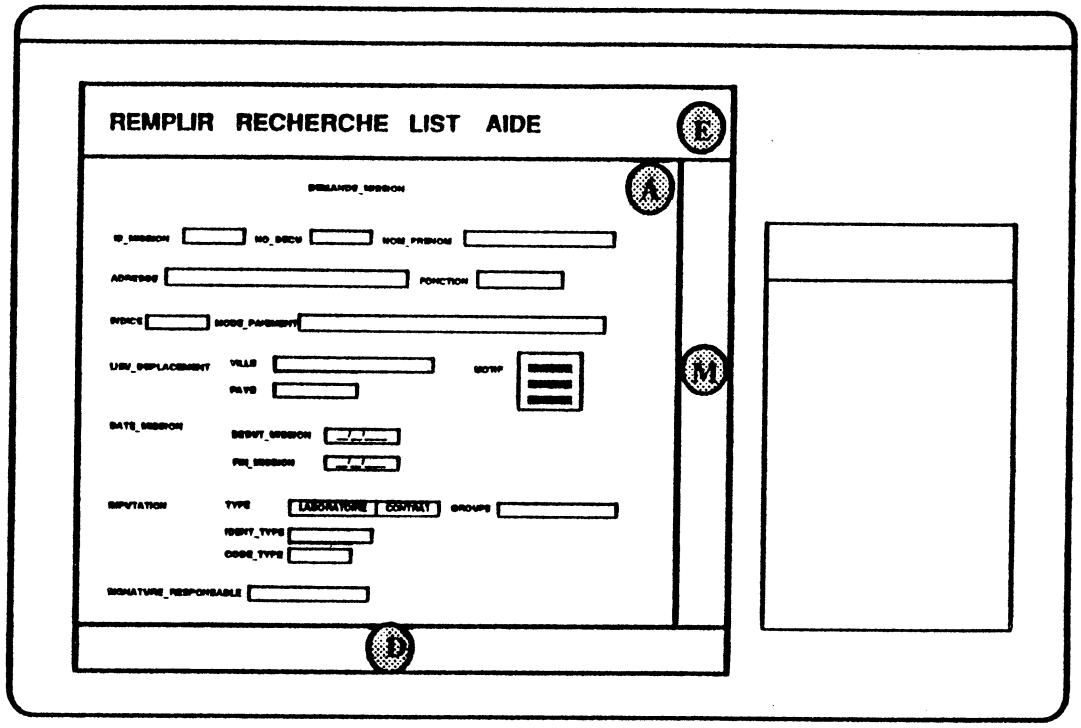


Figure 7.5 : écran GIF

Dans un certain environnement (par exemple REMPLIR), l'usager peut alors utiliser le formulaire : cliquer sur un élément. GIF fournit alors un ensemble de commandes d'édition en fonction du type de l'élément (texte, graphique), présentées sous forme d'un menu dans la zone M.

Le formulaire EDIT_FA

Le formulaire EDIT_FA est utilisé pour définir un schéma de FA et ses règles. Pour définir un nouveau schéma, on active DEFINIR. On donne un nom de FA, puis on définit son domaine ou les différents schémas le composant. On commence par définir les schémas simples (A:D, A:□), puis on les composent. On peut alors définir d'autres schémas plus complexes. La définition se termine lorsqu'on décrit un schéma portant le même nom que le FA. Toute la définition des schémas se fait en utilisant les commandes du menu (M) : nouveau, visuliser, supprimer, etc. On donne un nom de schéma et on choisit son constructeur :

Agrégat, Choix, Liste, Vue, Base (élément simple). Selon le choix effectué, une boîte s'affiche pour permettre de continuer la définition.

Dans la Figure 7.6, on montre comment a été défini l'élément "nom_prénom" pour le FA "demande_mission". L'usager a choisi B, une boîte s'affiche pour lui permettre de donner le domaine de l'élément (string(20)).

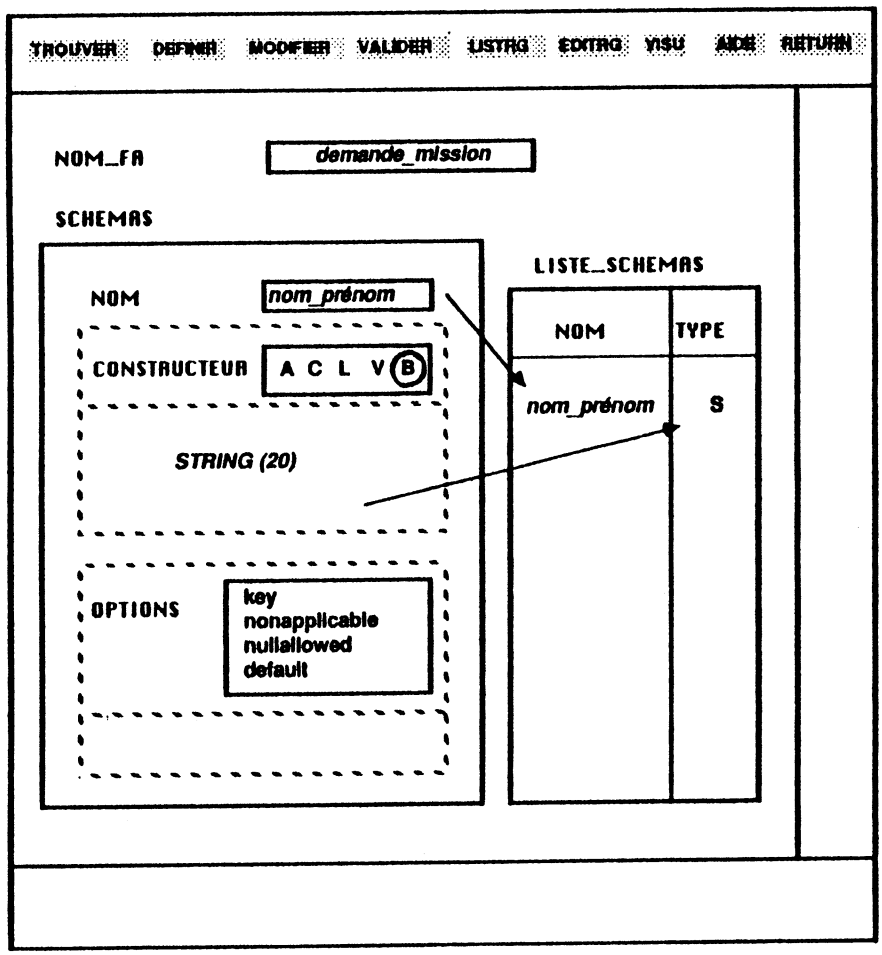


Figure 7.6 : Définition du FA *demande_mission* - élément *nom_prénom*

On garde une trace de chaque nouvelle définition. L'usager dispose dans la boîte LISTE_SCHEMAS des noms des schémas qu'il a décrit et de leur type (Simple ou Complexe). Dans la Figure 7.7, on montre comment est défini l'élément complexe *date_mission*. L'usager a choisi le constructeur Agrégat. Dans la zone message apparaît : "veuillez sélectionner le(s) schéma(s) composant *date_mission*". L'usager sélectionne les composants dans LISTE_SCHEMAS.

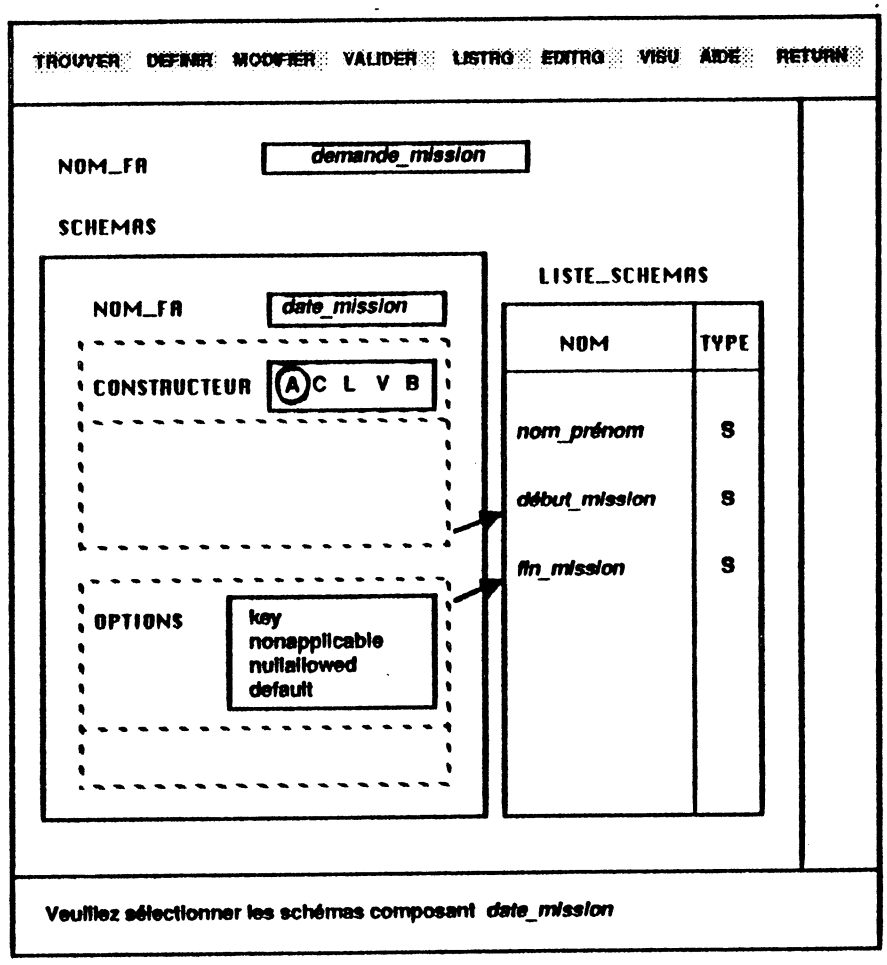


Figure 7.7 : Définition du FA *demande_mission* - élément *date_mission*

Le formulaire EDIT_FA fournit également les opérations : VALIDER (validation d'une opération), AIDE (aide à l'utilisation du formulaire). VISU : visualisation de la définition en cours sous forme d'un arbre ou de plusieurs arbres. RETURN permet de retourner au formulaire appelant. LISTRG pour lister les règles associées à un schéma de FA, EDITRG pour l'édition - définition, modification, ..- d'une règle avec le formulaire EDIT_RG (Figure 7.8). Dans ce formulaire, l'opération CONTROL permet de vérifier si la définition de règle affichée n'est pas incohérente (on peut rêver ...).

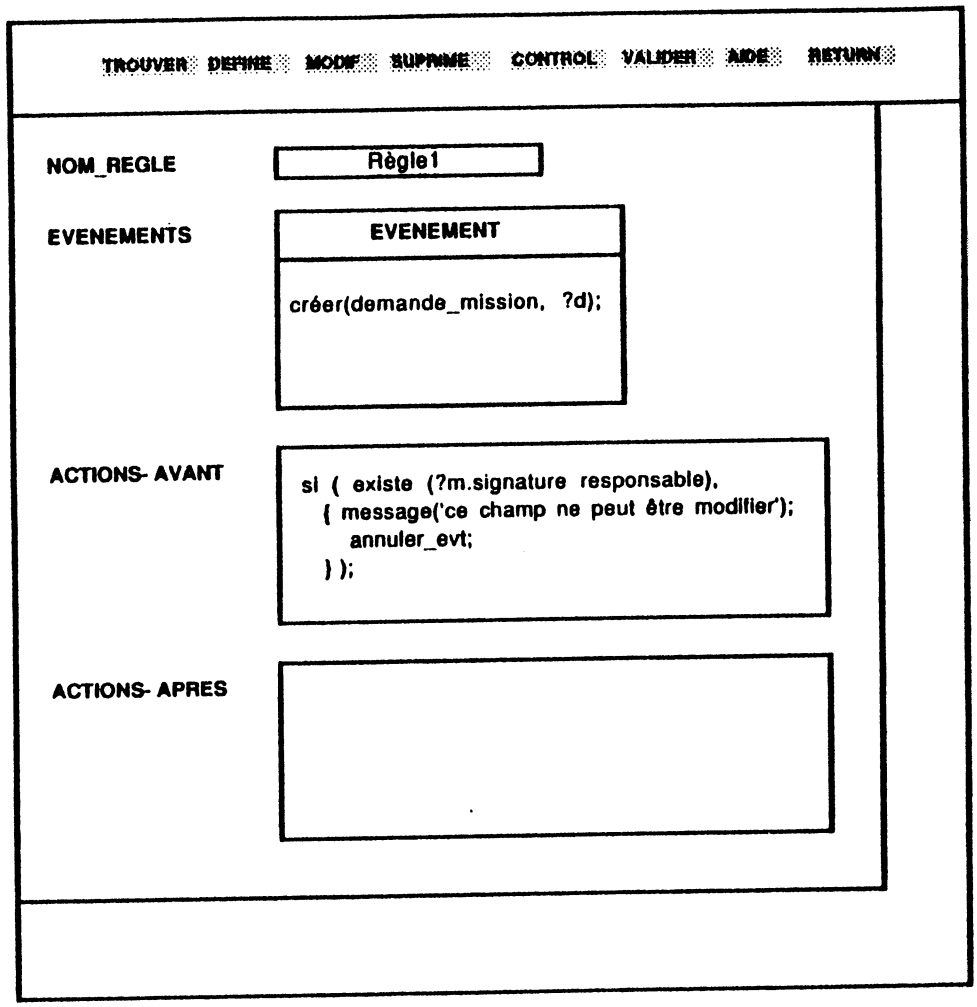


Figure 7.8 : Formulaire EDIT_RG

CONCLUSIONS

Peut-être y a-t-il d'autres connaissances à acquérir, d'autres interrogations à poser aujourd'hui, en partant, non de ce d'autres ont su, mais de ce qu'ils ont ignoré.

S. Moscovici

CONCLUSIONS

Résumé et contribution

Dans cette thèse, nous avons montré comment la notion de formulaire complexe avait un rôle important à jouer pour la description des divers aspects d'une application des bases de données multimédia (données, sémantique et interface).

Notre démarche a été dans un premier temps d'étudier les propositions existantes en matière de gestion de formulaires (Chapitre I). Cette étude nous a permis de constater que les travaux de recherche et de développement avaient mis l'accent (1) sur l'aspect "interactif" du formulaire : il est utilisé comme interface entre l'Homme et la machine, (2) sur les aspects "facile à appréhender" et "ergonomique" du formulaire : il est largement utilisé pour la spécification de modèles de bureau, pour la conception d'applications bureautiques, et plus généralement pour la conception des applications bases de données. Dans la première approche, on s'appuie sur l'idée que le formulaire est un objet que tout le monde manipule parce qu'il possède une sémantique implicite. La seconde voie est liée au fait que le formulaire est depuis toujours considéré par les organisateurs comme un modèle d'échange de l'information et un moyen d'améliorer la circulation de celle-ci. En considérant les aspects "bases de données" des systèmes proposés, nous constatons que le formulaire n'est pas un objet de la base : sa structure, sa valeur et sa présentation ne sont pas décrites et stockées explicitement dans la base. De plus peu de travaux s'intéressent au problème de la description et de la gestion de la dynamique d'un formulaire : toute sa sémantique est gérée dans les programmes d'application.

Cette thèse a débuté dans le cadre du projet TIGRE [VEL 84] sur les bases de données généralisées. Nous avons alors cherché à "généraliser" la notion de formulaire et nous avons choisi d'avoir une vision plus globale qui intègre divers aspects : le formulaire n'est plus seulement un objet d'interface mais également un objet des applications. Il est stocké comme un objet de la base, il est complexe parce qu'il est structuré, il contient des données alphanumériques classiques et des

données multimédia, il possède de la connaissance pour adapter son comportement aux opérations qui lui sont demandées.

Notre étude s'est alors poursuivie en dégageant les besoins en description et en manipulation des formulaires complexes (chapitre II). Nous avons très vite constaté que la spécification d'un outil interactif pour la gestion de formulaires complexes posait des problèmes similaires à ceux rencontrés à l'heure actuelle pour la mise en oeuvre des SGBD multimédia : modélisation et gestion des données, spécification et gestion de la sémantique, environnement de travail, etc.

Nous avons essayé de conserver les spécificités du formulaire en proposant en particulier un modèle "orienté objet" : le formulaire est défini par : (1) un FA qui décrit ses propriétés structurelles (schéma) et sémantiques (règles) et (2) une ou plusieurs Présentations. Le chapitre III présente la commande de définition d'un FA sans aborder ses aspects sémantiques. Nous avons choisi d'étudier des schémas hiérarchiques obtenus en utilisant les constructeurs : ensemble, liste, agrégat (n-uplet), choix et vue. On retrouve le concepts d'aggrégation et d'union des modèles sémantiques. Le constructeur vue offre un niveau d'intégration avec d'autres modèles : une relation pourra être vue comme un formulaire. Il permet également d'exprimer des associations entre deux FA et offre les opérateurs de spécialisation et de généralisation. Il autorise également la description de schéma cyclique.

Les opérations de manipulation et de mise à jour des FA ont été présentées au chapitre IV. L'algèbre proposée s'intègre dans le courant des algèbres pour objets structurés ou complexes et peut être vue comme une extension aux algèbres pour données relationnelles non en première forme normale. L'intégration de fonctions avec les opérations augmente la puissance du langage et permet une meilleure description de la sémantique des données.

Cette sémantique est modélisée par de la connaissance associée au schéma. Le chapitre V présente le concept de règles utilisé pour décrire la connaissance. Les nombreux exemples introduits montrent que l'on peut utiliser la notion de règle pour exprimer la sémantique d'une application et les contraintes intra ou inter objets de la base au sens large. Ici encore nous avons voulu conserver les spécificités du formulaire. Nous n'avons pas cherché à résoudre tous les problèmes liés à la dynamique des données qui dépassent largement le cadre de cette thèse. Nous avons choisi de nous appuyer sur l'aspect "interactif" du formulaire pour décrire et gérer son comportement en fonction des actions de l'utilisateur. Nous avons pu constater

que même en se donnant un cadre précis de nombreux problèmes restent posés : gestion des événements, cohérence des règles, etc ...

Le chapitre VI présente d'autres commandes autorisant une évolution de la signification d'un FA et donnant à GIF son caractère dynamique. Le chapitre VII présente les aspects d'interface avec l'utilisateur et plus particulièrement le concept de Présentation. Il permet la description d'un format externe pour un FA et les opérations spécifiques associées (moyens d'action des utilisateurs sur le FA au travers du format). Les exemples de formulaires donnés dans ce chapitre montre que les propositions faites ici peuvent être utilisées pour implanter GIF lui-même.

Recherches futures

Il est souhaitable de prolonger ce travail dans les directions suivantes :

1- Expérimentation

Notre souci de formaliser proprement l'ensemble des concepts proposés ne nous a pas permis de mener à bien une expérimentation concluante. Nous souhaitons que ce travail soit poursuivi dans le sens d'une implantation :

- du noyau de GIF. Il s'agit de mettre en oeuvre la commande DEF_FA et l'algèbre proposée. Nous avons choisi de réaliser ce noyau sur un SGBD relationnel et la correspondance entre le modèle de formulaires et le modèle relationnel a été établie [COL 86]. L'implantation des opérations fait appel à des techniques de programmation avancées et nous pensons d'ores et déjà simplifier les propositions faites ici.

- du gestionnaire de formulaires. Il s'agit de mettre en oeuvre l'ensemble des mécanismes d'interface pour GIF en jouant sur le méta-niveau du modèle : par exemple proposer un ensemble de méta-formulaires pour la définition et la manipulation des formulaires d'une application. Une expérience a été menée dans ce sens [CAB 85]. L'outil interactif développé permettait à un utilisateur de définir un formulaire, de l'activer et de le manipuler d'une manière conforme à sa sémantique. Cet outil utilisait un modèle de formulaires avec les concepts de schémas et de Présentations. La sémantique était décrite dans les opérations spécifiques de la Présentation. Cette expérience a été positive dans le sens où elle a permis de faire évoluer le modèle vers ce qu'il est aujourd'hui et parce qu'elle a montré que l'outil GIF pouvait être développé de manière incrémentale.

2- Les transactions

Comme le lecteur a pu le constater, nous avons choisi pour GIF un concept de transaction voisin de celui qui est proposé pour les SGBD relationnels. Ce concept n'est plus suffisant si l'on veut aujourd'hui manipuler des données complexes, pour lesquels les transactions sont longues. De plus un formulaire peut contenir des données provenant de différents objets et dans ce cas une transaction risque de bloquer trop longtemps les autres transactions demandant le formulaire ou un des objets qu'il utilise. Le problème de la gestion de la concurrence sur un formulaire doit donc être examiné si l'on veut fonctionner en mode multi-utilisateurs. Un étude plus approfondie doit également être faite pour la gestion des transactions imbriquées, nécessaires si l'on veut fonctionner en mode multi-formulaires sur le poste de travail. Pour ce qui est de la reprise après panne, nous avons déjà dit qu'il n'est pas envisageable de perdre tout le travail effectué sur un formulaire depuis le début de son utilisation. On peut alors envisager un mécanisme qui réalise des sauvegardes incrémentales.

3- Modélisation :

Pour notre algèbre de FA, il faudrait en montrer la complétude en étudiant les différentes notions de complétude proposées jusqu'à aujourd'hui [ABI 87]. Il faudrait également montrer que toute FA_expression est saine c'est à dire que son évaluation conduit toujours à un ensemble d'occurrences fini.

Par le constructeur vue (view), nous pouvons avoir des schémas cycliques : la mère d'une personne est une vue sur personne. Par contre, il faut interdire les cycles dans les données : par exemple, on ne peut pas avoir Zoé, mère de Fanny qui est la mère de Zoé. Il faudrait regarder s'il n'est pas possible de détecter ces cycles au plus tôt, dès que l'on propose la donnée comme valeur d'un élément d'occurrence. On peut songer à associer à tout schéma de FA une règle particulière, permettant de détecter ces cycles.

Nous pensons qu'il faut également poursuivre notre effort de formalisation pour le concept de règle. Les aspects interface pour le formulaire mérite une étude plus complète. Ce problème rejoint celui de la spécification des interfaces pour les nouveaux SGDB.

4- Applications formulaires

Dans un environnement bureautique, les tâches sont agencées d'une certaine manière, traduisant à la fois comment circulent les informations et leurs traitements successifs. Une application de cet environnement peut être modélisée par un ensemble de formulaires. A un moment donné, l'utilisation d'un formulaire, correspond à la réalisation d'une action ou d'une suite d'actions pour une tâche. La notion d'application formulaire permet de modéliser le fait que des formulaires de structures et sémantiques différentes sont liés entre eux. Pour définir une application, il ne suffit pas de décrire ses formulaires, il faut également :

- définir un schéma d'application décrivant comment s'enchaîne ses formulaires.
- exprimer la sémantique attachée à ce schéma : les actions à réaliser avant et/ou après l'activation d'un ou de plusieurs formulaires.
- définir la Présentation de l'application elle-même

Les constructeurs proposés ici permettent la description de la structure d'un formulaire. On peut se demander dans quelle mesure ils peuvent être utilisés pour la définition d'un schéma d'application qui liant les formulaires qui la compose. De même, on peut se se demander si le concept de règle peut être utilisé pour décrire quand peut ou doit être activer un formulaire. Ce problème rejoint le problème plus général de la gestion des procédures de bureau [HOE 87].

Pour conclure, nous considérons que notre travail se trouve au confluent de plusieurs tendances actuelles dans la recherche et le développement en base de données : modèle d'objets structurés/complexes et algèbre associée, représentation des connaissances, dynamique des données, interfaces pour les SGBD. La formalisation des concepts que nous proposons montre que ces tendances doivent et peuvent s'intégrer pour offrir des outils de conception et de mise en oeuvre des nouvelles applications. Cette thèse couvre un large domaine et il est fort probable que certains problèmes nous aient échappés ou aient été mal traités. Nous pensons néanmoins que les propositions faites ici constituent un cadre homogène pour décrire et traiter la structure et la dynamique des objets complexes, vus comme des formulaires.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ABI 84] ABITEBOUL S., BIDOIT N.
Non First Normal Form relations : an algebra allowing data restructuring
Rapport de Recherche. No 347. INRIA, Novembre 1984.
Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database System (1984). Journal of Computer and Sciences (December 1986).

[ABI 86a] ABITEBOUL S.
Manipulations d'objets complexes
Deuxièmes Journées Bases de Données Avancées. Giens, Avril 1986.

[ABI 86b] ABITEBOUL S., HULL R.
IFO : A Formal semantic database model
TR-84-304. Computer Science Departement - University of Southern California. (Los Angeles). April 1984.
ACM Transactions on Data Base System. 1986.

[ABI 86c] ABITEBOUL S., SCHOLL M., GARDARIN G., SIMON E.
Towards DBMSs for supporting new applications
Proc of 12th VLDB conference. KYOTO, august 1986.

[ABI 87] ABITEBOUL S., GRUMBACH S.
Bases de Données et Objets Structurés
Techniques et Sciences de l'Informatique. 1987. A paraître.

[ADI 81] ADIBA M.,
Derived relations : a unified mechanism for views, snapshots and distributed data
Proc 7th VLDB Conf. Cannes, Septembre 1981.

[ADI 83] ADIBA M.
Bases de Données et CAO, Bases de Données et Nouvelles Perspectives
Rapport du Groupe BD3, Janvier 83.

[ADI 87] ADIBA M., BUI QUANG N., COLLET C.
Aspects temporels, historiques et dynamiques dans les Bases de Données
Techniques et Sciences de l'Informatique. 1987. A paraître.

[ANT 81] ANTONELLIS V., ZONTA B.
Modelling events in Database Application design
Proc 7th VLDB Conf. Cannes, Septembre 1981.

[ANT 86] ANTUNES F.
Gestion des types abstraits dans le SGBD TIGRE
Rapport interne. Octobre 1986.

[AUT 86] AUTRAN J.
L'informatisation du devis descriptif de batiment- Spécification de l'interface destiné à l'utilisateur final
Note Interne IA 02/86. GAMSAU, Marseille.

[BAL 84] BALTAS S.
Accès concurrents aux documents
Rapport de recherche TIGRE, No 10. Janvier 1984.

[BAN 81] BANCILHON F., SPYRATOS N.
Updates semantics of relationnal views
ACM TODS. Vol 6, No 4. 1981.

[BAN 82] BANCILHON F., RICHARD P., SCHOLL M.
On line processing of compacted relations
Proc. VLDB Conf. Mexico city 1982.

[BAN 86] BANCILHON F., KIM W., KORTH H.F
A model for CAD transactions
Proc. VLDB Conf. Stockholm, August 1985.

[BAR 84] BARBIC F., CARLI M., PERNICI B., BRACCHI G.
A tool for form definition in office information systems specification
New Applications on Databases edited by G. Gardarin.
Based on Proc ICOD-2 Conf. Workshop Cambridge University, September 1983.

[BEN 85] BENSAID A.
Un modèle de données relationnel étendu
Thèse Docteur Ingénieur. INPG de Grenoble, Juin 1985.

[BER 86] BERZTISS A.
A high level specification of office information systems
SYSLAB Report No 42. Departement of Computer Sciences. Royal Institute of Technology and the University of Stockolm. 1986.

[BORG 85] BORGIDA A.
Language Features for Flexible Handling of Exceptions in Information Systems
ACM TODS volume 10 N°4, December 1985.

[BOR 81] BORRON H.J
The concept of type : considerations for document preparation, retrieval and communication
Projet KAYAK. INRIA, Septembre 1981.

[BRA 83] BRACCHI G., PERNICI B.
SOS : A conceptual model for office information systems
Proc of ACM SIGMOD. 1983.

[BRO 80] BRODIE M.L
The application of Data Types to Database Semantic integrity
Information Systems, Vol. 5 pp 287-296.

[BRO 81] BRODIE M.L
On modelling behavioural semantics of database
Proc 7th VLDB Conf. Cannes, September 1981.

[BRO 84] BRODIE M.L, MYLOPOULOS J., SCHMIDT J.W.
On conceptual modelling : perspectives from artificial intelligence, databases, and programming languages
On Conceptual Modeling. Springer Verlag, 1984.

[BRV 83] BOGO G., RICHY H., VATTON I.
Description des éléments du modèle de document
RR Tigre No 6. Grenoble. Septembre 1983.

[BUI 86] BUI QUANG N.
Aspects dynamiques et gestion du temps dans les systèmes de bases de données généralisées
Thèse Docteur de L'INP de Grenoble. Novembre 1986.

[BUL 82] BULLEN C.H
A case study of office workstation use
IBM research Lab. San. Jose (USA). RJ3405. March 1982.

[BUN 79] BUNEMAN P, FRANKEL R.E
FQL - A Functionnal Query Langage
Proc of ACM SIGMOD International Conf on the Management of Data, Boston, May 1979.

[BUR 87] BURSENS R., GUYOT J.
Temps + Dynamique + référentiel = Histoire
Troisièmes Journées Bases de Données Avancées. Port-Camargue, Mai 1987.

[CAR 79] CARLSON C.R, ARORA A.K
The updatability of relationnal views based on functional dependencies
Proc. 5th VLDB Conf. Rio de Janeiro, September 1979.

[CAB 85] CABALLERO A.
GIDAF : Gestionnaire Intéractif pour le Développement et l'Activation de Formulaires
Rapport de DEA Informatique. Laboratoire LGI (Grenoble). Juillet 1985.

[CAR 84] CARLSON A.L, WOO C., LOCHOVSKY F.H
Officeaid : An integrated Document Management system
Proc of ACM SIGOA. Conf on office information systems. Toronto, 1984.

[CAZ 83] CAZIN J., JACQUART R., MICHEL P.
FI : Un formalisme de description et de manipulation d'informations et d'expression de la cohérence
Actes des journées Prototypage. Lyon 1983.

[CHA 81] CHAMBERLIN D.D, KING J.C, SLUTZ D.R, TODD S.J.P, WADE B.W
JANUS : An interactive system for document preparation
Proc. ACM. Symposium on text manipulation. June 1981.

[CHE 76] CHEN P.
The Entity Relationship Model - Toward a unified view of data
ACM TODS, Vol 1, No 1, 1976.

[CHR 84] CHRISMENT C., POLETTI F., ZURFLUH G.
Architecture d'un système de gestion de base d'informations généralisées
Actes du séminaire INFORSID. Bandol, Mai 1984.

[CHR 85] CHRISMENT C., ZURFLUH G.
Modèle aggrégatif, langage de manipulation et interface multi-média
Premiéres Journées Bases de Données Avancées. Saint-Pierre de Chartreuse, Mars 1985.

[COD 79] CODD E.F
Extending the database relationnal model to capture more meaning
ACM Transactions on Database Systems. Vol 4, No 4. 1979

[COL 83] COLLET C.
Caractérisation des Applications Formulaires pour une base de données généralisées
Rapport de Recherche TIGRE. No 11. Septembre 1983.

[COL 84] COLLET C.
Gestion des Paramètres d'un document TIGRE
Rapport de recherche TIGRE. No 24. Novembre 1984.

[COL 85] COLLET C.
Les formulaires multimédia
Inforsid 85. Luchon, mai 1985.

[COL 86] COLLET C.
Représentation relationnelle du FA
Note Interne. TIGRE - LGI (Grenoble). Novembre 1986.

[COL 87] COLLET C.
Formulaires Multimédia Dynamiques
Journées de travail sur le thème "Le poste de travail dans les systèmes d'informations". Seyssel, Juin 1987.

[CZE 84] CZEJDO B., EMBLEY D.
Office form definition and processing using a relationnal data model
Proc. ACM SIGOA on office information systems. Toronto, June 1984.

[DEL 83] DELOBEL C., ADIBA M.
Bases de données et systèmes relationnels.
Dunnod informatique, Edition 1983.

[DEM 81] DEMO G.B, and al.
A tool for automatic design of business form databases
Inst. di Science dell' Informazione. Universita' di Torine. Turin, September 1981.

[DOU 87] DOUCET A., LEPENANT C.
Langages de quatrième génération et générateurs d'interfaces
Rapport technique GIP Altaïr No 3-87 (IN2-INRIA-LRI)

[ESC 87] ESCULIER C.
La tolérance sémantique dans les bases de données
Troisièmes Journées de Bases de Données Avancées. Port-Camargue, Mai 1987.

[ELL 80] ELLIS C.A
An office information system based on migrating processes
Xeroc Park. Palo Alto, December 1980.

[FAU 86] FAUVET M.C
CADB : un SGBD pour la CAO
Rapport de recherche TIGRE, No 33. Avril 1986.

[FER 82] FERRANS J.C
SEDL : A language for specifying integrity constraints on office forms
Proc. ACM SIGOA Conf. on ofice information systems. Phyladelphia, June 1982.

[FIK 85] FIK R., KEHLER T.
The role of frame based representation in reasonning
Communication of ACM. Vol 28, No 9. September 1985.

[FIS 83] FISHER P., THOMAS S.T
Operators on Non-First-Normal-Form Relations
Proc of the 7th Int. Computer Software Application Conference, Chicago. 1983.

[GEH 83] GEHANI N.H
High level form definition in office information systems
The Computer Journal. Vol 26, No 1, 1983.

[GIB 80] GIBBS S.
The OFS programmer's manual
Computer systems research group. University of Toronto, december 1980.

[GIB 84] GIBBS S.
An object-Oriented Office Data Model
Technical Report CSRG-154. January 1984.

[GOL 83] GOLDBERG A, ROBSON D.
SMALLTALK 80. The language and its implementation
Addison Wesley Publishing Company. 1983.

[HEO 87] HOEKSTRA R.
Une approche bases de données au support des procédures de bureau
Rapport de DEA informatique. USTMG - INPG Grenoble. Juillet 1987

[HUL 84] HULL R., KING R.
Semantic databases modeling : Survey, applications and research issues
U.S.C Computer Science Technical Report. 1986.

[JAE 82] JAESCHKE B., SCHECK H.J
Remarks on the Algebra of non first normal form relations
Proc PODS, Los Angeles. 1982

[JOL 81] JOLOBOFF V. et al.
Projet base de Données Textuelles
Rapport de Recherche No 273. IMAG, Novembre 1981.

[KAY 81] Projet KAYAK
Actes des Journées sur la bureautique. Paris, Mars 1981.

[KEL 86] KELLER A.M
Choosing a View Update Translator by Dialog at View Definition Time
Proc 12th VLDB Conf. Kyoto. August 1986.

[KIN 84] KING R., McLEOD D.
A unified model and methodoloy for conceptual database design
On Conceptual Modeling. Springer Verlag, 1984.

[KIT 84] KITAGAWA H., GOTOH M., MISAKI S., AZUMA M.
Form Document Management System SPECDOQ - Its Architecture and Implementation
ACM SIGOA on Office Information System. Toronto, June 1984.

[KOR 79] KORNATOWSKI J.Z
The MRS user's manual
Computer systems research group. University of Toronto, July 1979.

[KOR 80] KORNATOWSKI J.Z, CHEUNG C.K
The OFS User's manual
Computer Systems Research Group, university of Toronto. March 1980.

[LAM 84] LAMERSDOFF W., MULLER G., SCHMIDT J.W
Language support for office modelling
Proc of 10th VLDB Conf. Singapore, August 1984.

[LEP 87] LEPENANT C.
Le poste de travail dans le système de gestion ALTAIR
Journées de travail sur le thème "Le poste de travail dans les systèmes d'informations". Seyssel, Juin 1987.

[LER 85] LERAT N. , LIPSKI W.
Une appoche formelle aux valeurs nulles non-applicables
Premières Journées Bases de Données Avancées. Saint-Pierre de Chartreuse, Mars 1985.

[LIN 87] LINGAT J. NOBECOURT P., ROLLAND C.
Gestion de la dynamique des données dans un SGBD relationnel
Troisièmes Journées Bases de Données Avancées. Port-Camargue, Mai 1987.

[LOR 83] LORIE R., PLOUFFE W.
Complex objects and their use in design transaction
RR IBM. RJ 3706. San Jose, June 1983.

[LUC 85] LUCAS P.
On the versatility of knowledge representation.
IBM Research Laboratory. RJ 4573 (49041). January 1985

[LUM 81] LUM V.Y, SHU N.C, TUNG F., CHANG C.L
Automating business procedures with form processing
IBM research Labo. San José 1981.

[LUM 82] LUM V.Y, CHOY D.M, SHU N.C
OPAS : an office procedure automation system
IBM Reserch Lab. RJ3394. San Jose, February 1982.

[MAC 86] RIBARDIERE L. *Manuel 4ième DIMENSION*
Editions A.C.I. 6 avenue Franklin Roosevelt, 75008 PARIS. 1986.

[MAI 85] LE MAITRE J.
OFE : Un langage fonctionnel de manipulation de bases de données
Premières Journées Bases de données Avancées. Saint- Pierre de Chartreuse. Mars 1985.

[MAS 84] MASUNAGA Y.
A relationnal database view update translation mechanism
Proc of 10th VLDB Conf. Singapore, August 1984.

[MIN 75] MINSKY M.
A framework for representing knowledge
PH. Winston Ed. McGraw-hill 1975.

[MOR 77] MORIN E.
La nature de la nature
Editions du Seuil. 1977. Source des nombreuses citations présentes dans cette thèse.

[NEU 87] NEUHOLD E.J, KLAS W., SCHREFL M
Using object-oriented data base sytems for modelling and representing multimedia objects
Whorkshop on Multimedia databases and Complex Objets. Taormina. June 1987.

[NGU 86] NGUYEN G.T, RIEU D.
Semantics of CAD objects for generalized databases
Rapport de Recherche TIGRE, No 22. March 1986.

[OLI 86] OLIVARES J.
Traitement logique de l'intégrité et de l'organisation sémantique des connaissances dans les Systémes de Gestion de Bases de Données
Thèse de Docteur de l'INPG de Grenoble. Juin 1986.

[ORA 84] ORACLE IAF
Manuel utilisateur pour la conception d'application IAF
Oracle Corporation, Menlo Park, California. Original issue : October 1984.

[OSM 78] OSMAN I.M
The solution for some problems of defined relations
IBM UK Scientific Center. Peterlee, January 1978.

[PAO 78] PAOLINI P.
An alternative structure for database management systems
Proc. 4th VLDB Conf. Berlin, September 1978.

[PAR 87] PARENT C., SPACCAPIETRA S.
Un modèle et une algèbre pour les Bases de Données type Entité-Relation
Techniques et Sciences de l'Informatique. 1987. A paraître.

[PLA 87a] PLATEAU D.
Editeur d'objets structurés : un état de l'art
Rapport technique GIP Altaïr No 5-87 (IN2-INRIA-LRI)

[PLA 87b] PLATEAU D.
Interfaces et Bases de Données : un état de l'art et quelques propositions
Rapport technique GIP Altaïr No 6-87 (IN2-INRIA-LRI)

[PIS 86] PISTOR P., ANDERSEN F.
Designing a generalised NF2 model with an SQL-type language interface
Proc 12th VLDB Conf. Kyoto. August 1986.

[PIS 87] PISTOR P.
The advanced Information Management Prototype : Architecture and language interface overview

Troisièmes journées Bases de Données Avancées. Port-Camargue, Mai 1987.

[POT 86] POTTER W.D
A Unified Approach to Modeling Knowledge And Data
IFIP Conf. on Knowledge and Data, Aldeia das Acteias (Portugal). November 1986.

[RAB 81] RABITTI F., GIBBS S.
A distributed form management system with global query facilities
Computer systems research group. University of Toronto, July 1981.

[RIC 87] RICHARD P., BARBEDETTE G.
TOOL : Un langage typé orienté objet pour les bases de données
Troisièmes Journées Bases de Données Avancées, Port-Camargue. Mai 1987.

[RIE 85] RIEU D.
Modèle et fonctionnalités d'un SGBD pour les applications CAO
Thèse de Docteur de L'INPG de Grenoble. Juillet 1985.

[ROT 85a] ROTH M.A, KORTH H.F, SILBERSCHATZ A.
Extended Algebra and calculus for ¬1NF Relational Databases
Technical Report - Departement of Computer Science - University of Texas at Austin (1985).

[ROT 85b] ROTH M.A, KORTH H.F
Null values in ¬1NF Relational databases
Technical Report - Departement of Computer Science - University of Texas at Austin (1985).

[ROT 86] ROTH M.A, KORTH H.F, BATORY D.S
SQL/NF : A Query Language for ¬1NF Relational Databases
Technical Report - Departement of Computer Science - University of Texas at Austin (1986).

[ROW 82] ROWE L.A, SHOENS K.A
A form application development system
Proc. ACM SIGMOD Conf. orlando, May 1982.

[ROW 85] ROWE L.A
"fill-in-the-form" programming
Proc 11th. VLDB Conf. Stockholm, August 1985.

[SCH 82] SCHEK H.J, PISTOR P.
Data structures for an Integrated Database Management and Information Retrieval System
Proc of VLDB conf. Mexico city, August 1982.

[SCH 84] SCHEK H.J, SCHOLL M.H
An algebra for the relational model with relation valued attributes
Technical Report. Technische Hochschule Darmstadt. 1984.

[SHI 81] SHIPMAN D.
The Functional Data model and the Data Language DAPLEX
ACM Transactions on Database Systems, Vol 6, No 1, March 1981.

[SHU 80] SHU N.C, WONG H.K, LUM V.Y
Forms approach to application specifications
IBM Research Lab. San Jose, January 1980.

[SHU 82] SHU N.C, LUM V.Y, TUNG F.C, CHANG C.L
Specification of Forms Processing and Business Procedures for Office Automation
IEEE Transactions on software engineering, Vol 8, No 5. September 1982.

[SHU 83] SHU N.C, WONG H.K, LUM V.Y
Forms Approach to Requirements Specification for Database Design
ACM SIGMOD. International Conference on management of data. San Jose May 1983.

[SHU 85] SHU N.C
FORMAL : A Forms-Oriented, Visual-Directed Application Development System
IEE Transactions on software engineering, 1985.

[SIM 85] SIMON E., VALDURIEZ P.
La compilation de contraintes d'intégrité exprimées dans un langage de haut niveau

Premières Journées Bases de Données Avancées. Saint-Pierre de Chartreuse. Mars 1985.

[STO 75] STONEBRAKER M.
Implementation of integrity constraints and views by query modification
Proc ACM SIGMOD Conf. 1975.

[STO 83] STONEBRAKER M., WOODFILL J.
An implementation of hypothetical relations
Proc of 11th VLDB Conf. Florence, September 1983.

[STO 84] STONEBRAKER M.
Adding Semantic knowledge to a relationnal database system
On Conceptual Modeling. Springer Verlag, 1984.

[TES 86] TESTEMALE C.
Représentation et traitement d'informations incomplètes
Support de cours : Introduction à la logique et ses applications aux Bases de données. Paris Janvier 1986.

[TEX 85] TEXIER M.
System de gestion de formulaires pour les applications bureautiques
Rapport Interne BULL-TRANSAC. Equipe I.A. Septembre 1985.

[TSI 80] TSICHRITZIS D.
OFS : An integrated form management system
Proc. 6th VLDB Conf. Montréal, September 1980.

[TSI 82] TSICHRITZIS D.
Form management
Communications ACM. Vol 25, No 7. July 1982.

[TSI 85] TSICHRITZIS D., NIESTRASZ O.
An Object-oriented Environment For OIS Applications
Proc. 11th VLDB Conf. Stockholm, August 1985.

[TSI 87] TSICHRITZIS D., FIUME E, GIBBS S.
KNOs : KNowledge acquisition, dissemination and manipulation objects
ACM Transactions on office information system. Vol 15 No 1. January 1987.

[VEL 84] VELEZ F.
Un modèle et un langage pour les bases de données généralisées
Thèse Docteur Ingénieur de l'INPG de Grenoble. Septembre 1985.

[VIG 86] VIGNARD P.
Représentations de connaissances - mécanisme d'exploitation et d'apprentissage
INRIA collection Didactique. 1986.

[ZAN 82] ZANIOLO C.
Database relations with null values
ACM SIGACT-SIGMOD, Symposium on principles of database systems. 1982.

[ZAN 84] ZANIOLO C., TSUR S.
An implementation of GEM - supporting a semantic data model on a relational back-end
ACM SIGMOD. Proc of annual meeting. Boston, June 1984.

[ZAN 85] ZANIOLO C.
The representation and Deductive Retrieval of Complex Objects
Proc of 11th VLDB Conf. Stockholm, August 1985.

[ZAN 86] ZANIOLO C. and al.
Object oriented database systems and knowledge systems
Expert Database systems. Benjamin Cumomimgs. Merlo Park, 1986.
ACM 82

[ZDO 84] ZDONIK S.B
Object management System Concepts
ACM 1984

[ZLO 77] ZLOOF M.
Query-By-Example : a database language
IBM systems Journal. Vol 16, No 4, 1982.

[ZLO 82] ZLOOF M.
Office-By-Example : a business language that unifies data and word processing and electronic mail
IBM systems Journal. Vol 21, No 3, 1982.

ANNEXES

ANNEXE 1

Exemples de définitions de Formulaires Abstraits

1- Définition du FA correspondant au formulaire "Demande de mission"

N° _____

MATHEMATIQUES APPLIQUEES – INFORMATIQUE
GRENOBLE

DEMANDE DE MISSION

Nom et prénom: _____

Adresse personnelle: _____

N° sécurité sociale: _____

Mode de paiement: _____

Grade: _____ Echelon: _____ Indice: _____

Date et heure de départ de Grenoble:

le _____ à _____

Date et heure de retour à Grenoble:

le _____ à _____

Si ces dates ont été modifiées, le signaler au retour.

Lieu et motif du déplacement: _____

_____

Imputation de la dépense: _____

Visa du responsable de l'équipe de recherche avec mention des modalités éventuelles de remboursement: _____

_____

_____

Formulaire "Demande de mission"

Définition du FA demande_mission

```
DEF_FA

/* schéma */

demande_mission
begin
        key id_mission : integer end key;
        no_secu : integer;
        nom_prénom : string(20);
        adresse : string(30);
        fonction : string(10);
        indice : integer;
        mode_paiement : string(40);
        lieu_déplacement : begin
                           ville : string(20);
                           pays : string(10);
                           end;
        motif : texte;
        date-mission : begin
                       début_mission : time:
                       retour_mission : time:
                       end;
        imputation : begin
                     type : case of laboratoire; contrat; end;
                     ident_type : string(10);
                     code_type : integer;
                     end;
        groupe : string (15);
        signature_responsable : signature [nullallowed];
end;

/* règles_FA */


DEF_R R1


WHEN créer(demande_mission, ?m);
AFTER
    wait_evt(affecter (?m.id_mission, ?i));
      wait_evt(affecter (?m.no_secu, ?n));
    });
```

DEF_R R2

WHEN affecter(?d.date-mission.début-mission, ?m);
affecter(?d.date-mission.retour-mission, ?r);

BEFORE

si(precede(?r, ?m),
{ message('date retour-mission doit être supérieure à date début-mission');
annuler_evt;
});

DEF_R R3

WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s);

BEFORE

si(et (existe(?d.imputation.type, ?t), (existe(?d.imputation.ident_type, ?i),
(existe(?d.imputation.code_type, ?c)),
{ FA(RES, sélection(listeresponsable, et (=(type, ?t),
=(ident_type, ?i), =(code_type, ?c),
sélection(responsables, ∃ (responsable, =(nom, ?USER))));
si(vide(RES),
{ message('Vous ne disposez pas des droits pour signer une
demande de mission ');
annuler_evt;
});
},
{ message('Avant de signer, veuillez préciser sur quel contrat ou laboratoire
les frais occasionnés par la mission doivent être imputés ');
annuler_evt;
}); /* fin */

DEF_R R4

WHEN créer(demande_mission, ?m); affecter(?m.imputation.type, ?t);
affecter(?m.imputation.ident_type. ?i);
affecter(?m.imputation.code_type, ?c); fin_T;

BEFORE

FA(COD, élaguer(sélection(tablecode, et (=(type, ?i), =(ident, ?i)) : type,
ident);
si(∈(?c, COD),
{ message('code de contrat ou de laboratoire inconnu');
annuler_evt;
});

```
DEF_R R5

WHEN affecter(?d.id_mission, ?i);
BEFORE
si( existe(?d.id_mission), annuler_evt;);
FA( DEM, UNIQUE sélection(demande_mission d, =(?d.id_mission, ?i)));
si( existe(DEM), annuler_evt);


DEF_R R6

WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s)
BEFORE
si( existe(?d.signature_responsable),
    { message("ce champ ne peut pas être modifié");
      annuler_evt;
    } );


DEF_R R7
begin
WHEN créer(demande_mission, ?d); affecter(?d.signature_responsable, ?s);
AFTER
changemode(?d, LE-EFA, all except ?s);
changemode(?d, except ?d.date_mission.retour_mission, LE-EFA, ?s);
end;

DEF_R R8

WHEN créer(demande_mission, ?d); affecter(?d.no_secu, ?n);
BEFORE
FA (PERS, UNIQUE élaguer( sélection( personne p, = (p.no_ss, ?n)) :
                                échelon, salaire) );
si( non(existe(PERS)),
    { message('personne inexistante');
      annuler_evt;}
    );
AFTER
pour (PERS, ?p)
   affecter(?d.nom_prénom, concat(?p.nom, ?p.prénom);
   affecter(?d.adresse, ?p.adresse);
   affecter(?d.indice, ?p.indice);
   affecter(?d.fonction, ?p.fonction);
   affecter(?d.mode_paiement, ?p.mode_paiement);
fin;
```

DEF_R R9

WHEN créer(demande_mission, ?d);affecter(?d.adresse, ?a);
BEFORE
FA(PERS, UNIQUE sélection(personne, = (no_ss, ?d.no_secu));
si(non(existe(PERS)), annuler_evt);
AFTER
pour (PERS, ?p)
  affecter(?p.adresse, ?a);
fin;

DEF_R R10

WHEN créer(demande_mission); fin_T;
BEFORE
si(et (non (existe(?d.date_mission)), non (=(?USER, 'abida'))),
  { message('Il faut donner une valeur aux éléments de *date_mission*');
    annuler_evt;
  });
AFTER
si(non(existe(?d.date_mission)),
  affecter(?d.EXCEPTION, oui)
  );

DEF_R R11

WHEN créer(demande_mission, ?d); fin_T;
BEFORE
si(non(existe(?d.date_mission),
  { FA(MIS_EX, sélection(demande_mission, = (date_mission, ϕ)));
    si(>(count (MIS_EX), 10), annuler_evt);
  })

2- Définition du FA correspondant au formulaire "Remboursement de frais"

Exercice 19	N° d'U.E.R. et compte budgétaire	Mandat

DOIT L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

U.E.R
Laboratoire :
ou contrat :

(labo ou
code
(contrat

à M.
mode de paiement
fonctions
indice réel
se rendant à
dates

(France :
groupe
(Etranger :

VOYAGE A. R. SNCF en classe

l'intéressé(e) (avoir)
DECLARE (ne pas avoir) bénéficié d'une réduction de %

Facture de l'Agence :

ou divers :

SEJOUR départ de Grenoble le à h.
retour à Grenoble le à h.
soit : découchers à F.
repas à F.
arrivée à l'étranger le à h.
départ de l'étranger le à h.
soit : jours à x

frais de mission

arrêté le présent état à la somme de :

frais d'inscription

total dû

Saint-Martin d'Hères, le

Nom et signature
du responsable du
laboratoire
ou
contrat

l'intéressé,

l'ordonnateur,

Formulaire "Remboursement de frais"

Définition du FA : remboursement_frais

DEF_FA

/* schéma */

```
remboursement_frais :
begin
        key id_frais : integer end key;
        entête : begin
                  exercice : time;
                  no-uer : integer;
                  mandat : ident_code;
                  end;
        université : begin
                    identification : string (20);
                    code : ident_code;
                    end;
        envers : begin
                  nom : string(20);
                  paiement : string(30);
                  fonction : string(10);
                  indice : integer;
                  lieu_mission : string(20);
                  date_mission : time_interval;
                  groupe : begin
                            type_groupe : france; étranger; end;
                            valeur_groupe : 1; 2; 3; 4; 5;
                            end;
                  end;
        voyage : begin
                  train : begin
                           aller_retour : string(30);
                           classe : 1; 2; end;
                           prix_train : argent;
                           réduction : Case of
                                        oui : begin
                                              pourcentage : integer;
                                              montant : argent;
                                              end;
                                       non;
                                      end;
                           suppléments : list(0,5) of sup : begin
                                                            ident_sup : string(20);
                                                            prix_sup : argent;
                                                            end;
                           montant_supplément : argent;
                           end;
                  prix_avion : argent;
                  divers : begin
                            séjour_personnel: time_interval;
                            autre : text;
                            end;
```

```
                end;
        séjour : begin
                france : begin
                        départ_grenoble : time;
                        retour_grenoble : time;
                        nb_découchers : integer;
                        prix_découcher : argent;
                        montant_découcher : argent;
                        nb_repas : integer;
                        prix_repas : argent;
                        montant_repas : argent;
                        end;
                étranger : begin
                        arrivée_étranger : time;
                        départ_étranger : time;
                        nb_jours : integer;
                        prix_jour : argent;
                        montant_étranger : argent;
                        end;
                end;
        récapitulatif : begin
                        frais_mission : argent;
                        somme_de : string(60);
                        frais_inscription : argent [nonapplicable];
                        total : argent;
                        avance : argent;
                        reste_dû : argent;
                        date_remboursement : time > hour;
                        responsable_de : string(10);
                        nom_responsable : string(10);
                        signature_responsable : image [nullallowed];
                        signature_intéressé : image [nullallowed];
                        signature_ordonnateur : image [nullallowed];
                        end;
end;

/* règles_FA */


DEF_R R12


WHEN affecter(?r.voyage.suppléments, ?p)
AFTER
affecter(?r.voyage.montant_supplément)
            + (?r.voyage.montant_supplément, ?p.prix-sup));
```

DEF_R R13

WHEN créer(remboursement_frais, ?r);
affecter(?.r.envers.groupe.type-groupe, ?t);
affecter(?r.envers.indice, ?i); affecter(?r.envers.fonction, ?f);

BEFORE

FA(GR, UNIQUE élaguer(sélection(tablegroupe, =(indice, ?i)) : indice) ;
si(non(existe(GR)), annuler_evt);

AFTER

```
si( =(?t, france),
    { si( =(?f, "enseignant"), affecter(?r.envers.groupe.valeur_groupe, 1),
          /* sinon */
          pour (GR, ?g)
            affecter(?r,envers.groupe.valeur_groupe, ?g.grfrance);
          fin;
        );
    },
    /* t = étranger */
    pour (GR, ?g)
      affecter( ?r.envers.groupe.valeur_groupe, ?g.grétranger)
    fin;
);
```

DEF_R R14

WHEN affecter(?r.frais_mission, ?m); affecter(?r.frais_inscription, ?i);
AFTER affecter(?r.total, +(?m, ?i));

DEF_R R15

WHEN affecter(?r.récapitulatif.total, ?t);

BEFORE

```
si( >(?t, 50000),
    { message('Montant du remboursement supérieur à 50000F
                Veuillez confirmez qu'il s'agit d'une exception');
      wait_evt( affecter( ?t.EXCEPTION ) );
      si( = (CWE, 0), annuler_evt,
                      si( =(?t.EXCEPTION, 'non'), annuler_evt )
        );
    });
```

DEF_R R16

WHEN créer(remboursement_frais, ?r); affecter(?r.id_frais, ?i);

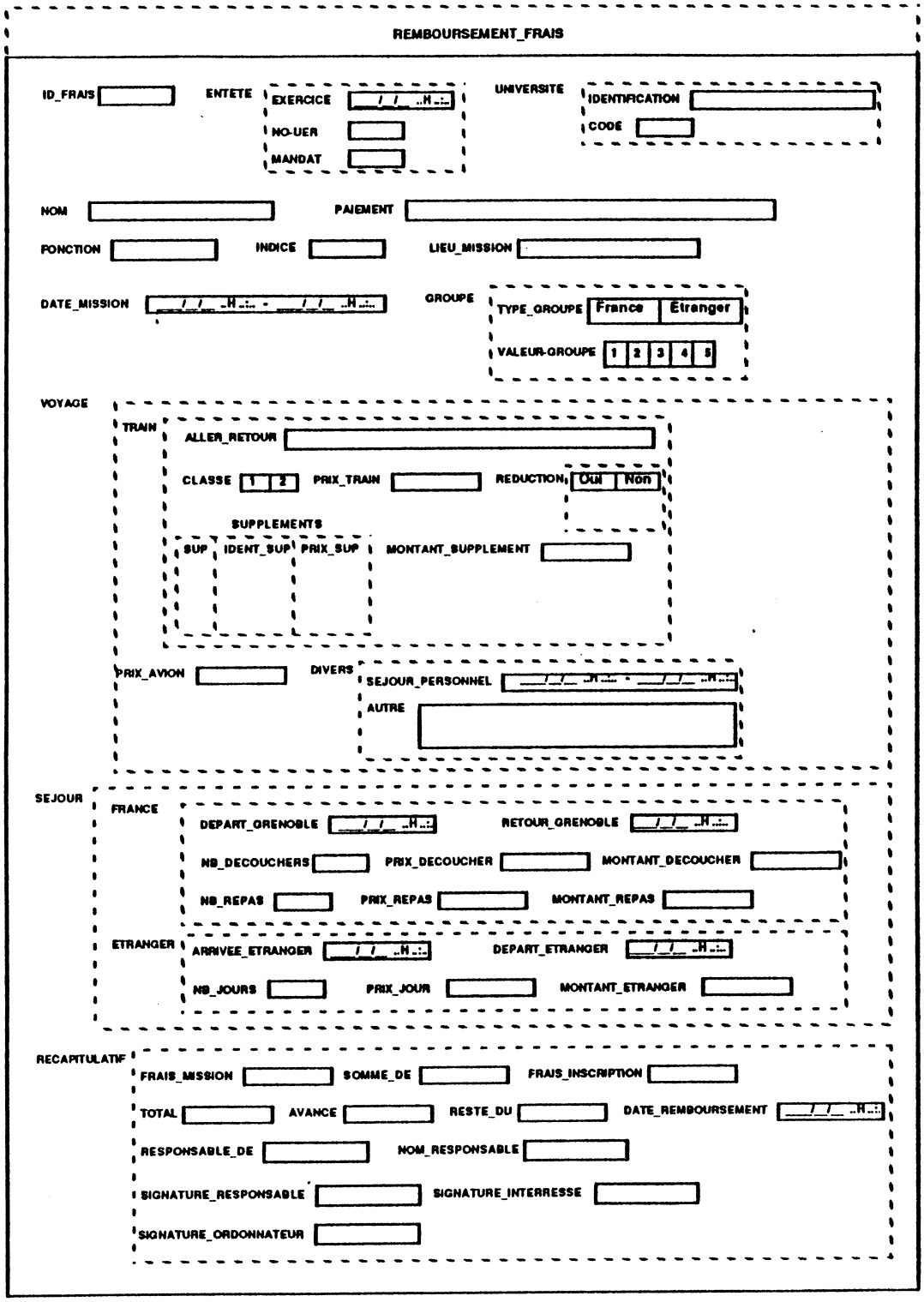
BEFORE

FA(MISSION, UNIQUE sélection(demande_mission, = (id_mission, ?i)));
si(non(existe(MISSION)),
  { message('pas de mission correspondante');
    annuler_evt;
  });

AFTER

pour (MISSION, ?m)
  affecter(?r.envers.nom, ?m.nom_prénom);
  affecter(?r.envers_paiement, ?m.mode_paiement);
  affecter(?r.envers.fonction, ?m.fonction);
  affecter(?r.envers.indice, ?m.indice);
  affecter(?r.envers.lieu_mission,
            concat(m.lieu_déplacement.ville. ?m.lieu_déplacement.pays));
  affecter(?r.envers.date_mission,
   interval_temps(?m.date_mission.début_mission, ?m.date_mission.retour_mis-
   sion));
fin;

Format standard pour le FA remboursement_frais



3- Définition du FA correspondant au formulaire "Professeur"

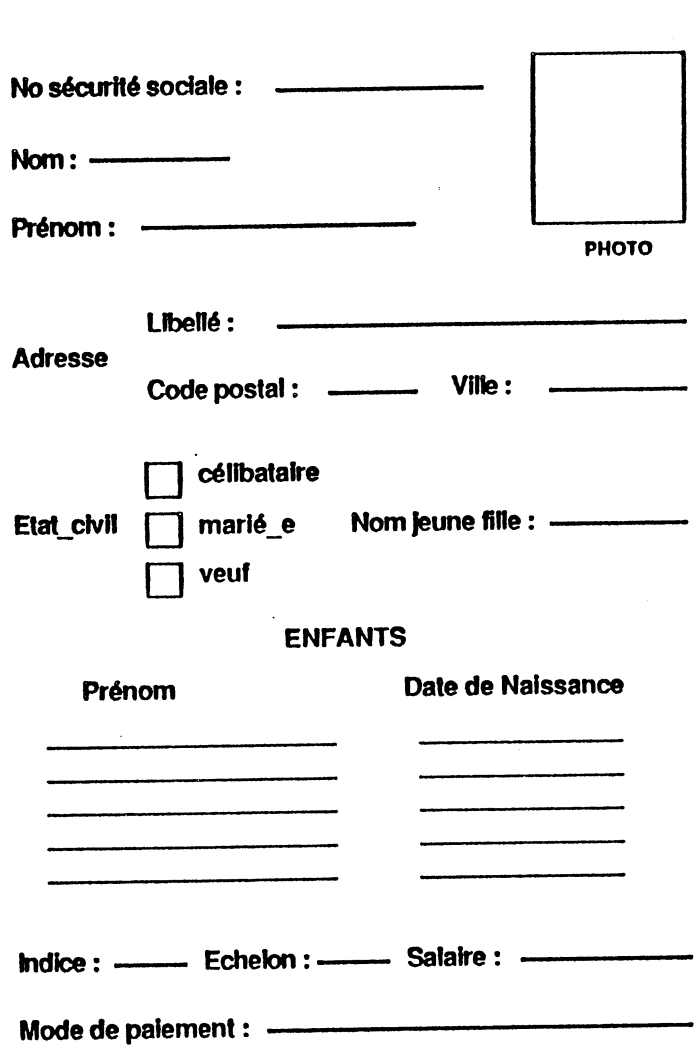


Figure 3 : Formulaire "Professeur"

Définition du FA "professeur"

DEF_FA

/* schéma_FA */

professeur : begin
  car_person : view of
    élaguer (sélection (personne p, = (p.fonction, "professeur")):
    fonction);

```
    photo : image;
    état-civil : case of célibataire; marié_e; veuf_ve; end;
    nom_jf: string (20) [nonapplicable];
    enfants : list (0,5) of enfant :
                        begin
                            key prénom : string (20)end key for enfants;
                            date-naissance : time > hour;
                        end ;
end;
```

Description complète (du point de vue structurel) de l'élément "car_person" :

```
car_person : begin
            key no_ss : integer end key;
            nom : string (10);
            prénom : string (20);
            adresse : string(50);
            indice : integer;
            échelon : integer;
            salaire : integer;
            mode-paiement : string (30);
            end;
```

/* Règles_FA */

DEF_R R17

WHEN créer(professeur, ?p); affecter(?p.car_person.salaire, ?s)
BEFORE

```
si( <=( ?s, ?p.car_person.salaire) ,
  { message('le salaire ne peut pas diminuer');
    annuler_evt } ) ;

FA( SAL, sup×( élaguer( sélection(personne, et ( =(fonction, professeur),
                                                 ≠(no_ss, ?p.car_person.no_ss)) )
                        : nom, prénom, adresse, indice, fonction, mode_paiement)
                )
   );
si ( ≤(  /( +( moy (SAL), ?s), 2) , 12000),
   { message('Avec cette valeur pour le salaire, la moyenne des salaires des
                professeurs devient supérieure à 12000 F');
     annuler_evt; } );
```

DEF_R R18

WHEN créer(professeur, ?pf);
affecter(?pf.car_person.adresse, ?a); affecter(?pf.car_person.indice, ?i);
AFTER
FA(PERS, UNIQUE sélection(personne, et (=(fonction, 'professeur'),
=(no_ss, ?pf.car_person.no_ss))));
pour (PERS, ?p)
affecter(?p.adresse, ?a);
affecter(?p.indice, ?i);
fin;

DEF_R R19

WHEN supprimer(?pf.car_person); fin_T;
AFTER détruire(professeur, ?pf);

ANNEXE 2

Compatibilité de schémas - Opérateurs de comparaison

Dans cette annexe, nous présentons les règles de compatibilité de schémas de FA. Ces règles sont à rapprocher des règles de compatibilté pour les types dans les langages. Nous énoncons les opérateurs de comparaison définis sur les FA.

1- Règles de compatibilité

- Un schéma est compatible avec lui même si l'opération d'égalité (=) est défini entre les occurrences du schéma

- Les schémas de la forme A:entier, A:réel, A:booléen, A:chaîne, A:image, A:texte sont incompatibles entre eux.

- Un schéma S = A: *da* avec *da* $\neq \square$ est compatible avec un schéma S' = A':*da*

  Soit S = A:D et S' = A':D, alors
  dom(S) = {A:a | a $\in$ D} et dom(S') = {A':a' | a' $\in$ D}. On voit bien que S et S' sont compatibles puisque a et a' appartiennent au même domaine D.

  Par exemple, on peut écrire une expression de sélection élémentaire :

  < (frais_inscription, frais_mission)

  "frais_inscription et "frais_mission" sont deux éléments de "récapitulatif" (élément du FA "remboursement_frais"). Leurs schémas sont : "frais_inscription: argent" et "frais_mission: argent". "frais_inscription: *500*" et "frais_mission: *800*" sont des occurrences possibles pour ces schémas. Elles sont définies sur le même domaine "argent".

  Soit S = A: [S1, S2, ..., Sn] et S' = A': [S1, S2, ..., Sn] alors :
  - dom(S) = {A: [s1, s2, ..., sn] | $\forall$ i$\in$ [1...n], si $\in$ dom(Si)}
  - dom(S') = {A': [s1', s2', ..., sn'] | $\forall$ i$\in$ [1...n], si' $\in$ dom(Si)}
  si et si' appartiennent au même domaine (dom(Si)) donc S et S' sont compatibles.

En procédant de la même manière, on montre facilement que des schémas :
- S = A: $<S1>_n$ et S'= A': $<S1>_n$
- S = A: {S1} et S'= A': {S1}

sont compatibles. On utilise les opérateurs de comparaison :

$=, \neq, \subset, \subseteq, \supset, \supseteq$

Soit S = A: S1 | S2 |...| Sn et S' = A': S1 | S2 | ... | Sn alors :
- dom(S) = {A: s | $\exists$ i$\in$ [1...n], s $\in$ dom(Si)}
- dom(S') = {A': s' | $\exists$ j$\in$ [1...n], s' $\in$ dom(Sj)}

Les schémas S et S' sont compatibles mais la comparaison de valeur est possible si et seulement si s et s' appartiennent au même domaine c'est dire et si et seulement si les schémas Si et Sj (choix réalisé) sont compatibles.

Par contre deux schémas S = A:□ et S' = A':□ ne sont pas compatibles puisque leurs domaines de valeur sont respectivement {A} et {A'}.

- Un schéma S = A: S1 | S2 | ...| Sn avec Si = Ai:*di* pour i $\in$ [1...n], est compatible avec un schéma S' = Ai:*di*.

  On peut alors écrire des expressions de sélection élémentaires de la forme : = (A, Ai) ou $\neq$ (A, Ai).

- Un schéma S = A: $<S1>_n$ est compatible avec un schéma S1.

  Par exemple le schéma notes:$<$note:D$>_3$ est compatible avec le schéma note:D. Soit v $\in$ dom(notes) et v = "notes:<n1, n2, n3>" et on peut écrire une expression : notes<2> $\in$ v. L'opérateur $\in$ vérifie si la valeur note<2> appartient à la liste des notes <n1, n2, n3>.

- Un schéma S = A: {S1} est compatible avec un schéma S1 = A1:*dl*.

  De même que pour les listes, on peut écrire une condition (expression de sélection élémentaire) de la forme : A1 $\in$ A.

2- Opérateurs de comparaison

Schémas	Opérateurs
A:entier, A:réel	$=, \neq, >, \geq, <, \leq$
A: D D un domaine ≠ entier, réel, □ A: [S1, S2, ..., Sn] A: S1 | S2 ... |Sn	$=, \neq$
A: $<S1>_n$ A: {S1}	$=, \neq, \subset, \subseteq, \supset, \supseteq$

3- Comportement de la valeur ϕ vis à vis des comparateurs

Soit un FA Γ (S, Δ) et A, A1 des éléments de Γ.

comp est le comparateur "=" ou "≠"

a) ESe ≡ comp(A, ϕ),

pour une occurrence particulière O de Γ,
O ⊨ comp (A, ϕ) ssi O.A comp A:ϕ

b) ESe ≡ comp(A1, A2) A1 et A2 sont des éléments du FA Γ

pour une occurrence particulière O de Γ :
si O.A = A:ϕ et O.A1 = A1:ϕ alors comp(A, A1) est faux
si O.A ≠ A:ϕ et O.A1 = A1:ϕ alors comp(A, A1) est faux

- comp $\in \{ <, \leq, >, \geq \in, \subset, \subseteq, \supset, \supseteq \}$

  comp(A, ϕ) et comp(A, A1) sont faux

4- Relation d'ordre sur les domaines contenant des valeurs nulles

On nomme R("avant") la relation d'ordre définie sur un domaine D et utilisée pour ordonner ses valeurs de manière croissante. On considère le FA F de schéma A:*da* et un élément A1 de ce FA

a) A1 a le schéma A1:D.

   Une valeur pour A1 est A1:*a1*, *a1* $\in$ D-{ϕ} ou bien A:ϕ. Pour une opération **ordonner** (A, asc(A1)), on applique la relation R en considérant que ϕ R *a1* : on donne d'abord les occurrences les moins significatives.

b) A1 a le schéma $A1:D_{nullallowed}$.

   Une valeur pour A1 est A1:*a1*, *a1* $\in$ D-{ϕ,?} ou bien A:?. Pour une opération **ordonner** (A, asc(A1)), on applique la relation R en considérant que ? R *a1* : on donne d'abord les occurrences inconnues.

c) A1 a le schéma $A1:D_{nonapplicable}$.

   Une valeur pour A1 est A1:*a1*, *a1* $\in$ D-{ϕ,∂} ou bien A:∂. Pour une opération **ordonner** (A, asc(A1)), on applique la relation R en considérant que ∂ R *a1* : on donne d'abord les occurrences où l'élément A1 n'a pas été appliqué.

d) A1 a le schéma $(A1:D)_{nullallowed, nonapplicable}$.

   Une valeur pour A1 est A1:*a1*, *a1* $\in$ D-{ϕ, ∂, ?} ou bien A:∂, A:?. Pour une opération **ordonner** (A, asc(A1)), on applique la relation R en considérant les extensions b), c) et ∂ R ?.

Pour une opération **ordonner** (A, des(A1)), on considère la réciproque de R ("après"). On conserve les règles a), b) c) et d) en remplacant R par R^{-1}; Dans le cas a) on a ϕ R^{-1} *a1* : on donne d'abord les occurrences les plus significatives.

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT-INGENIEUR, DOCTORAT USTMG

Vu les dispositions de l'Arrêté du 16 avril 1974,

Vu les dispositions de l'Arrêté du 5 juillet 1984,

Vu les rapports de Mr. ABITEBOUL.....

Mr. DELOBEL.....

Melle. COLLET. christine..... est autorisé

à présenter une thèse en vue de l'obtention du DOCTORAT. U.S.T.M.G.

. Spécialité. Informatique.....

Grenoble, le 16 NOV. 1987

Le Président de l'Université Scientifique
Technologique et Médicale

J.J RAYAN

RESUME

Pour les applications des bases de données multimédia (bureautiques, médicales, CAO), on a besoin de : (1) décrire, stocker et traiter interactivement des objets considérés comme complexes parce qu'ils sont structurés et contiennent des informations de nature alphanumérique ou de nature multimédia : texte, image, voix, etc (2) gérer les aspects dynamiques des objets et (3) établir des liens entre les objets de la base et les objets apparaissant seulement au niveau du schéma externe de la base pour l'application.

Traditionnellement, le formulaire est soit un document servant à décrire, structurer et améliorer la circulation des informations au sein d'une organisation, soit un objet des interfaces entre l'Homme et la machine. Pour les bases de données multimédia, nous intégrons ces deux tendances en proposant un modèle de formulaires complexes et ses opérations associées. Notre modèle se rattache à la classe des modèles de données relationnelles "non sous première forme normale". Il offre un cadre formel pour décrire et traiter la structure, la dynamique et la présentation des objets d'une application comme des formulaires. Cette approche permet de mieux gérer l'intégrité sémantique d'une application en offrant une vision et un traitement homogène des objets d'une base de données au sens large (données classiques ou multimédia, formulaires, documents, etc).

Mots-clés : Formulaires, Bases de données multimédia, Modèle de données, Objets complexes, Algèbre, Dynamique des données, Schéma externe, Interfaces utilisateur.