Traitement logique de l'intégrité et de l'organisation sémantique des connaissances dans les systèmes de gestion de bases de données

Judith Olivares

T H E S E

présentée par

OLIVARES Judith

pour obtenir le titre de **DOCTEUR**

de **l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(arrêté ministériel du 5 juillet 1984)

Spécialité : *INFORMATIQUE*

TRAITEMENT LOGIQUE DE L'INTEGRITE ET DE L'ORGANISATION SEMANTIQUE DES CONNAISSANCES DANS LES SYSTEMES DE GESTION DE BASES DE DONNEES

Date de soutenance : *19 Juin 1986*

Composition du jury :

Président	*C. Delobel*
Rapporteurs	*J. Kouloumdjian*
	J.M. Nicolas
Examinateurs	*M. Adiba*
	J. Mossière
	G.T. Nguyen

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

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

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

.../...

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

PROFESSEURS ASSOCIES

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

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

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

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

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

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

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

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

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

.../...

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

.../...

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

LABORATOIRES EXTERIEURS

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

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : **Monsieur M. MERMET**
Directeur des Etudes et de la formation : **Monsieur J. LEVASSEUR**
Directeur des recherches : **Monsieur J. LEVY**
Secrétaire Général : **Mademoiselle M. CLERGUE**

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

Je tiens à remercier

Monsieur Claude Delobel, Professeur à l'Université Scientifique et Médicale de Grenoble, de me faire l'honneur de présider le jury de cette thèse.

Monsieur Michel Adiba, Professeur à l'Université Scientifique et Médicale de Grenoble et Directeur de cette thèse, pour m'avoir acueilli au sein de son équipe de recherche et pour avoir jugé et apporté des conseils opportuns à la rédaction de ce travail.

Monsieur J. Kouloumdjian, Professeur à l'Université de Lyon I et Monsieur Jean-Marie Nicolas, Directeur de Recherche à "The European Computer-Industry Research Center" de Munich, d'avoir bien voulu apporter leur jugement sur ce travail.

Monsieur Jacques Mossière, Directeur du Laboratoire de Génie Informatique à l'Institut de Mathématiques Appliquées de Grenoble et Professeur à l'Ecole Nationale Supérieure d'Informatique et Mathématiques Appliquées de Grenoble, pour sa participation à ce jury.

Monsieur Gia Toan Nguyen, Chercheur à l'Inria, pour son aide constante tout au long de mon doctorat. Ses conseils et encouragements représentent sans nul doute un facteur décisif dans l'aboutissement de ce travail.

Mon ami Olivier Oudot, pour avoir bien voulu effectuer le dur travail des corrections préliminaires à la rédaction de cette thèse et pour l'encouragement constant et chaleureux qu'il m'a témoigné pendant ces dernières années.

Mes collègues de l'équipe de Bases de Données, pour leur amitié aussi bien à l'intérieur qu'à l'extérieur de l'environnement de travail.

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

Il me faut également remercier la Fondation Vénézuélienne "Gran Mariscal d'Ayacucho" pour son soutien financier sans lequel ce travail n'aurait pu être accompli.

A mis padres

RESUME

Ce travail présente une approche générale du traitement de *l'Intégrité Sémantique* des données gérées par les Systèmes de Gestion de Bases de Données (SGBD). Une organisation des connaissances (lois générales, données, etc) d'une réalité/environnement est également présentée, accompagnée d'une méthode permettant leur définition cohérente. Les idées proposées ont été formalisées au moyen du Calcul des Prédicats du 1er ordre puis mises en oeuvre par l'intermédiaire du langage Prolog.

Mots clés :

Logique et Bases de Données, Intégrité Sémantique, Contraintes d'Intégrité, Systèmes de Gestion de Bases de Données (SGBD) et Déduction, Modèle Logique, Information Incomplète.

TABLE DES MATIERES

INTRODUCTION

Depuis l'apparition des premiers Systèmes de Gestion de Bases de Données vers la fin des années 60, de nombreux SGBDs ont été développés et largement commercialisés aujourd'hui. Ils contribuent pour une grand part à l'amélioration de la représentation et du traitement des informations d'un environnement donné. L'information est d'abord organisée à un niveau conceptuel où l'on conçoit une représentation abstraite de l'environnement puis représentée selon un schéma dans un langage fournit par le SGBD. Cette phase s'effectue de façon indépendante des traitements spécifiques que doivent subir les données. Celles-ci sont ainsi stockées dans une base de données (BD) en accord avec un schéma de représentation interne en complète conformité avec le schéma conceptuel.

Tout SGBD soutient un *modèle conceptuel* auquel est associé son *schéma conceptuel* de représentation. Ces schémas permettent, en général, d'expliciter le côté statique d'un environnement supposé complètement connu. Leur évolution pendant ces dernières années, se situe essentiellement sur :

- l'inclusion, dans le schéma, du côté dynamique de l'environnement (leurs opérations propres) [CASA 82], [STON 84],
- la représentation d'objets à structure complexe voire de grande taille [LOPE 83],
- la spécification des informations à caractère incomplète [LEVE 84].

Sous les schémas logiques, basés sur une logique particulière, la représentation de l'information est faite par des formules bien formées du langage associé. Mise à part le fait de fournir un cadre conceptuel rigoureux pour l'étude et l'analyse des problèmes liés à la représentation et le traitement des informations, les schémas logiques et les méthodes de preuve automatique de théorèmes ouvrent plus particulièrement un chemin dans :

- la spécification et le traitement des propriétés sous forme de lois générales pour les informations,

- le traitement uniforme de l'intégrité sémantique de l'information,
- la spécification et le traitement des informations implicites par des lois générales.

Des améliorations dans ces domaines particuliers permettent d'élargir les capacités des SGBDs. Elles ont également des répercussions directes sur les applications se traduisant par une diminution du volume de données à stocker dans la base de données et un haut degré de correction sémantique de l'information.

Ce travail orienté dans cette voie se base en particulier sur la logique des prédicats du 1er ordre. Nous proposons un méta-schéma ou ensemble de formules logiques (méta-axiomes) dont l'organisation sémantique implicite permet, par un schéma particulier à un environnement, de traiter d'une façon uniforme, intégrale et cohérente les aspects qui concernent la sémantique et la dynamique de l'environnement.

Les idées soutenues ici sont concrétisées au moyen d'une mise en oeuvre hypothétique d'un Système Logique d'Intégrité Sémantique à trois facettes différentes et que nous appelons SYCSLOG. L'objectif principal de ce système serait ainsi d'assurer un traitement uniforme, et le plus complet possible, de l'intégrité sémantique de l'information d'une application. Un deuxième objectif serait de permettre l'intérrogation des informations implicites et des données via le SGBD.

Ce système s'appuie sur un modèle logique dont la spécification correspond au méta-schéma proposé dans la théorie. La base du système est de pouvoir établir des correspondances entre les objets conceptuels d'un modèle conceptuel donné et les objets logiques du modèle.[1] De cette façon, tout schéma conceptuel peut être traduit de façon automatique dans un schéma logique (ensemble de formules bien formées sous forme de clauses) qui constitue l'outil de base du traitement effectué par SYCSLOG pour l'application.

Un schéma logique est stocké dans une base de connaissances (BC). L'une des composantes principales de SYCSLOG est une base de méta-connaissances (META-LOG) qui contient la description syntaxique et sémantique de chaque objet logique. Une interface logique d'accès aux connaissances (LCD) exploite la base META-LOG et simultanément, pour chaque application, un échantillon de la BD est construit assurant ainsi une évolution cohérente de la BC.

[1] Une correspondance à été établi, entre les objets conceptuels du modèle de données TIGRE [LOPE 83] et les objets logiques de SYCSLOG dans [NGUY 85]$_2$.

L'interface logique de vérification des données (LVD) est la porte d'accès à SYCSLOG pour une transaction sous le SGBD. Elle effectue toutes les vérifications sémantiques, en fonction des spécifications dans la BC, mises en cause par la transaction.

Dans le chapitre 1 nous présentons les approches les plus connues dans le traitement de l'intégrité sémantique. Leurs caractéristiques, du point de vue du langage de spécification des contraintes d'intégrité (CIs) et des mécanismes de vérification et de maintien de la cohérence, sont présentées d'une façon générale.

Le chapitre 2 contient une partie introductive aux principaux termes et concepts logiques que nous utilisons aux chapitres suivants. Une deuxième partie passe en revue les principaux résultats obtenus dans l'étude du domaine Bases de Données par la Logique. Le chapitre se termine avec une description de l'approche théorique de SYCSLOG pour une bases de données. Dans cette approche nous présentons une catégorisation des axiomes de la théorie du 1er ordre qui définit la BD. Nous verrons que cette approche correspond d'une façon générale au point de vue de théorie de la preuve soutenu par Reiter [REIT 84].

Le chapitre 3 présente une description de l'architecture et des principales composantes de SYCSLOG.

Le chapitre 4 décrit le modèle logique de SYCSLOG et les méta-règles qui le définissent dans META-LOG.

Le chapitre 5 contient dans une première partie la description de la méthode permettant d'assurer l'intégrité sémantique des BCs : la présentation et la preuve du théorème de l'échantillon d'une BD ainsi que la méthode de construction et d'exploitation de cet échantillon. La deuxième partie est dédiée à une description plus détaillée de l'interface logique de vérification des données.

Enfin, nous décrivons dans les conclusions trois différentes facettes d'utilisation de SYCSLOG et les perspectives envisageables sous cette approche.

CHAPITRE 1

LE CONCEPT D'INTEGRITE SEMANTIQUE DANS LES SGBD

CHAPITRE 1

LE CONCEPT D'INTEGRITE SEMANTIQUE DANS LES BASES DE DONNEES

Depuis leur apparition dans les années 40, les ordinateurs sont utilisés pour le traitement d'informations dans différents domaines scientifiques, commerciaux, etc. Une méthode élémentaire pour obtenir la solution d'un problème consiste à chercher, par l'analyse, un algorithme ou une procédure de résolution de ce problème : un calcul, un traitement de données, etc. Puis l'algorithme est écrit comme un ensemble d'**instructions**, dans un langage de programmation adapté au domaine du problème.[1] Ce langage fournit généralement un ensemble de stuctures prédéfinies de stockage physique de données (des tableaux en mémoire principale, des fichiers en mémoire secondaire, etc) parmi lesquelles on effectue un choix pour stocker les données du problème. Le résultat est un **programme** qui exécute les instructions et utilise les données stockées pour produire un résultat ou effectuer un traitement sur les données.

Cette technique utilisée au début de l'informatique, provoque la génération d'un ensemble de programmes indépendants les uns des autres dans le sens où chaque programme gère ses propres données. Une conséquence naturelle est une prolifération de données qui entraîne des incohérences. Ainsi par exemple, si deux programmes traitent une même information (ex. le salaire d'un employé) et si l'un d'eux doit modifier la donnée correspondante alors il y a pour l'employé, à ce moment, deux salaires différents stockés dans deux fichiers différents. Ceci est naturellement incohérent.

Plus tard comme palliatif surgit le partage de données par les programmes. De cette manière, le salaire d'un employé se trouve stocké dans un seul fichier de données qui est partagé par les programmes. Ceux-ci doivent contenir la même description de la structure physique de stockage des données. La conséquence immédiate de ce choix est le statisme de la structure de stockage des données. Sa

[1] Le langage Fortran pour le calcul scientifique, le langage PL1 pour les applications commerciales, etc.

modification entraîne la modification de tous les programmes qui l'utilisent accroissant ainsi la dépendance entre données et programmes.

A ce stade (1962), les premiers Systèmes de Gestion de Bases de Données (SGBD) font leur apparition. Le but principal est d'assurer l'indépendance entre les données et les programmes (**indépendance de données**). Les données d'un environnement sont représentées dans un langage déclaratif, le **Langage de Définition de Données (LDD)**, en accord avec un **modèle conceptuel**. Cette représentation est obtenue de façon indépendante des traitements effectués sur les données par l'environnement.

Les données sont ainsi définies à deux niveaux :

au niveau externe

celui-ci est connu par les programmes et est indépendant de la structure de stockage de données (*schéma externe*).

au niveau interne

celui-ci est invisible aux programmes et contient les descriptions des structures physiques de stockage (*schéma interne*).

La modification de la structure physique de stockage de données entraîne ainsi uniquement la modification du schéma interne. La correspondance entre les schémas externe et interne est assurée par des routines du SGBD.

Sous un SGBD, le traitement de données est effectué via le niveau externe moyennant un langage spécial appelé **Langage de Manipulation de Données (LMD)**. Ce langage fournit des commandes et des instructions (insérer, modifier et éliminer) qui exécutent des opérations atomiques sur la base de données (BD). Les demandes de traitement de données sont effectuées par des *usagers* de la BD qui formulent leurs **requêtes**, soit *d'intérrogation* simplement soit de *modification* de données.

Le modèle conceptuel fournit par le SGBD comporte un ensemble d'objets conceptuels qui sont utilisés pendant la **modélisation** de l'environnement. L'environnement est ainsi vu comme un ensemble d'objets chacun étant associé à un objet conceptuel. De nombreux modèles conceptuels ont été proposés depuis l'apparition des SGBDs. Nous citons les plus répandus jusqu'à présent.

Dans le **modèle hiérarchique** (1967) les objets conceptuels sont des ensembles d'entités et des associations hiérarchiques entre ensembles d'entités. C'est à dire, des associations du type (1:N) une entité (père) à N entités (fils).

Le **modèle réseau** (1969) est une généralisation du modèle hiérarchique ou les associations ne sont pas seulement 1:N, mais M:N.

Dans le **modèle relationnel** (1970) les objets conceptuels sont des **relations** au sens mathématique du terme. Une relation est un ensemble de **n-uplets** obtenus par le produit cartésien de n ensembles de valeurs ou **domaines**.

Nous avons présenté un bref aperçu de l'évolution du traitement des données jusqu'à l'apparition de Systèmes de Gestions de Bases de Données. Dans ce qui suit, nous allons définir la notion d'intégrité de données et par la suite nous dédier plus particulièrement à l'intégrité sémantique des données.

1. La Notion d'Intégrité

L'intégrité des données a aujourd'hui un grand champ d'application groupant tout ce qui fait référence à la cohérence logique de données :

Gestion des transactions

Une *transaction* est une suite finie d'actions sur les objets BD qui fait passer la BD d'un état cohérent dans un autre état cohérent [ADIB 82]. La gestion de transactions concerne les mécanismes qu'il est nécessaire d'assurer pour qu'une transaction soit exécutée ou dans son ensemble ou pas du tout.

Concurrence d'accès

Elle assure le traitement cohérent de l'information lors du partage de la BD par plusieurs usagers, c'est à dire lorsque plusieurs transactions s'exécutent concurremment. Les mécanismes qui assurent la concurrence d'accès se chargent de trouver un *ordonnancement cohérent* pour les actions des transactions exécutées concurremment.

Intégrité sémantique

Il s'agit d'assurer que les données stockées sont sémantiquement correctes.

Intégrité syntaxique

Elle concerne la correspondance entre le schéma interne (définition de structures de stockage, chemins d'accès, indices, etc) et les données sur les mémoires sécondaires (stockage physique, accès, etc).

Sécurité de fonctionnement

Elle concerne les incohérences produites en cas de pannes du matériel et du logiciel.

L'*intégrité* est une des caractéristiques vitales des Bases de Données en conjonction avec *la confidentialité* et *l'indépendance physique et logique* des données. La confidentialité assure que les données seront accédées par les usagers selon leurs droits d'accès. L'indépendance physique/logique permet la modification de données au niveau de la représentation interne/conceptuelle sans aucune conséquence sur les programmes d'application.

Depuis le début de leur exploitation, les SGBDs ont assuré ces caractéristiques avec une plus ou moins grande réussite. De nombreaux sous-systèmes spécialisés ont été conçus pour la gestion de transactions et la gestion d'accès concurrents pour des environnements centralisés ou répartis. De même, la correspondance entre les données et leur représentation conceptuelle est assurée par des procédures spécialisées chargées de contrôler toute modification des données.

Les tendançes actuelles sont de chercher à améliorer de plus en plus les performances de la gestion de transactions et d'accès concurrents sous environnements centralisés ou répartis et pour des bases de données contenant une plus large gamme de données (images, voix, etc). Des études de l'intégrité sémantique sur des SGBDs dotés de capacités déductives [NICO 77], [REIT 79], [CHAN 79] ont été menées en utilisant la logique en tant qu'outil conceptuel d'aide à la modélisation et en tant qu'outil inférentiel pour des traitements déductifs. L'approche "types abstraits de données" [BROD 79], [BROD 80], [STON 84] est explorée pour doter les SGBDs de mécanismes de spécification de haut niveau et de vérification efficace de la sémantique des applications.

2. L'Intégrité Sémantique

Le champ d'application de l'intégrité sémantique concerne toutes les activités, dans un SGBD, qui assurent que les données stockées physiquement dans une BD sont "correctes" d'un point de vue de leur sémantique ou de leur signification.

Assurer que les données sont sémantiquement correctes implique avoir un certain nombre de critères ou de règles qui permettent au SGBD de décider si les données sont en "bon état", c'est à dire **cohérentes** selon ces critères. Par exemple, les données correspondantes aux salaire des employés d'une entreprise peuvent être considérées correctes si elles sont numériques, supérieures au SMIG (4000 FF) et inférieures à un plafond (50000 FF).

Avant l'apparition du modèle relationnel, l'intégrité sémantique était assurée par les SGBDs grâce à des procédures spécialisées, codées dans un langage procédural, appelées par les routines de contrôle d'accès [DATE 77]. D'autres contrôles sémantiques ne pouvaient être créés qu'au niveau des programmes de l'application, entraînant le statisme et la rigidité des applications. Une modification d'une procédure d'intégrité à l'intérieur d'un programme faisait donc atteinte au concept d'indépendance de données, tant mis en valeur par les SGBDs.

Aujourd'hui, les critères ou règles de correction sont des énoncés de cohérence appélés contraintes d'intégrité sémantique ou tout simplement **contraintes d'intégrité** (CI).

Supposons maintenant que nous établisons par une CI que le salaire d'un employé doit être toujours supérieur au SMIG (4000FF) et en même temps inférieur à 3000FF. Cette contrainte est logiquement contradictoire ou incohérente. L'établissement d'un ensemble de CIs cohérentes entre elles est donc une activité de départ indispensable pour pouvoir assurer la correction sémantique des données d'une application.

La vérification de l'intégrité sémantique pour une application se décompose donc en trois phases :

1) *la définition des contraintes d'intégrité,*
2) *la vérification de la cohérence des contraintes d'intégrité et*
3) *la vérification de la cohérence des données.*

La première et la deuxième phase doivent s'effectuer toujours ensemble pendant la période de modélisation de l'application. Ces phases accomplies, la troisième est

exécutée tout au long de la vie de la base de données de l'application.

2.1. Les contraintes d'intégrité

Les SGBDs relationnels [CODD 70], depuis leur début, ont compté parmi leurs objectifs une amélioration de l'intégrité. Ceci a été atteint par la spécification, dans un langage déclaratif, d'énoncés d'intégrité au niveau du schéma conceptuel (**contraintes d'intégrité**). L'idée consiste à exprimer les contraintes sur les données relationnelles par des prédicats ou des assertions.

Une **contrainte d'intégrité** (CI) est donc une assertion exprimée dans un langage déclaratif. Celui-ci peut être un langage d'intérrogation servant en même temps de langage de définition de données, basé sur le calcul relationnel, étendu pour permettre la spécification explicite de contraintes comme dans la plupart des approches relationnelles. Il peut être aussi un langage servant uniquement à définir des données et fournissant des clauses spéciales pour ces spécifications. Ces langages sont toujours de haut niveau, faciles à utiliser et leur capacité d'expression est assez élevée. Le calcul de prédicats du 1^{er} ordre est aussi utilisé comme outil de spécification de CIs donnant une portée plus générale à la spécification.

Des nombreuses classifications ont été proposées dans la littérature sur le traitement de CIs. Parmi ces classifications nous pouvons citer les suivantes :

contraintes mono-relationnelles ou multirelationnelles

selon que la contrainte porte sur une seule ou plusieurs relations.

contraintes individuelles ou d'ensemble

selon que la contrainte porte respectivement sur un seul n-uplet ou sur plusieurs.

contraintes statiques ou dynamiques

selon que la contrainte porte sur un seul état de la BD ou sur des états consécutifs.

contraintes immédiates ou différées

selon que leur vérification est effectuée sans délai ou retardée jusqu'à la fin de la transaction qui contient l'action qui met en cause la contrainte.

2.2. La cohérence des contraintes d'intégrité

Dans ce domaine, peu abordé dans la littérature sur l'intégrité sémantique, il s'agit d'assurer qu'il n'y a pas de conflicts entre les contraintes d'intégrité. L'approche "Logique et BDs" permet de le traiter à partir du concept de cohérence logique d'un ensemble de formules bien formées. Une formule bien formée est cohérente avec un ensemble de formules si et seulement si l'on ne peut pas démontrer au même temps la formule et sa négation à partir de l'ensemble de formules.

2.3. La vérification de la cohérence des données

La **vérification** de la cohérence des données est faite par des algorithmes qui, à partir d'une requête de modification et de l'état actuel de la BD, déterminent pour toute assertion si elle continue à être valide (vraie) sous le nouvel état obtenu après l'exécution de la requête de modification. Cette vérification est effectuée par des mécanismes auxquels on demande de l'efficacité, un coût minimal et de la généralité. La plupart d'entre-eux sont exécutés en général avant l'exécution de la requête, n'autorisant ainsi leur exécution que si elle n'amène pas la BD dans un état incohérent. D'autres sont exécutés après l'exécution de la requête, déclenchant à posteriori des mécanismes de "défaite" de la transaction s'ils trouvent que la BD est passée dans un état sans aucune suite. Ceux-ci sont de moins en moins mis en oeuvre à cause de leur coût élevé.

Le **maintien** de la cohérence des données concerne les actions qui doivent être exécutées automatiquement lors de violations des contraintes. Les actions : correctives ou informatives sont définies par le langage de spécification de CIs.

3. Des Sous-systèmes d'Intégrité Sémantique

Un *Sous-système d'Intégrité Sémantique* dans un SGBD doit assurer la spécification *cohérente* de CIs, la vérification et le maintien de la cohérence des données d'une façon efficace et au moindre coût. Parmi les caractéristiques principales souhaitées pour un sous-système d'intégrité sémantique nous citons les suivantes :

a) Fournir un langage de spécification : le langage de spécification de CIs fournit par un sous-système d'intégrité sémantique doit être :

- **simple :** de façon à pouvoir être utilisé par une large gamme d'utilisateurs
- **de haut niveau :** permettant ainsi la spécification de CIs par des assertions déclaratives sans avoir besoin de spécifier leur procédures de vérification. Ceci doit relever du sous-système d'intégrité sémantique
- **précis :** ne permettant pas d'ambiguités de signification des assertions
- **complet :** le langage permet de spécifier toute contrainte sémantique sur les données
- **général :** permettant ainsi la spécification d'un plus grand nombre d'assertions

En outre un langage de spécification de CIs doit permettre:

- **la spécification cohérente de CIs :** celle-ci doit être assurée de manière à ce que les CIs définissent des états possibles pour les données de la base.
- **la dynamicité des assertions :** la spécification et la modification d'assertions doit pouvoir se faire à n'importe quel moment pendant la durée de vie d'une application. Cette dynamicité doit être accompagnée par des mécanismes qui assurent que la nouvelle assertion est valide sur l'état actuel de la BD
- **actions en cas de violation :** la spécification d'actions à exécuter de façon automatique lors de violations
- **actions de maintien de l'intégrité :** la spécification d'actions à exécuter de façon automatique lors de certaines mises à jour de données.

b) Fournir des mécanismes de vérification de la cohérence de données avec les caractéristiques suivantes :

- **complets :** la complétude dans ce sens signifie la vérification de toute assertion spécifiée moyennant le langage
- **exhaustifs :** tout énoncé de mise à jour de données doit pouvoir être traité pour assurer qu'il n'amène pas la BD dans un état incohérent
- **généraux :** la généralité du mécanisme de vérification permet d'assurer la cohérence de données pour des mises à jour générales, évitant ainsi la prolifération d'algorithmes qui dépendent de chaque mise à jour à effectuer sur les données.

c) Fournir des mécanismes d'aide à la conception de CI : des contraintes induites par des assertions déjà définies doivent être incorporées de façon automatique à l'ensemble d'assertions.

d) Fournir des mécanismes de maintien de la cohérence de données : ces mécanismes doivent exécuter, en cas de violation de la cohérence de données, des actions automatiques spécifiées dans les contraintes ou dans les mises à jour spécifiques. Des mécanismes de stockage d'information statistiques sur la vérification de CI doivent être offerts. Des informations telles que : fréquence de vérification d'une CI, temps d'exécution d'une CI, etc, permettent ainsi d'effectuer des mesures sur les rapports efficacité/coût pour les CI.

Dans ce qui suit nous passons en revue les approches les plus significatives dans le traitement de CIs dont la plupart ont été proposées pour les SGBD relationnels. Les différences de base entre les différentes approches pour le traitement de CIs reposent surtout sur les mécanismes de vérification et de maintien de la cohérence des données. Les méthodes décrites, d'une façon générale, sont difficiles à comparer d'un point de vue de l'efficacité et du coût impliqué par ces traitements.

3.1. Approche INGRES

Les contraintes d'intégrité dans le SGBD relationnel INGRES [STON 75] sont exprimées en QUEL ("QUEry Language") par des assertions. Chaque assertion est composée d'une partie déclarative comportant des relations intervenant dans la contrainte : la déclaration RANGE assigne une variable n-uplet à chaque relation.[1] Une partie qualificative permet de qualifier les variables n-uplets apparaissant dans la déclaration RANGE. Cette qualification est une série de conditions liées par les opérateurs logiques AND et OR préfixés par le mot INTEGRITY. Par exemple, l'énoncé suivant [STON 77] exprime une dépendance fonctionnelle entre un étage et un département :

```
RANGE OF D IS DEPARTEMENT
INTEGRITY COUNT(D.ETAGE#  BY  D.DEPT) = 1
```

Les mots ETAGE# et DEPT sont des attributs de la relation DEPARTEMENT. La dépendance est exprimée par le fait qu'un n-uplet, pour un étage et un département donnés, doit apparaître une seule fois (COUNT= 1) dans l'extension de la relation DEPARTEMENT.

---

[1] Les instantiations d'une variable n-uplet d'une relation sont ses n-uplets.

La capacité d'expression des énoncés d'intégrité dans QUEL est assez large, néammoins l'expression des contraintes d'intégrité dynamiques n'est pas envisagée ni l'expression de contraintes différées.

La méthode de vérification de CIs sous INGRES est une des plus répandues dans les SGBDs relationnels. Le principe de base est le suivant :

Toute interaction avec les données est immédiatement contrôlée au niveau du langage d'interrogation pour garantir l'absence de violation d'intégrité.

Ainsi, la vérification d'une CI mise en question par une requête de modification donnée RM se réduit à l'obtention d'une qualification plus contraignante pour la qualification de RM. Donc, la modification de données est effectuée si et seulement si cette qualification contraignante est vraie.
La méthode dispose de quatre algorithmes de coût croissant pour déterminer la nouvelle qualification de RM, selon que les assertions portent sur :

1) une variable de n-uplet libre d'agrégats,[1]

2) plusieurs variables de n-uplet libres d'agrégats avec une seule variable de n-uplet portant sur la relation qui est en train d'être modifiée,

3) plusieurs variables de n-uplet libres d'agrégats avec deux ou plusieurs variables de n-uplet portant sur la relation qui est en train d'être modifiée,

4) des agrégats

Cette méthode possède l'avantage de ne pas avoir besoin d'algorithmes de vérification spéciaux pour les CIs. Cela relève de l'interpréteur ou compilateur du langage d'intérrogation. Cependant, la nature de la méthode empêche le traitement de contraintes en différé et le déclenchement d'actions lors de violations.

Récemment, Stonbraker [STON 84] a proposé une extension à cette approche initiale pour le SGBD INGRES. Des règles d'intégrité sont spécifiées dans le langage RAISIN ("Rules from AI Specified for INGRES"). Une règle est une séquence de clauses de la forme :

ON (*condition*) THEN (*action*)

Les *conditions* spécifient des contraintes sur la requête de modification à vérifier

[1] Un agrégat est une fonction qui associe à un ensemble de valeurs dans la BD une valeur. Celle-ci peut être la moyenne statistique de l'ensemble de valeurs, la valeur minimale ou maximale de l'ensemble, etc.

avant l'exécution des *actions*. Si ces contraintes sont validées, les actions puis la requête de modification sont exécutées. Sous cette approche l'expression de contraintes dynamiques a été envisagée par l'introduction du mot clé NEW pour désigner la nouvelle valeur pour un champ. Par contre, les contraintes différées ne sont pas considérées dans leur totalité. On peut seulement différer une action jusqu'à la fin d'une commande spécifiée dans la partie *condition*.

La vérification de CIs sous cette approche est faite par le système général de traitement de règles du langage RAISIN. Des détails sur celui-ci n'ont pas été donnés par l'auteur; cependant, sous cette nouvelle approche peut être fourni un mécanisme de maintien d'intégrité grâce à l'exploitation des actions spécifiées dans les règles RAISIN.

3.2. Approche HAMMER et MC. LEOD

Cette approche [HAMM 75] classe les CI en deux catégories :

contraintes de domaine : pour la définition de domaines

contraintes de relation : pour spécifier des propriétés ou des associations entre des relations.

Les auteurs proposent l'utilisation d'un langage de haut niveau qui contient un langage d'intérrogation tels que : QUEL, SEQUEL, etc. Ce langage doit permettre la spécification de :

la nature de la contrainte : de n-uplet ou de colonne, selon que la contrainte porte respectivement sur les valeurs dans les n-uplets de la relation ou sur les valeurs d'un attribut de la relation.

certaines dépendances entre relations : par l'utilisation de clauses : *one-to-one* et *subset-of*. Ces dépendances sont établies entre les valeurs de colonnes de deux relations.

actions de maintien : permettant d'indiquer des messages à envoyer lors de violation ou de validité de la contrainte. Des actions correctives à déclencher automatiquement sont aussi considérées.

Leur approche offre la possibilité d'exprimer une gamme assez large : contraintes avec des agrégats, contraintes de fonctionnalité, contraintes dynamiques, contraintes différées jusqu'à l'exécution d'une opération donnée par la clause

"enforcement", etc.

La vérification et le maintien de CIs est assuré par les composants du sous-système d'intégrité sémantique suivants :

le processeur du langage : traduit les CIs dans un format interne

le vérifieur de CIs : détermine les contraintes à vérifier (travaillant sur le format interne) après l'exécution d'un ou plusieurs changements dans la BD et effectue les vérifications correspondantes

le processeur violation-action : exécute les actions indiquées dans les contraintes en cas de violation

le contrôleur de compatibilité : doit assurer que l'ensemble de contraintes est libre de conflits et des propriétés indésirables. Des détails sur celui-ci n'ont pas été donnés par les auteurs.

Le format interne des contraintes et les algorithmes utilisés par le vérificateur de contraintes ne sont pas décrits par les auteurs. Le maintien de l'intégrité est assuré par le processeur violation-action. Leur approche donne une portée étendue et intégrée au traitement de CIs.

3.3. Approche Système R

Une autre approche est celle de Eswaran et Chamberlin [ESWA 75] proposée pour le système R et le langage d'intérrogation SEQUEL. La classification est la suivante :

- assertions sur n-uplet *vs* assertions d'ensemble
- assertions d'état *vs* assertions différées
- assertions appelées lors de : mise à jour, insertion, élimination et intérrogation
- assertions faibles *vs* assertions fortes : les assertions faibles produisent un message d'erreur mais n'interdisent jamais une opération donnée. Les fortes expriment une interdiction.

Par rapport aux deux approches antérieures une nouvelle possibilité est ajoutée : de pouvoir expliciter les opérations pour lesquelles l'assertion devra être vérifiée. En outre, dans SEQUEL a été inclu la possibilité de spécifier des actions spontanées ("triggers") déclenchées automatiquement lorsque des opérations se produisent dans la BD.

Pour la vérification et le maintien de CIs les auteurs proposent un sous-système d'intégrité sémantique similaire à celui de Hammer et Mc Leod.

3.4. Approche SGBDs déductifs

Nicolas et Yazdanian [NICO 78] ont proposé une approche pour la spécification de l'intégrité sémantique pour des SGBDs déductifs. Leur approche consiste à utiliser des **lois générales** classées en :

- **lois d'état** : faisant référence à un seul état de la réalité
- **lois de transition** : faisant référence aux états consécutifs d'une réalité.

Certaines **lois d'état** sont utilisées en tant que **règles de dérivation** pour dériver des informations implicites, d'autres en tant que **règles d'intégrité** pour le maintien de la cohérence de la BD. Les règles d'intégrité sont des formules du calcul de prédicats du 1er ordre. Les lois de transition sont exprimées en fonction de nouvelles relations appelées *relations d'action*. Pour toute relation dans la BD, trois relations d'action sont assignées : UPD-R, DEL-R et ENT-R pour mise à jour, élimination et insert, respectivement. Ces relations contiennent (uniquement lors des vérifications des lois de transition) :

- si UPD-R : un n-uplet contenant les données correspondantes à la mise à jour, avant et après
- si DEL-R : le n-uplet à éliminer
- si ENT-R : le n-uplet à ajouter

Une loi de transition est une règle exprimant une contrainte sur une de ces relations. Une seule expression pour une loi peut définir plusieurs règles de transition concernant une modification. Ceci est fait en spécifiant comme conséquent de la règle une relation définissant les transitions possibles.

Cette approche a l'avantage d'être générale et de posséder la capacité d'expression puissante du calcul de prédicats du 1er ordre. Cependant, des contraintes d'intégrité du type *différées* ne sont pas envisagées.

Les idées proposées par Nicolas et Yazdanian ont commencé à être implantées dans BDGEN : "A deductive DBMS" [NICO 82]. Ils proposent dans un premier temps l'utilisation du processus de génération pour vérifier certaines classes de contraintes :

C1 & ... & Cn --> INCONSISTENT(X).

un fait de la forme INCONSISTENT(X), où X est un identificateur de la CI, est généré si l'inconsistence définie par C1 &...& Cn se produit. (Cette classe de contraintes a été aussi proposée dans [KOWA 78]). Les contraintes sont ainsi vérifiées par un système de programmation logique. Le maintien de l'intégrité n'est pas envisagé sous cette approche.

Nicolas et Yazdanian [NICO 77] considèrent le problème de la cohérence d'un ensemble de règles de transition qui est traité du même point de vue que la cohérence logique d'un ensemble de formules bien formées. La cohérence d'un ensemble de règles de transition est assurée à tout moment en déterminant si l'ensemble de règles empêche toute opération d'un type donné sur une relation. Pour cela, il est suffisant de prouver que la formule exprimant l'interdiction de toute opération de ce type pour la relation est une conséquence logique de l'ensemble de règles. Donc, si elle est une conséquence logique c'est qu'il y a une contradiction. L'incohérence est résolue en ajoutant à l'ensemble des règles de transition la formule qui interdit l'opération sur la relation.

3.5. Approche Bernstein et al.

Bernstein et al. [BERN 80] proposent l'utilisation d'une classe restreinte d'assertions, les assertions *"two-free"*. Celles-ci portent sur deux variables n-uplets référencant chacune une relation. Des contraintes sur des attributs dans les n-uplets représentés par les variables peuvent ainsi être exprimées. Leur approche par rapport à la spécification n'est pas significative dû à leur caractéristique si restreinte. Néanmoins, l'apport à la vérification a plus d'intérêt.

Les auteurs proposent d'augmenter la base de données avec des informations redondantes dont les coûts de maintien sont faibles par rapport au gain qu'elles autorisent. Les informations ajoutées à la BD sont des agrégats qui caractérisent l'ensemble des valeurs de la BD (MIN, MAX, etc). Ainsi, les preuves de cohérence qui entraînent des calculs d'agrégats sont faites avec les agrégats stockés à la place des valeurs individuelles. Depuis, cette idée a été très utilisée dans les méthodes de vérification de CI avec des agrégats.

La méthode de vérification consiste à générer au moment de la compilation de la requête de modification, une preuve de cohérence pour une assertion donnée. La preuve dépend de l'opération de modification. Chaque preuve ainsi construite est

exécutée avant la requête de modification. Un désavantage de la méthode est le fait que la génération de la preuve dépend de l'opération de modification particulière, et la méthode doit donc considérer beaucoup de cas.

Plus tard, cette méthode a été améliorée et généralisée à une gamme de CI plus large dans [BERN 81] et [BERN 82]. Les auteurs ont enlevé la restriction posée à la spécification d'assertions. Celles-ci sont ainsi considérées comme des formules du calcul de prédicats à variable n-uplet. La stratégie de vérification proposée est la suivante :

1) simplifier les assertions lors de leurs définitions

2) les assertions simplifiées sont traduites dans des requêtes d'intérrogation optimisées lors de leur exécution.

La clé de la méthode consiste à trouver un sous-état minimal de l'état de la BD modifié par la requête tel qu'il puisse être combiné avec l'assertion originale pour produire l'assertion simplifiée. La méthode est efficace car pour la simplification de l'assertion aucun accès à la BD n'est fait. En outre, lors de l'exécution des requêtes correspondantes à l'assertion simplifiée les techniques d'optimisation de requêtes sont utilisées ainsi que l'information redondante pour les agrégats.

Le maintien de l'intégrité sous cette approche n'est pas envisagé.

3.6. Approche Date

C. J. Date dans [DATE 81] propose un langage déclaratif basé sur le langage UDL ("Unified Database Language"). Son extension permet d'exprimer entre autres les contraintes référentielles dans le modèle relationnel. Une contrainte référentielle est produite lorsqu'une relation fait référence à une autre. Par exemple, la relation EMPLOYE d'attributs N-sec-soc, Ndep, Age et Sexe fait référence à la relation DEPARTEMENT via l'attribut Ndep.

Une règle d'intégrité peut contenir :

une pré-condition : permet d'indiquer quand la contrainte sera vérifiée : à la fin de la transaction, avant ou après une opération de modification de données.

la contrainte elle-même : elle est spécifiée par un prédicat.

en cas de violation : c'est une unité exécutée lors de la violation de la contrainte.

Une clause permet la spécification de contraintes référentielles dans la partie pré-condition d'une règle d'intégrité. Des mises à jour ou des éliminations en cascade de relations référencées, des éliminations de n-uplets référencés, etc peuvent être exprimés.

Les algorithmes de vérification et de maintien de CI sous cette approche n'ont pas été donnés par l'auteur. Il soutient qu'un système d'intégrité sémantique doit disposer d'une représentation interne pour les contraintes référentielles et il propose comme telle un "graphe de référence" où chaque référence est représentée par deux relations contenant des informations sur les relations référencées. Lors de la vérification des contraintes référentielles le graphe de référence est consulté.

Sous cette approche la mise en oeuvre d'un mécanisme du maintien de l'intégrité est tout à fait possible grâce à la capacité d'expression du langage de spécification de CIs proposé.

3.7. Approche Henschen et al.

La spécification sous cette approche [HENS 83] est similaire à celle de Nicolas [NICO 78]. Une CI est une formule bien formée du calcul de prédicats du 1er ordre. La méthode de vérification proposée par Henschen et al. utilise un démonstrateur de thèorèmes pour générer des preuves qui détermineront la validité d'une contrainte. La méthode générale consiste à :

1) affirmer la contrainte sur l'état actuel de la BD : la contrainte est considérée sous la forme clausale de la logique,[1]

2) exprimer le nouvel état en fonction de l'actuel et de la **forme générale** de la requête de modification : ce pas produit un ensemble de clauses à partir des axiomes de transition pour chaque requête générale de modification

3) nier la contrainte sous ce nouvel état : des clauses correspondantes à la négation de la contrainte sous l'état nouveau sont créées par l'utilisation du symbole de prédicat RNEW

[1] Nous renvoyons le lecteur à la section 1 du chapitre 2 où il trouvera un rappel sur la terminologie est les notions utilisées dans la démonstration automatique de théorèmes.

4) essayer d'obtenir une contradiction à partir de l'ensemble de clauses ainsi obtenu.

Si le démonstrateur de théorèmes arrive à une contradiction, l'opération de modification ne violera certainement pas la contrainte. Dans le cas contraire, une stratégie est appliquée par le démonstrateur de façon à générer des preuves qui complétéron les réfutations si elles sont vérifiées sur l'état actuel de la BD. Il est suffisant de compléter une réfutation pour valider la contrainte.
La méthode est très intéressante mais les auteurs n'ont pas pu démontrer sa complétude : si l'une des preuves générées réussit, elle est suffisante pour garantir la validité de la contrainte sous le nouvel état mais si toutes les preuves générées échouent on ne peut pas conclure que la contrainte est violée.
Le maintien de l'intégrité n'a pas été envisagé.

Conclusion :

Nous avons montré quelques approches parmi les plus connues dans le traitement en de CIs en général. Jusqu'ici, en ce qui concerne la spécification de CIs, une grand partie des travaux ont en commun les caractéristiques suivantes :

- Le SGBD sous-jacent est le relationnel.
- Les CI sont des assertions exprimées dans un langage d'intérrogation relationnel lequel est étendu à cette fin ou considéré comme inclus dans un autre langage de spécification de CI.
- La capacité d'expression de CIs par le langage de spécification est plus au moins augmentée par l'insertion dans les langages de clauses tels que : NEW-OLD, IMMEDIATE-DIFFERE, ONE-TO-ONE (dépendance fonctionnelle), sub-set-of (spécialisation), etc.

Des travaux plus récents [HENS 83], [WEBE 83], [SIMO 84] proposent des assertions soit en tant que formules du calcul de prédicats du 1^{er} ordre soit en tant que formules du calcul relationnel à variable n-uplet.

En résumant les moyens exploités pour augmenter l'efficacité et réduire le coût de la vérification de CIs, nous constatons que ceux-ci reposent sur les points suivants :

- simplification d'assertions lors de leurs définitions.

- génération de preuves en fonction de la requête de modification de façon à ce que l'assertion à prouver soit la plus selective possible, ceci lors de la compilation de la requête.

- maintenance d'information redondante pour les calculs des agrégats (MIN, MAX, AVERAGE, COUNT, etc). Lors de la définition d'une assertion portant sur des agrégats, des informations spécifiques à cet agrégat sont stockées puis modifiées à chaque requête de modification les concernant. En principe, cette information doit être facile à accèder et à maintenir.

- éviter la production d'algorithmes spéciaux de vérification et de maintien par l'exploitation d'un système logique de programmation fournissant un moteur d'inférence qui substitue ces algorithmes.

- tirer profit du fait que la CI est validée sous l'état actuel.

CHAPITRE 2

LOGIQUE ET BASES DE DONNEES

CHAPITRE 2

LOGIQUE ET BASES DE DONNEES

Nous présentons dans ce chapitre, tout d'abord les définitions des termes logiques nécessaires à la compréhension du domaine Logique et Bases de Données [GALL 79]. Ensuite, nous faisons un aperçu bref des points les plus souvents traités dans la littérature de ce domaine. Enfin, et suite à la présentation des approches existantes pour formaliser une BD dans le cadre de la logique, nous présentons notre point de vue sur ce domaine en particulier.

1. Quelques définitions de termes logiques

Un **système formel** comprend des symboles, des mots ou termes, des règles de formation de formules ou expressions bien formées (f.b.f), des axiomes, des théorèmes et des règles de déduction. Les systèmes formels de la logique sont construits dans le but de donner aux théories mathématiques, formulées par le système, un degré important de précision et de rigueur.[1]

Les **axiomes** sont des f.b.f acceptées sans démonstration.

Les **théorèmes** sont des f.b.f démontrables à partir des axiomes en utilisant les **règles de déduction**.

Une **théorie** est un ensemble de f.b.f selon un système formel qui explique un domaine donné.

Le **calcul des prédicats du 1er ordre** est une théorie du 1er ordre qui ne contient pas d'axiomes propres, tous ses axiomes sont logiques.
L'ensemble des symboles du calcul des prédicats du 1er ordre est le suivant :

- les symboles de ponctuation : ( ) ,
- les connecteurs logiques : $\sim$ $\wedge$ $\vee$ $\Rightarrow$ et $\equiv$ ou $\Leftrightarrow$
- les symboles de fonctions : f, g, ..

[1] Les systèmes formels ou langages sont appelés parfois *calculus* (calcul des propositions, calcul des prédicats).

- les symboles de prédicats : p, q, ...
- les quantificateurs : $\forall$ (universel) $\exists$ (existentiel)
- les symboles de variables : x, y, z ,...

Une **constante** est un cas particulier de fonction d'arité zero. On les note généralement a, b, c, ...

Un **terme** est :
- une constante ou une variable
- si f est un symbole de fonction d'arité n et t_1, t_2,....,t_n sont des termes, alors f(t_1, t_2,....,t_n) est un terme.

Si t_1, t_2,....,t_n sont des termes et P un symbole de prédicat d'arité n alors P(t_1,t_2,...,t_n) est une **formule atomique** ou **atome**. t_1,...,t_n sont les arguments du prédicat P. Un prédicat d'arité zero ou **proposition** est aussi une formule atomique ou atome (comme V (vrai) et F (faux)).

Les **règles de formation** de formules bien formées (f.b.f) sous la forme standard sont les suivantes :

(1) Toute formule atomique est une f.b.f
(2) Si A et B sont des f.b.f est x est une variable alors les expressions suivantes sont aussi des f.b.f :

$$\sim A \qquad A \Rightarrow B \qquad (\forall x)A \qquad (\exists x)A \qquad A \wedge B \qquad A \vee B \qquad A \equiv B$$

Les **règles de déduction ou d'inférence**[1] du calcul des prédicats du 1er ordre sont :

Modus Ponens : De P et P $\Rightarrow$ Q inférer Q
Généralisation : De P inférer $(\forall x)$ P

Une **théorie du 1er ordre** est une théorie basée sur le calcul des prédicats du 1er ordre. Leurs axiomes sont classés en : **axiomes logiques** (ceux du système logique) et **axiomes non-logiques** ou propres (ceux de la théorie proprement dite).

Une f.b.f dans laquelle toutes les variables sont quantifiées (universelle ou existentiellement) est dite **fermée**. Si elle n'est pas fermée, elle est **ouverte**. Les formules fermées vraies sont parfois appelées **assertions**.

---

[1] Nous n'abordons pas les théorèmes et les axiomes logiques du calcul des prédicats du 1er ordre. Le lecteur intéressé peut consulter [CHEN 74].

La **fermeture universelle (resp. existentielle)** d'une f.b.f W est une autre formule obtenue à partir de W en la préfixant avec des quantificateurs universels (resp. existentiels) portant sur les variables libres de la formule. Par exemple, la fermeture existentielle de la formule P(X) est $(\exists X)P(X)$. Si W est une formule fermée (elle n'a pas de variables libres) ses fermetures existentielles et universelles coïncident avec W.

Une f.b.f **libre de fonctions** est une formule dans laquelle il n'y a pas de symboles de fonction (à l'exception des constantes).

Un **littéral** est une formule atomique ou la négation d'une formule atomique.

Une **clause** est une disjonction d'un ou plusieurs littéraux.

Une f.b.f est sous **forme clausale** si elle a la forme suivante :

$$\forall x_{i1} \ldots . \forall x_{im} [C_1 \wedge C_2 \wedge \ldots \wedge C_n]$$

où chaque C_i $(1 \leq i \leq n)$ est une clause et $x_{i1},\ldots,x_{im}$ sont toutes les variables apparaissant dans les C_i.

Il existe une méthode de transformation d'une f.b.f W standard sous forme clausale. Pour des f.b.f fermées la méthode consiste à :

1) Eliminer les quantificateurs rédondants dans W. C'est à dire, éliminer tout quantificateur $\forall Xi$ ou $\exists Xi$ dont le champ d'application ne contient pas la variable Xi.
2) Déterminer la portée de chaque variable quantifiée et renommer ces variables de manière à ce que chaque quantificateur de la formule porte sur une variable différente.
3) Eliminer tous les connecteurs différents de : $\vee$, $\wedge$ et $\sim$. L'élimination est faite en accord avec les règles d'équivalence pour les connecteurs logiques.
4) Déplacer les $\sim$ à l'intérieur de la formule : $\sim(\forall X)P(X)$ par $(\exists X)\sim P(X)$; $\sim(\exists X)P(X)$ par $(\forall X)\sim P(X)$; $\sim(A \vee B)$ par $(\sim A \wedge \sim B)$; $\sim(A \wedge B)$ par $(\sim A \vee \sim B)$ et $\sim\sim A$ par A.
5) Déplacer vers l'intérieur les quantificateurs selon leurs champs d'application. Ex. $(\forall X)[P(X) \vee Q(a)]$ par $(\forall X)P(X) \vee Q(a)$.
6) Eliminer les quantificateurs existentiels en substituant les variables existentielles par des fonctions de Skolem. Une fonction de Skolem a autant d'arguments que des variables universelles quantifient la formule. Ainsi par exemple, $(\forall X)(\exists Y)P(X,Y)$ est remplacé par $(\forall X)P(X,f(X))$ où la fonction f est une fonction de Skolem.

7) Déplacer tous les quantificateurs universels au début de la formule.
8) Appliquer la distributivité du connecteur $\wedge$ par rapport à $\vee$. Remplacer (A $\wedge$ B) $\vee$ C par (A $\vee$ C) $\wedge$ (B $\vee$ C).

Il existe deux façons d'aborder le problème de la détermination de la *validité* d'une f.b.f : approche *sémantique* et approche *syntaxique*.

1.1. Approche sémantique

Sous cette approche, l'objectif est d'assigner des valeurs de vérité (vrai ou faux) aux formules atomiques et aux f.b.f par l'utilisation de tables de vérité pour les connecteurs logiques. Cette approche, parfois appelée **théorie des modèles**, s'intéresse à la *signification* et à la **validité** des expressions bien formées.

Tout d'abord, on assigne une signification en construisant une interprétation de la f.b.f. Une **interprétation** d'une f.b.f libre de fonctions est un triplet $<D,C,P>$. D est un ensemble de valeurs ou constantes appelé **domaine de l'interprétation** ou **univers du discours**. C et P sont des applications qui assignent à chaque symbole variable et à chaque symbole de prédicat à n arguments respectivement des constantes dans D ou des relations n-aires sur le domaine D.

Toute f.b.f ainsi interprétée peut être évaluée à "vrai" ou "faux" par l'utilisation de tables de vérité pour les connecteurs logiques apparaissant dans la formule.

Deux f.b.f W1 et W2 sont **équivalentes** si elles ont la même valeur de vérité pour toute interprétation.

Une f.b.f W est **valide** si elle s'évalue à "vrai" pour toute interprétation.

Une f.b.f W est **invalide** si et seulement si il existe au moins une interprétation pour laquelle la formule W est fausse.

Une f.b.f W est **insatisfaisable** si elle s'évalue à "faux" pour toute interprétation. Donc, une f.b.f W est valide si et seulement si la négation $\sim$W est insatisfaisable.

Une f.b.f W est **satisfaisable** s'il existe au moins une interprétation pour laquelle la formule s'évalue à "vrai". Cette interprétation est appelée un **modèle** de la formule W.

Un **modèle** d'un ensemble S de f.b.f est une interprétation sous laquelle toutes les formules dans S s'évaluent à "vrai".

1.2. Approche syntaxique

Dans le cas syntaxique, parfois appelé **théorie de la preuve**, on s'intéresse à la **preuve** des f.b.f à partir des axiomes et d'autres théorèmes dans la théorie. Une preuve consiste ainsi en une série de f.b.f obtenues par l'application des règles d'inférence aux axiomes et aux théorèmes et dont la dernière est la f.b.f à prouver. Les f.b.f ainsi prouvées peuvent être ajoutées à la théorie en tant que théorèmes.

1.2.1. Méthode de résolution

La forme clausale d'une f.b.f est particulièrement importante dans la démonstration automatique de théorèmes, notamment dans la **méthode de résolution** basée sur une règle d'inférence, la résolution. D'après cette méthode pour démontrer le théorème $A \wedge B \Rightarrow T$, il suffit de montrer l'insatisfaisabilité de $A \wedge B \wedge \sim T$. Cette dernière formule est mise sous forme clausale. Si elle est insatisfaisable,[1] l'algorithme de la méthode le détecte (donc la formule originale est un théorème).

D'une façon générale,[2] prouver l'insatisfaisabilité d'un ensemble S de clauses consiste à appliquer répétitivement la règle de résolution. Chaque pas d'application de la règle revient à obtenir une clause résolvante C3 à partir de deux clauses C1 et C2 de S. La résolvante C3 ainsi obtenue est ajoutée à l'ensemble S. Si l'ensemble de clauses est insatisfaisable, la clause vide est obtenue à un moment donné comme résolvante de clauses contradictoires C et $\sim$C. Si l'ensemble de clauses est satisfaisable les ensembles successifs deviennent stables.

Une caractéristique importante de la méthode de mise sous forme clausale est de préserver l'insatisfaisabilité de la formule. Cependant, l'équivalence de la formule n'est pas maintenue lorsque la formule sous forme standard contient des quantificateurs existentiels qui entraînent l'introduction des fonctions de Skolem dans la forme clausale. Ce résultat connu dans la logique comme le *théorème de*

---

1 Dans des termes syntaxiques, une f.b.f est insatisfaisable si la procédure de preuve de la formule détecte une contradiction.

2 Nous renvoyons le lecteur à [NILS 79], [CAFE 82] pour plus de détails sur la méthode de résolution.

Skolem permet de résoudre le problème de la détection d'une contradiction à l'intérieur d'un ensemble de f.b.f sous forme standard. Les formules sont mises sous la forme clausale puis on détermine si l'ensemble de clauses ainsi obtenu est insatisfaisable.

2. Traitement logique des principaux problèmes dans les BDs

L'utilisation de la logique en tant que cadre conceptuel pour l'étude de bases de données permet, par une analyse rigoureuse, de préciser la plupart des problèmes qui se présentent dans ce domaine et de leur proposer des solutions. En particulier, le calcul des prédicats du 1er ordre a été utilisé dans des domaines spécifiques tels que : l'interrogation de bases de données, la spécification et vérification de contraintes d'intégrité, la spécification et manipulation d'informations incomplètes, les bases de données déductives, etc.

2.1. L'interrogation de bases de données

Des langages issus de la logique sont proposés pour la formulation de requêtes d'interrogation de données. Le calcul des prédicats sous leur forme standard a été utilisé par Nicolas et Gallaire dans [NICO 78], la logique des clauses de Horn par Kowalski [KOWA 78], Kellogs et al. [KELL 78] et par Minker [MINK 78]. Dans DEDUCE-2 [CHAN 78], un langage logique sous forme symbolique a été étendu pour permettre l'expression explicite de quantificateurs numériques.[1] Reiter [REIT 78] considère une logique typée pour les requêtes. Toute variable apparaissant dans une requête doit satisfaire une propriété donnée définie par un type.

Deux approches sont en général prises en compte pour l'évaluation des requêtes, une approche compilée et l'autre interprétée. Sous l'approche compilée, le langage logique est situé dans un environnement relationnel sous lequel se trouvent stockés les données sous forme de relations. Ainsi, une requête dans le langage d'interrogation est traduite dans une requête portant sur des relations et sur des opérateurs de l'algèbre relationnelle. L'évaluation de la requête relève du SGBD relationnel. Des techniques de démonstration de théorèmes ont été utilisées pour la

---

[1] Un quantificateur numérique est de la forme "∃ op n". Où op est un des opérateurs de comparaison ($\leq$, $<$, $\geq$, $>$, $=$) et n est un entier. Par exemple, ∃=3Y signifie qu'il existent exactement 3 valeurs pour la variable Y.

transformation de la requête initiale [REIT 78], [KELL 79] et pour la vérification de la cohérence de la requête avec les contraintes d'intégrité [CHAN 80]. Sous l'autre approche (interprétée) un démonstrateur de théorèmes sous forme de clauses de Horn, étendu pour permettre l'utilisation de la négation par l'échec, est utilisé pour l'évaluation de la requête [CLAR 79].[1] Cette dernière approche est la moins proposée car elle nécessite des stratégies de résolution appropriées à l'évaluation des requêtes, en particulier pour les requêtes ouvertes,[2] dans lesquelles un grand nombre de données peuvent intervenir.

2.2. Spécification et vérification de contraintes d'intégrité

Un langage logique d'interrogation de données peut également servir à spécifier des lois générales sur ces données. Parmi ces lois, certaines peuvent servir aussi bien pour le contrôle de l'intégrité que pour la dérivation des données implicites en fonction des données explicitement stockées dans la BD. Certains auteurs les considèrent uniquement pour le contrôle d'intégrité $[NICO\ 78]_1$, pouvant ainsi générer des données à stocker explicitement dans la BD. D'autres les considèrent uniquement en tant que règles de dérivation se dédiant ainsi fortement à répondre à des requêtes déductives [CLAR 78]. Enfin une autre vue, plus générale, consiste à les utiliser dans ces deux sens [KOWA 78], [MINK 78], [CHAN 78]. Une partie des lois est choisie pour être utilisée pour le contrôle de l'intégrité, l'autre partie est choisie pour être interprétée en tant que règles de dérivation. Le concept de règle de dérivation correspond ainsi à celui de vue dans le modèle relationnel augmenté avec la capacité de récursivité.

La vérification de contraintes d'intégrité sous le point de vue de la logique a été traitée en détails dans $[NICO\ 78]_1$. Leur approche se base sur la sémantique de l'implication logique à l'intérieur d'une formule sous forme clausale. Cela leur permet, étant donné une opération sur la BD, de déterminer quelles contraintes (sous forme de clause) peuvent être mises en question par la dite opération. Une approche différente est celle de Kowalski [KOWA 78] dans laquelle la vérification de CIs est indépendante des opérations effectuées sur la BD. Elle est effectuée en déclenchant périodiquement la preuve, par un démonstrateur de théorèmes, du

---

1 Si le démonstrateur échoue pour démontrer P alors ~P est considéré comme vrai.

2 en opposition aux requêtes fermées. Une requête fermée admet comme seule réponse oui ou non.

théorème : $\vdash$ INCONSISTENCE(X). Si, lors de la vérification, la BD se trouve dans un état incohérent, le théorème peut être prouvé et la variable X est instanciée à l'identificateur de la contrainte qui n'est pas satisfaite. Cette approche a été mise en oeuvre dans [NICO 82]$_2$.

2.3. Spécification et manipulation d'informations incomplètes

Le terme "information incomplète" fait référence à la représentation d'information lorsqu'une partie du domaine de l'application n'est pas connue.

Du point de vue de Reiter [REIT 84] l'information incomplète peut être représentée et manipulée par la logique sur la base de concepts suivants :

a) l'interprétation de formules sous la forme standard de Skolem

b) *l'hypothèse du monde fermé* : on suppose que ce que l'on connaît (donc représenté) est vrai et tout le reste est automatiquement faux.

c) les formules disjonctives

La constante de Skolem représente dans une formule une valeur inconnue. Ainsi, l'énoncé: *"un fournisseur fournit la pièce* p_1 mais on ne sait pas exactement quel fournisseur", peut être spécifié par la formule du 1er ordre suivante :

$$(\exists X)\ [\textbf{fournit}(X,p_1)\ \wedge\ \textbf{fournisseur}(X)]$$

dont la forme standard de Skolem est :

$$\textbf{fournit}(w,p_1)\ \wedge\ \textbf{fournisseur}(w)$$

w, la constante de Skolem, est une **valeur nulle**.

L'hypothèse du monde fermé, matérialisée par les axiomes de fermeture du domaine[1] et de complétude[2] pour chaque prédicat, permet de répondre aux requêtes d'interrogation sur les données de la base de la même façon qu'en absence de valeurs nulles. En outre, on peut prouver $\vdash$ **fournit**(w,p_1) et $\vdash$ **fournisseur**(w). Pour cela les axiomes de complétude pour les prédicats **fournit**(X,Y) et **fournisseur**(X) doivent être modifiés pour comporter toutes les paires possibles (w,Y) pour le prédicat "fournit" et X=w pour le prédicat fournisseur. De même, l'axiome de

---

1 Définit toutes les constantes qui apparaissent dans la base de données.

2 Définit pour un prédicat P tous les n-uplets (X_1, ..., X_n) pour lesquels $P(X_1,\ldots,X_n)\equiv$vrai.

fermeture du domaine doit comporter la constante w.
Les formules disjonctives permettent de représenter des énoncés de la forme : *"le fournisseur* f_1 fournit la pièce p_1 ou la pièce p_2 mais on ne sait pas exactement laquelle". La formule disjonctive correspondante est :

$$\textbf{fournit}(f_1,p_1) \vee \textbf{fournit}(f_1,p_2)$$

Cet énoncé est matérialisé en rajoutant à l'axiome de complétude pour le prédicat "fournit" ces alternatives. Ainsi, l'axiome de complétude résultant est :

$$\ldots\ldots \vee [=(X,f_1) \wedge =(Y,p_1)] \vee [=(X,f_1) \wedge =(Y,p_2)] \Rightarrow \textbf{fournit}(X,Y).$$

Le point de vue de Levesque [LEVE 84] se base sur l'utilisation d'un langage logique dans lequel est contenu le calcul des prédicats du 1er ordre. Ce langage permet d'exprimer de façon différenciée *ce qui est connu* (et ce qui n'est pas connu respectivement) de *ce qui est vrai* dans un domaine donné. De cette façon, on peut expliciter quand l'hypothèse du monde fermé ne peut pas être appliquée. Par exemple, la formule ($\exists$X) [fournisseur (X) $\wedge$ ~Kfournisseur(X)] établi qu'il existe au moins un fournisseur dont on ne connait pas (~K) l'existence. L'application de l'hypothèse du monde fermé serait donc contradictoire avec cet énoncé.

Cette approche offre la possibilité de formuler, en plus des requêtes d'interrogation sur le domaine, des requêtes sur la représentation même des connaissances. Notamment : *les connaissances représentées par un prédicat donné sont-elles incomplètes ?*. L'hypothèse d'un monde fermé peut ainsi être appliquée à certains prédicats et à d'autres non.

2.4. Les bases de données déductives

Les systèmes de réponse aux questions en Intelligence Artificielle ont été les premiers à utiliser la logique pour représenter et traiter les connaissances. Des questions peuvent trouver une réponse moyennant un petit ensemble de connaissances et un mécanisme d'inférence logique. Au point de départ des sgbds déductifs se trouve l'idée d'étendre les sgbds conventionnels avec cette possibilité pour permettre le traitement d'un ensemble plus grand de connaissances, le contrôle de leur intégrité ou cohérence, le traitement de connaissances incomplètes, etc. Ce qu'il y a d'essentiel dans un sgbd déductif est la posibilité de dérivation par des axiomes de déduction de nouveaux faits à partir de faits existants explicitement dans la base de données ou d'autres faits également dérivés. Selon l'hypothèse utilisée pour le

traitement des informations négatives et le type d'axiomes de déduction acceptés par un sgbd déductif, ces sgbds ont été classés [GALL 82] en :

définis : un axiome établi l'hypothèse de négation en cas d'échec et les axiomes de déduction sont des clauses définies. Une clause définie est une clause contenant un et seulement un littéral positif.

indéfinis : un axiome établi l'hypothèse généralisée de négation en cas d'échec[1] et les axiomes de déduction sont des clauses définies et/ou indéfinies.

La justification de cette catégorisation s'appuie sur le fait que l'hypothèse de négation en cas d'échec peut conduire à des incohérences en présence de clauses indéfinies. Par exemple, si la BD contient le fait P(a) et l'axiome de déduction $P(X) \Rightarrow Q(X) \vee R(X)$, puisque P(a) est vrai $Q(a) \vee R(a)$ aussi. $\sim Q(a)$ est vrai par négation en cas d'échec car on ne peut pas démontrer Q(a). De même pour $\sim R(a)$. Donc, $\sim Q(a) \wedge \sim R(a) \equiv$ vrai est contradictoire avec $Q(a) \vee R(a) \equiv$ vrai.

3. Les approches dans le domaine : Logique et Bases de Données

L'utilisation de la logique en tant que cadre conceptuel pour l'étude et la formalisation des problèmes dans les Bases de Données a donné lieu jusqu'à présent à différentes approches pour modéliser une réalité donnée.

Une première approche, présentée en détail par Nicolas et Gallaire dans [NICO 78]$_3$ et très répandue dans les systèmes de réponse aux questions en Intelligence Artificielle, propose que la réalité soit perçue comme une théorie du 1er ordre dont les axiomes propres sont les f.b.f associées à l'information explicite (faits élémentaires et lois générales). Les lois générales sont ainsi utilisées uniquement pour dériver des informations implicites. Celles-ci, et les requêtes d'interrogation fermées, sont des théorèmes à prouver dans la théorie par des procédures syntaxiques telles que la résolution. Une requête ouverte portant sur une formule donnée $W(x_1,\ldots,x_n)$ constitue l'ensemble de preuves pour la fermeture existentielle $\exists x_1 \ldots \exists x_n\ W(x_1,\ldots,x_n)$ de la formule. L'information négative est représentée soit de façon explicite (par l'insertion de faits élémentaires négatifs) soit implicite, c'est à dire qu'une réponse sur des informations négatives est dérivée à partir des

[1] Nous renvoyons le lecteur à [MINK 82] pour une description détaillée de l'hypothèse généralisée de négation en cas d'échec.

informations positives et des lois d'unicité ou de complétude pour les prédicats. Les principaux désavantages de cette approche sont la non utilisation de certaines lois générales en tant que contraintes d'intégrité et la pauvreté du point de vue sémantique des données.

Une deuxième approche présentée de même par les auteurs consiste à percevoir la réalité par des lois générales et des faits élémentaires connus complètement. Les faits élémentaires (exclusivement positifs) constituent une interprétation, au sens logique du terme, des lois générales. L'utilisation de règles générales en règles de génération pouvant produire des redondances, elles sont utilisées comme des contraintes d'intégrité à vérifier lors de mises à jour de faits élémentaires. Un meilleur traitement sémantique de données est considéré par l'utilisation de relations unaires de façon similaire à la notion de domaine. Les requêtes d'interrogation fermées ne sont pas considérées comme des théorèmes à prouver mais comme des formules devant être évaluées moyennant des tables de vérité (tables relationnelles) dans l'interprétation de la théorie.

Une troisième approche constitue une extension de la deuxième. Chaque loi générale est utilisée, soit en tant que règle de dérivation d'information implicite soit en tant que contrainte d'intégrité. Pour les règles de dérivation deux possibilités d'utilisation sont présentées. Sous la première, une relation est définie uniquement par une loi en fonction de relations de base (faits élémentaires) à partir desquelles les n-uplets correspondants sont dérivés lors de requêtes d'interrogation. Sous l'autre possibilité, une relation peut être définie en partie par des faits élémentaires et en partie par une règle de dérivation.

Ces deux dernières approches sémantiques correspondent à la vue *modèle théorique*, en opposition à celle syntaxique de *théorie de la preuve* soutenue par Reiter [REIT 84]. Reiter affirme que la vue modèle théorique ne peut pas donner une formalisation pour les informations incomplètes étant donné qu'elle se base sur la notion d'interprétation. Un fait de la forme $P(a) \vee P(b)$ peut seulement être représenté par les trois interprétations possibles $\{P(a)\}$, $\{P(b)\}$, $\{P(a) \wedge P(b)\}$ ce qui oblige à connaître P(a) et/ou P(b).
La vue théorie de la preuve considère une base de données relationnelle comme un triplet (R, T, CI). R est le langage relationnel qui définit les symboles constants et les noms de prédicats. T est une théorie relationnelle du 1^{er} ordre dont les axiomes définissent la BD. CI est l'ensemble de f.b.f constituant les contraintes d'intégrité. Une contrainte d'intégrité $w \in CI$ est satisfaite si et seulement si $T \vdash w$. Toute requête d'interrogation est vue comme un ensemble de théorèmes à prouver. Cette

vue fournit une solution pour la représentation de l'information incomplète. Nous renvoyons le lecteur à [REIT 84] pour une description formelle et détaillée de cette approche.

4. L'approche logique de SYCSLOG

SYCSLOG fournit un modèle logique d'aide à la représentation structurée de connaissances d'une réalité donnée et qui facilite l'utilisation du calcul des prédicats du 1er ordre dans la représentation de connaissances. Parmi les formules représentant la connaissance certaines servent de base pour dériver d'autres connaissances, d'autres servent à assurer la cohérence d'une partie de ces connaissances. Dans des termes logiques formels, les premières sont des axiomes d'une théorie du 1er ordre qui permettent, par un mécanisme automatique de raisonnement, de dériver des théorèmes. Ceci correspond d'une façon générale à l'approche théorie de la preuve proposée par Reiter [REIT 84] cependant nous faisons une catégorisation différente de l'ensemble d'axiomes qui composent une théorie du 1er ordre donnée. La catégorisation d'axiomes a pour but d'organiser les connaissances pour une exploitation cohérente et productive.

Considérons que nous sommes en présence d'une réalité finie que nous voulons modéliser et que nous connaissons complètement ou partiellement. Dans cette réalité nous trouvons, bien entendu, des individus et des objets concrets ou abstraits appelés couramment "entités" dans le domaine de la modélisation conceptuelle. De même, on remarque des liens ou des relations d'une entité avec d'autres entités. Toute entité a un certain nombre de propriétés intéressantes, c'est à dire que nous voulons les enregistrer pour les consulter simplement ou pour les utiliser dans des processus un peu plus élaborés, tels que des analyses, des diagnostics, des prédictions, etc. Pour ceux-ci la connaissance dont nous disposons est parfois incomplète. Autrement dit, il se peut que nous ne connaissions pas, pour une entité donnée, certains attributs parmis ceux que nous avons choisis comme "intéressants". Par exemple, dans un milieu médical, le fait que nous ne connaissions pas si une personne a déjà eu l'hépatite pourrait servir à diagnostiquer en présence d'un certain nombre de symptômes, que : "si la personne a déjà eu l'hépatite, elle a forte chance d'avoir une infection du foie". Comme autre exemple, si un appareil de mesure météorologique est en panne en un point précis, des prévisions sont toujours possibles en utilisant des valeurs hypothétiques pour ce facteur.

Dans ce qui suit, les entités et les relations seront décrites comme dans le domaine de la logique, c'est à dire par des propositions prédicatives ou attributives (portant sur des prédicats de propriété) et par des propositions relationnelles (portant sur des prédicats de relation). La modélisation d'une réalité sous SYCSLOG est faite en considération de sa nature complète ou incomplète. Ainsi, chaque entité ou relation dans l'environnement est située explicitement dans un monde connu (monde fermé) ou dans un monde partiellement connu (monde ouvert).

4.1. Les axiomes

Les informations dans une entreprise, les connaissances sur un domaine expert ou d'une réalité donnée sont représentées, dans SYCSLOG, par un ensemble de formules bien formées, en accord avec un modèle logique. Ces formules sont classées en fonction de ce qu'elles représentent, c'est à dire soit l'information ou la connaissance même de la réalité, soit l'information sur la représentation de la réalité. Les *axiomes* sont des formules qui représentent des connaissances explicites d'une réalité que nous savons toujours vraies. Ils sont classés en :

Γ_1 : *Axiomes d'unicité de propriété*

Γ_2 : *Axiomes de structure*

Γ_3 : *Axiomes de base*

- Γ_3^+ : *Axiomes de base positifs*
- Γ_3^- : *Axiomes de base négatifs*
- $\Gamma_3^\vee$: *Axiomes de base disjonctifs*
- $\Gamma_3^?$: *Axiomes des valeurs nulles*

Γ_4 : *Axiomes de contenu*

Γ_5 : *Axiomes de déduction*

Γ_6 : *Axiomes d'un monde fermé*

Γ_7 : *Axiomes d'un monde ouvert*

4.1.1. Axiomes d'unicité de propriété (Γ_1)

Un axiome d'unicité de propriété est une f.b.f fermée et libre de fonctions. Il définit un ensemble de termes perçus comme une classe possédant uniquement une propriété p en commun. L'axiome a deux formes selon si l'on peut dénombrer les différents termes ou non. Dans ce dernier cas les termes doivent satisfaire une serie de restrictions de base (rb_1, ..., rb_q ; $q \geq 1$) qui pour un terme x sont : entier(x), atome(x) et des conditions formées avec les opérateurs de comparaison pour les entiers ($<$, $>$, $=$).

Forme 1 :

$$[p(t_1) \vee .. \vee p(t_n)] \wedge (\forall x)[x \neq t_1 \wedge ... \wedge x \neq t_n \Rightarrow \sim p(x)]$$

Cet axiome exclut de la classe déterminée par t_1, ..., t_n tout autre terme. Autrement dit, l'axiome établi l'hypothèse du monde fermé pour la classe de termes qui ont la caractéristique p.

Forme 2 :

$$(\forall x)[p(x) \Rightarrow rb_1(x) \wedge ... \wedge rb_q(x)]$$

Tout terme x qui satisfait la propriété p doit satisfaire aussi les restrictions rb_1, ..., rb_q. De même, un terme qui ne satisfait pas au moins une restriction ne satisfait pas la propriété p.
L'ensemble d'axiomes d'unicité de propriété dans la théorie est noté Γ_1.

Exemple 1

Considérons les termes : lundi, mardi,...,samedi et dimanche. Ceux-ci ont en commun la caractéristique d'être des jours de la semaine. Si cette caractéristique est nommée "jour" l'axiome d'unicité de propriété suivant peut être établi :

$$[jour(lundi) \vee ... \vee jour(dimanche)] \wedge$$
$$(\forall x)[x \neq lundi \wedge ... \wedge x \neq dimanche \Rightarrow \sim jour(x)]$$

Exemple 2

Considérons la propriété " être un salaire " groupant tous les salaires. Bien entendu, dénombrer tous les salaires connus serait très long et nous ne serions pas sûrs d'être exhaustifs. Par contre, nous pouvons dire qu'un salaire est une valeur entière comprise dans une certaine plage de valeurs et ceci est tout à fait envisageable. Donc, l'axiome d'unicité de propriété pour la propriété "salaire" est :

$$(\forall s)[\text{salaire}(s) \Rightarrow (\text{entier}(s) \wedge s \geq 1500 \wedge s \leq 50000)]$$

4.1.2. Axiomes de structure (Γ_2)

Un axiome de structure définit, pour un ensemble de termes, un ensemble de propriétés communes $p_1, \ldots, p_n$ à chacune desquelles correspond un axiome d'unicité. Chaque terme est distingué dans l'ensemble des termes par une valeur I d'une propriété id à laquelle correspond également un axiome d'unicité. Un nom p est assigné à l'ensemble de termes. Un axiome de structure est très lié à la structure physique qui ont les données explicites de l'ensemble de termes. Autrement dit, dans la BD il y aura un enregistrement de nom p dont les champs composants sont id, $p_1, \ldots, p_n$. Ainsi, un axiome de structure sert à déduire si une structure donnée $p(I,x_1,\ldots,x_n)$ satisfait les restrictions sémantiques spécifiées dans les axiomes d'unicité de propriété correspondantes. Parmi les propriétés $p_1, \ldots, p_n$ on peut permettre que certaines prennent des valeurs nulles, représentant ainsi des valeurs inconnues. Une valeur nulle est représentée de façon générale par le symbole constant ?.

Un axiome de structure peut aussi définir une relation r entre plusieurs ensembles de termes $e_1, \ldots, e_q$ déjà groupés par des axiomes de structure. Les termes de ces ensembles sont ainsi mis en relation et des propriétés communes à ces liens peuvent s'établir.

L'axiome de structure a donc les formes suivantes :

Forme 1 :

a) $\forall I\ \forall x_1,\ldots,\forall x_n\ [p(I,x_1,\ldots,x_n) \Rightarrow id(I) \wedge p_1(x_1) \wedge \ldots \wedge p_n(x_n)]$

b) $\forall I\ \forall x_1,\ldots,\forall x_n\ [p(I,x_1,\ldots,x_n) \Rightarrow$

$$id(I) \wedge p_1(x_1) \wedge \ldots \wedge (p_i(x_i) \vee p_i(?)) \wedge \ldots \wedge p_n(x_n)]$$

L'axiome a) exclut la possibilité pour les propriétés p_1, ..., p_n de prendre des valeurs nulles. L'axiome b) inclut cette possibilité pour la propriété p_i. Par définition, dans un axiome de structure la propriété id qui identifie l'ensemble de termes ne peut pas prendre des valeurs nulles.

Forme 2 :

a) $\forall I_1,\ldots,\forall I_q\ \forall x_1,\ldots \forall x_n\ \forall x_1^1 \ldots \forall x_{nq}^q$

$[r(I_1, \ldots, I_q, x_1, \ldots, x_n) \Rightarrow$

$e_1(I_1, x_1^1, \ldots, x_{n1}^1) \wedge \ldots \wedge e_q(I_q, x_1^q, \ldots, x_{nq}^q) \wedge p_1(x_1) \wedge \ldots \wedge p_n(x_n)]$

et $q \geq 2$

b) De même que la forme 1b pour les propriétés p_1, ..., p_n

Exemple 3

Considérons l'ensemble des employés ayant en commun les propriétés suivantes : avoir un numéro de sécurité sociale qui les identifie de façon distincte, avoir un salaire, un nom et une adresse. A chaque propriété est affecté un axiome d'unicité de propriété dans Γ_1. L'axiome de structure correspondant est :

$\forall$Nsec $\forall$S $\forall$N $\forall$A

[employé(Nsec,S,N,A) $\Rightarrow$

sec-soc(Nsec) $\wedge$ salaire(S) $\wedge$ nom(N) $\wedge$ (adresse(A) $\vee$ adresse(?))]

Exemple 4

Considérons une relation entre les employés et les départements d'une entreprise. La relation étant : " les employés sont affectés aux départements à partir d'une certaine date ". Nommons la relation "assignation". L'axiome de structure pour une telle relation est :

$\forall$Nsec $\forall$Ndep $\forall$Date

[assignation(Nsec, Ndep, Date) $\Rightarrow$

employé(Nsec, S, N, A) $\wedge$ département(Ndep, Chef) $\wedge$ date(Date)]

4.1.3. Axiomes de base (Γ_3)

Les axiomes de base sont les connaissances explicites pour les ensembles de termes et les relations définis par les axiomes de structure. Donc, pour un ensemble p de termes identifiés par les valeurs i_1, ..., i_q et dont les valeurs correspondantes aux propriétés p_1,..., p_n sont v_1^1,..., v_n^1,..., v_1^q, ...,v_n^q, les axiomes de base ont la forme suivante :

Axiomes de base positifs (Γ_3^+) :

$$[\ \text{vrai} \Rightarrow p(i_1, v_1^1,\ldots, v_n^1)\ ]$$

$$\vdots$$

$$[\ \text{vrai} \Rightarrow p(i_q, v_1^q,\ldots, v_n^q)\ ]$$

Axiomes de base négatifs (Γ_3^-) :

$$[\ \text{vrai} \Rightarrow \text{not}(p(i_1, v_1^1,\ldots, v_n^1))\ ]$$

$$\vdots$$

$$[\ \text{vrai} \Rightarrow \text{not}(p(i_q, v_1^q,\ldots, v_n^q))\ ]$$

Axiomes de base disjonctifs ($\Gamma_3^\vee$) :

$$[\ \text{vrai} \Rightarrow p(i', v_1',\ldots, v_n') \vee q(i'', v_1'',\ldots, v_n'')\ ]$$

Cet axiome exprime le fait que nous ne pouvons jamais établir des axiomes de base tels que not(p(i', v_1',..., v_n')) et not(q(i'', v_1'', ..., v_n'')) soient vrais simultanement.

Axiomes des valeurs nulles ($\Gamma_3^?$) :

$$[\ \text{vrai} \Rightarrow p_j(v_j^i)\ ]$$

Pour ces axiomes de base et par valeur v_j^i inconnue, un axiome définissant cette valeur est ajoutée à la théorie si et seulement si dans l'axiome de structure a été spécifié que la propriété p_j pouvait prendre des valeurs nulles. Par exemple, si la valeur v_1^1 pour la propriété p_1 dans le premier axiome de base positif est inconnue alors nous ajoutons l'axiome :

$$[\ \text{vrai} \Rightarrow p_1(v_1{}^1)\ ]$$

Bien entendu, l'élimination de cet axiome de la théorie s'impose lorsque la valeur $v_1{}^1$ pour la propriété p_1 dans l'axiome de base correspondant devient connue. De la même façon, on établit des axiomes de base pour les relations.

Exemple 5

Les axiomes de base suivants peuvent être formulés pour l'exemple précédent :

[vrai ⇒ employe(1000000, 3500, 'Jean Dupond', 'Paris')]
[vrai ⇒ employe(1010000, 4500, 'Marie Martinez', '?1')]
[vrai ⇒ adresse('?1')]
[vrai ⇒ not(employé(1050000, 6000, 'Jacques R', 'Lyon'))]
[vrai ⇒ employé(1060000, 5500, 'Pierre Lefevre', 'Lyon')] ∨
employé(1060000, 5500, 'Pierre Lefevre', 'Grenoble')]

L'avant dernier axiome pourrait signifier que l'employé de numéro 1050000 et de nom Jacques R. ne travaille plus dans l'entreprise. Le dernier axiome signifierait que l'on ne connaît pas pour l'instant l'adresse de l'employé 1060000 mais qu'il est certain qu'il habite ou à Lyon ou à Grenoble.

4.1.4. Axiomes de contenu (Γ_4)

Les axiomes de contenu permettent de définir des ensembles de termes tel que des axiomes de structure mais d'un point de vue externe-interne ou logique-physique. A un ensemble p de termes identifiés par la propriété id et ayant les propriétés en commun p_1, ..., p_n lui correspondent les axiomes de contenu suivants :

$$\forall I\ \forall x_1,\ldots,\forall x_n\ [\ p(I,x_1,\ldots,x_n) \Rightarrow p_i(p(I), x_i)\ ],\ i=1,\ldots,n$$

Ces axiomes servent, à partir des connaissances stockées explicitement sous la forme $p(I,x_1, \ldots, x_n)$, à dériver des connaissances de la forme : " le terme identifié par la valeur I a la valeur x_i pour la propriété p_i". Autrement dit, ces axiomes permettent la dérivation de connaissances indépendamment de la structure de stockage associée par l'axiome de structure.

On établit de la même façon des axiomes de contenu pour des connaissances stockées physiquement sous la forme not(p(I,x_1,...,x_n)). Ainsi $\Gamma_4 = \Gamma_4^+ \cup \Gamma_4^-$.

Exemple 6

Les axiomes de contenu pour l'exemple 3 précédent sont :

$\forall$Nsec $\forall$S $\forall$N $\forall$A [employé(Nsec, S, N, A) $\Rightarrow$ salaire-employé(Nsec, S)]
$\forall$Nsec $\forall$S $\forall$N $\forall$A [employé(Nsec, S, N, A) $\Rightarrow$ nom-employé(Nsec, N)]
$\forall$Nsec $\forall$S $\forall$N $\forall$A [employé(Nsec, S, N, A) $\Rightarrow$ adresse-employé(Nsec,A)]

4.1.5. Axiomes de déduction (Γ_5)

Un axiome de déduction définit ou un ensemble de termes ou une relation p dont les connaissances peuvent être déterminées de façon implicite à partir de connaissances explicites et/ou implicites pour d'autres ensembles de termes et relations (q_{ij}). L'antécédant de l'axiome est une disjonction de conjonctions de q_{ij}. Chaque disjonction est une alternative possible de dérivation des connaissances pour le prédicat p.

$$\forall x_1 \ldots \forall x_n \; \forall \ldots \{[\ldots. \vee (q_{i1}(\ldots..) \wedge \ldots. \wedge q_{iin}(\ldots..)) \vee \ldots.] \Rightarrow p(x_1,\ldots,x_n) \}$$

et $1 \leq i \leq m$, $1 \leq i_n \leq n$

Exemple 7

Supposons que la relation père a été définie par un axiome de structure et que nous voulons définir la relation ancêtre à partir de la relation père. L'axiome de déduction nécessaire est :

$\forall$F $\forall$P $\forall$X [père(P,F) $\vee$ (père(P,X) $\wedge$ ancêtre(X,F)) $\Rightarrow$ ancêtre(P,F)]

4.1.6. Axiomes d'un monde fermé (Γ_6)

Un axiome d'un monde fermé est formulé pour un ensemble de termes ou pour une relation dont les connaissances sont supposées connues soit explicitement par des axiomes de base positifs soit implicitement par des axiomes de déduction.

L'axiome a les formes suivantes :

Forme 1

$\forall I\ \forall x_1,\ldots,\forall x_n$

$\{ [[vrai \Rightarrow p(I,x_1,\ldots,x_n)] \Rightarrow p(I,x_1,\ldots,x_n)] \wedge$

$[not[vrai \Rightarrow p(I,x_1,\ldots,x_n)] \Rightarrow not(p(I,x_1,\ldots,x_n))] \}$

L'axiome établi que la connaissance $p(I,x_1,\ldots,x_n)$ est déduite à partir de l'axiome de base positif $[vrai \Rightarrow p(I,x_1,\ldots,x_n)]$. L'absence de ce dernier permet d'en déduire $not(p(I,x_1,\ldots,x_n))$.

Forme 2

$\forall x_1,\ldots,\forall x_n$

$\{ not[[\ldots. \vee (q_{i1}(\ldots..) \wedge \ldots. \wedge q_{iin}(\ldots)) \vee \ldots] \Rightarrow p(x_1,\ldots,x_n)]$

$\Rightarrow not(p(x_1, .., x_n)) \}$

L'axiome établi que la connaissance $not(p(x_1,\ldots,x_n))$ est déduite si l'on ne peut pas déduire $p(x_1,\ldots,x_n)$ par l'axiome de déduction correspondant.

4.1.7. Axiomes d'un monde ouvert (Γ_7)

Un axiome d'un monde ouvert est formulé pour un ensemble de termes ou pour une relation dont les connaissances explicites sont supposées connues mais exprimées par des axiomes de base positifs et négatifs. L'axiome a la forme suivante :

$\forall I\ \forall x_1 \ldots \forall x_n$

$\{ [[vrai \Rightarrow p(I,x_1,\ldots,x_n)] \Rightarrow p(I,x_1,\ldots,x_n)] \wedge$

$[[vrai \Rightarrow not(p(I,x_1,\ldots,x_n))] \Rightarrow not\ (p(I,x_1,\ldots,x_n))] \}$

L'axiome établit que les connaissances $p(I,x_1,\ldots,x_n)$ et leur négation sont déduites uniquement à partir des axiomes de base respectivement, positifs et négatifs.

4.1.8. Lois d'intégrité

Les *lois d'intégrité* sont des formules qui représentent les lois générales satisfaites implicitement par les connaissances explicitées au moyen des axiomes de base. En conséquence, elles sont du point de vue logique des formules démontrables à partir de l'ensemble d'axiomes.

Une lois d'intégrité W est une f.b.f fermée, libre de fonctions et cohérente avec l'ensemble d'axiomes $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_7$. Dans ce sens si $\Gamma \vdash W$ alors $\sim(\Gamma \vdash \sim W)$.
Toute lois d'intégrité cohérente avec la théorie Γ peut être ajoutée à la théorie en tant que théorème.

4.1.9. Une théorie du 1er ordre non-contradictoire

La construction de la théorie du 1er ordre Γ doit se faire de façon à assurer sa non-contradiction. Autrement dit, on ne peut pas démontrer à la fois qu'une formule est un théorème et que la négation de cette formule est encore un théorème. Ainsi, l'adjonction ou l'élimination d'un nouvel axiome A dans Γ ne doit pas invalider les théorèmes existants dans la théorie (lois d'intégrité). Donc, pour tout axiome A et pour tout théorème T :

$$\Gamma \vdash T \Leftrightarrow \Gamma \cup A \vdash T$$

Parallèlement, l'adjonction d'une loi d'intégrité T ne doit pas rendre la théorie contradictoire. De cette manière, un théorème T est ajouté si et seulement s'il est démontrable à partir de la théorie :

$$\Gamma \vdash T$$

4.2. Les théorèmes

Les théorèmes à prouver à partir d'une théorie ainsi construite, mais qui ne sont pas rajoutés à la théorie, sont des f.b.f libres de fonctions et classés en deux catégories selon que la preuve du théorème fait ou non intervenir les axiomes correspondants à un monde fermé (Γ_6) ou à un monde ouvert (Γ_7).

4.2.1. Les théorèmes dérivés d'un monde fermé

Un théorème T_{MF} est dérivé d'un monde fermé si sa preuve n'est possible que par l'utilisation des axiomes de la théorie Γ sans compter les axiomes d'un monde ouvert. Autrement dit :

$$(\Gamma - \Gamma_7) \vdash T_{MF}$$

Selon les axiomes qui apparaissent dans la preuve d'un théorème, celui-ci est considéré en tant que :

- **requête de vérification de définition :**

si le théorème à prouver est une formule fermée permettant de vérifier des définitions par les axiomes d'unicité de propriété Γ_1 et les axiomes de structure. C'est à dire :

$$\Gamma_1 \cup \Gamma_2 \cup \Gamma_3^? \vdash T_{MF}$$

Il est aussi vrai que :

$$\Gamma_3^+ \vdash T_{MF} \Rightarrow \Gamma_1 \cup \Gamma_2 \vdash T_{MF}$$

- **requête d'interrogation positive :**

si dans la preuve du théorème participent les axiomes de base positifs (Γ_3^+) et/ou de contenu (Γ_4). Autrement dit :

(1) $(\Gamma_1 \cup \Gamma_2) \cup \Gamma_6 \cup \Gamma_3^+ \vdash T_{MF}$

(2) $(\Gamma_1 \cup \Gamma_2) \cup \Gamma_6 \cup \Gamma_3^+ \cup \Gamma_4 \vdash T_{MF}$

Une requête d'interrogation positive peut être fermée ou ouverte selon que le théorème est respectivement une formule fermée ou ouverte. La preuve d'une formule ouverte $p(x_1,\ldots,x_n)$, $x_1, \ldots, x_n$ étant des variables libres, est équivalente par définition à la preuve de sa fermeture universelle ou de façon équivalente à l'ensemble des preuves de sa fermeture existentielle.

- **requête déductive positive :**

si le théorème est prouvé par les axiomes de déduction (Γ_5) et les axiomes de base positifs ou de contenu. C'est à dire :

$$(\Gamma_1 \cup \Gamma_2) \cup \Gamma_6 \cup \Gamma_3^+ \cup \Gamma_4^+ \cup \Gamma_5 \vdash T_{MF}$$

où not($\Gamma_3^+ \neq \emptyset \wedge \Gamma_4^+ \neq \emptyset$).

4.2.2. Les théorèmes dérivés d'un monde ouvert

Un théorème T_{MO} est dérivé d'un monde ouvert si dans sa preuve participent les axiomes d'un monde ouvert. D'une autre manière :

$$\Gamma \vdash T_{MO}$$

De même, selon les axiomes utilisés dans la preuve d'un théorème T_{MO} il est considéré comme :

- **requête d'interrogation négative :**

si le théorème est prouvé par les axiomes de base (Γ_3) et/ou de contenu (Γ_4) et les axiomes d'un monde ouvert. Ainsi :

$$(\Gamma_1 \cup \Gamma_2) \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_7 \vdash T_{MO}$$

Les requêtes d'interrogation négatives sont fermées ou ouvertes selon que le théorème est respectivement une formule fermée ou ouverte.

- **requête de déduction négative :**

si dans la preuve du théorème participent les axiomes d'un monde ouvert (Γ_7) et les axiomes de déduction (Γ_5). Autrement dit :

$$(\Gamma_1 \cup \Gamma_2) \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5 \cup \Gamma_6 \cup \Gamma_7 \vdash T_{MO}$$

Conclusion :

Dans ce chapitre, nous avons commencé par présenter un aperçu des points les plus couramment traités dans le domaine Logique et Bases de Données. Puis, nous avons décrit brièvement les approches les plus importantes pour concevoir une base de données sous une optique logique. Ceci nous a servi de cadre introductif à l'approche que nous proposons dans ce travail.

En résumant, la modélisation d'un environnement sous l'approche SYC-SLOG passe par la construction d'une théorie du 1er ordre particulière ($\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_7$) dont les axiomes correspondent aux méta-axiomes présentés. Cette

catégorisation d'axiomes permet d'organiser de façon cohérente les connaissances de l'environnement. Chaque catégorie a ainsi une fonction bien précise pendant la période d'exploitation des connaissances. Cette catégorisation a également permis de définir les requêtes possibles d'interrogation sur l'ensemble de connaissances, par la spécification des ensembles d'axiomes nécessaires à la preuve de certains théorèmes de la théorie.

CHAPITRE 3

SYCSLOG

SYSTEME LOGIQUE D'INTEGRITE SEMANTIQUE

CHAPITRE 3

SYCSLOG
SYSTEME LOGIQUE D'INTEGRITE SEMANTIQUE

Le système SYCSLOG est conçu dans le but de son intégration à un SGBD existant (basé sur le modèle relationel, entity-relationship, sémantique, etc) pour lui doter des capacités déductives qui permettent d'assurer principalement l'intégrité sémantique de données ainsi que de leur définitions. Dans un tel contexte, étant donné la capacité représentative de la logique des prédicats et la disponibilité de langages de représentation de connaissances basés sur le calcul des prédicats du 1^{er} ordre, notre système d'intégrité sémantique se sert du langage Prolog et de la logique des prédicats comme outils de modélisation.

1. Fonctionnalités générales et architecture

La figure 1 montre le schéma architectural du système SYCSLOG dont les principales composantes sont :

BD

c'est la base de données de l'application. Lorsque l'on définit pour l'application des prédicats à évaluer sous un monde ouvert, elle est considérée comme séparée en deux bases distinctes : BD^+ contenant les faits explicites positifs et BD^- contenant les faits explicites négatifs.

SGBD

c'est le système de gestion de la BD. Celui-ci, *indépendant* de SYCSLOG, peut être basé sur n'importe quel modèle conceptuel de données. En effet, SYCSLOG est indépendant de tout modèle conceptuel.

BC

cette base de connaissances contient les connaissances générales liées à la BD de l'application, c'est à dire les énoncés de définition de données, les contraintes d'intégrité, les règles de déduction, etc.

Echant

c'est un échantillon représentatif de la BD. Il sert à vérifier la cohérence des énoncés dans la BC.

BMC

la base de méta-connaissances META-LOG est indépendante de toute application. Elle contient les méta-règles qui définissent le modèle logique de SYCSLOG.

Traitement des connaissances

cette partie assure l'ensemble des opérations nécessaires au maintien global de l'intégrité et à l'exploitation générale des connaissances.

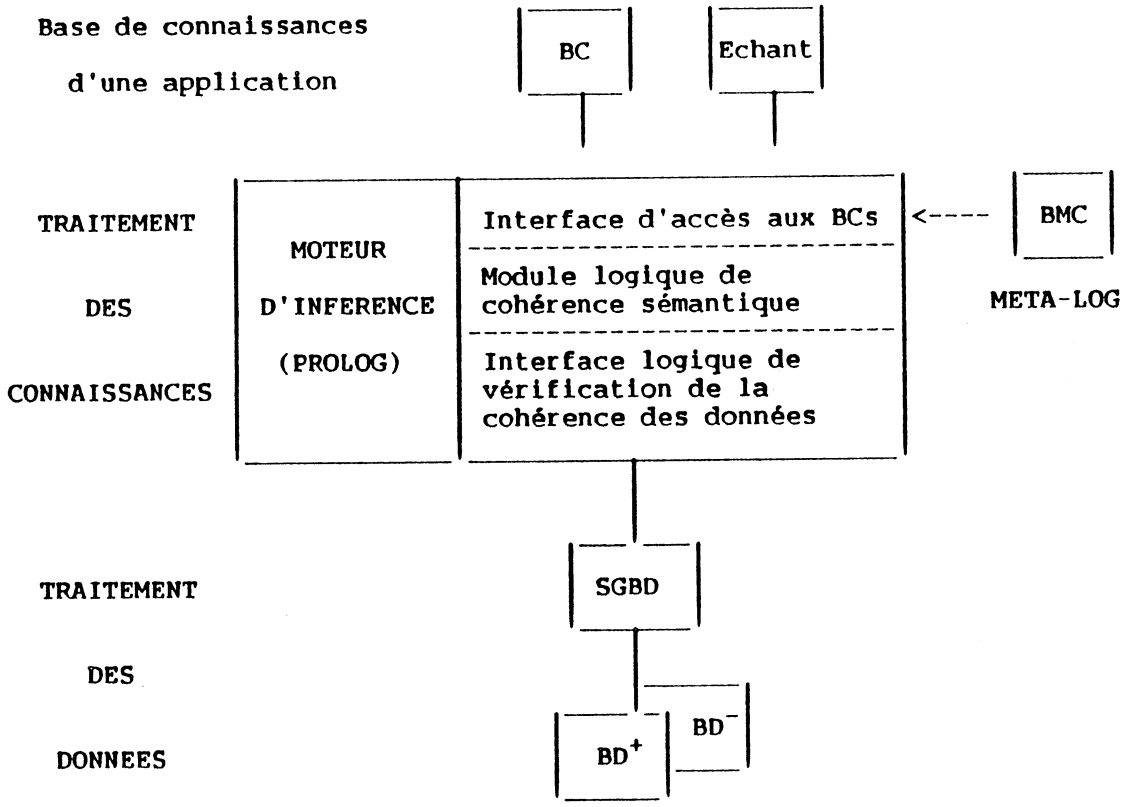


Figure 1 : *Schéma Architectural de SYCSLOG*

Les connaissances spécifiques d'une application sont représentées dans SYCSLOG sous quatre formes :

des faits explicites, positifs et négatifs : ils correspondent aux axiomes de base positifs et négatifs de la théorie. Ils sont stockés sous le SGBD particulier dans des bases de données notées BD^+ et BD^-. Par exemple, les faits explicites {p(a), ~p(b)} sont stockés sous la forme :

BD^+	BD^-
p(a)	p(b)
.	.
.	.
.	.

des faits implicites positifs : ils sont dérivés à partir des faits explicites et des règles de dérivation correspondantes aux axiomes de déduction. A chaque axiome de déduction correspond un ensemble de clauses définies. Chacune constitue ainsi une règle de dérivation. Puisque les faits implicites positifs peuvent être déduits à tout moment, ils ne sont pas stockés; seule la règle de dérivation est stockée dans la BC de l'application. Par exemple, les faits implicites positifs p(a) et p(b) peuvent être dérivés à partir de :

BD^+	BC
q(a)	p(X) ← q(X)
q(b)	

des règles de cohérence ou contraintes d'intégrité : qui assurent la cohérence des connaissances représentées par les faits explicites positifs et négatifs. Elles correspondent aux lois d'intégrité dans la théorie. Chaque loi est mise sous forme de clause (C) et est identifiée par une valeur I. La contrainte est ainsi stockée :

BC
contrainte(I) ← C
.
.

des faits explicites disjonctifs : ils correspondent aux axiomes disjonctifs de base de la théorie. Ils sont stockés en tant que contraintes d'intégrité dans la BC de l'application. Par exemple, le fait disjonctif P ∨ Q est stocké comme suit :

BC

contrainte(I) ← P ∨ Q
.
.

Les formules atomiques P et Q sont évaluées selon leurs mondes respectifs (fermé ou ouvert).

Les connaissances ainsi structurées sont gérées par SYCSLOG en conjonction avec le SGBD spécialisé dans le traitement de faits explicites. Les fonctionnalités offertes pour cette symbiose peuvent être groupées en trois niveaux :

(1) **Au niveau de la capacité de traitement** : grâce à SYCSLOG les fonctionalités du SGBD se trouvent augmentées avec un mécanisme de déduction intervenant dans toute interaction avec la BC de l'application, (traitement de vues, contrôle d'intégrité, réponse aux requêtes déductives).

(2) **Au niveau de la capacité de représentation des connaissances** : des connaissances définies de façon récursive et utilisées pour dériver des faits implicites peuvent être représentées par des formules sous la forme standard de la logique. La possibilité de modéliser sous les approches "monde fermé" et "monde ouvert" permet un traitement plus réaliste des connaissances de nature différente. On peut enfin exprimer des lois générales sur les données avec le même langage de représentation.

(3) **Au niveau de l'intégrité sémantique**, une cohérence globale des connaissances est assurée. Au niveau de la BC, l'adjonction de toute information est contrôlée par l'intermédiaire d'une interface d'accès aux connaissances et de la base META-LOG, de manière à ne pas introduire des incohérences dans la BC. Pour le contrôle d'intégrité au niveau de la BD, un traitement uniforme est donné à toutes les contraintes d'intégrité. Les faits explicites sont modifiés en accord avec les règles de cohérence par l'intermédiaire d'une interface logique de vérification de données.

2. La base de méta-connaissances META-LOG

La base META-LOG constitue une représentation du modèle conceptuel utilisé par SYCSLOG : *le modèle logique*. Les connaissances dans META-LOG sont décrites sous forme de clauses de Horn par l'intermédiaire du langage Prolog. Chaque objet logique de ce modèle est défini par une méta-règle qui décrit la syntaxe et la sémantique de l'objet. C'est à dire ses propriétés structurelles et sa dynamicité. META-LOG assure que toute représentation d'un objet d'une application par un objet logique donné est valide si et seulement si la connaissance de l'objet satisfait la méta-règle correspondante à l'objet logique.

L'exploitation de la base META-LOG n'est possible que par l'intermédiaire de l'interface d'accès aux connaissances. Ceci permet d'assurer une définition et une modification dynamique et cohérente des bases de connaissances des applications, tout en respectant les spécifications du modèle logique, c'est à dire les méta-règles de la base META-LOG. De cette façon, le concepteur d'une application peut créer de nouvelles entités, de nouvelles relations ou de nouvelles contraintes d'intégrité. De même, il peut modifier ou éliminer celles qui éxistent déjà en étant sûr que la base de connaissances reste toujours dans un état cohérent.

3. L'interface d'accès aux BCs

La fonction de l'interface d'accès aux connaissances est d'effectuer tous les contrôles sémantiques définis dans la base de métaconnaissances META-LOG. Ce contrôle utilise un échantillon de la BD que le module logique de cohérence sémantique maintient à jour au fur et à mesure de l'évolution de la BC. De cette manière, l'évolution cohérente de toute BC est assurée.

L'interface considère que chaque méta-règle définit une classe d'objets : la classe des prédicats définis en extension, la classe des prédicats définis en compréhension dont l'extension se trouve dans la BD, etc.

L'activation d'une méta-règle entraîne un traitement particulier pour un objet particulier de la classe définie par la méta-règle. Les activations des méta-règles sont déclenchées par l'interface lors de la réception de messages de la part du concepteur. Une communication interactive à un niveau assez élevé est ainsi établie entre le concepteur et l'interface par l'intermédiaire de messages selon une hiérarchie pré-fixée. Un message de début de communication contient le nom de la classe à

laquelle appartient l'objet à traiter et le traitement particulier à l'objet. Des messages subséquents assurent la réception par l'interface des informations nécessaires au bon terme d'une complète exécution d'une méta-règle.[1]

4. Le moteur d'inférence

Le moteur d'inférence de SYCSLOG utilise comme base le moteur d'inférence du langage Prolog. Celui-ci est enveloppé d'une couche qui permet :

- le traitement de requêtes dans SYCSLOG. L'évaluation de requêtes peut ainsi se faire non seulement sous un monde fermé, hypothèse de Prolog, mais aussi sous un monde ouvert.
- une communication directe avec le SGBD pour la recherche de données stockées dans les bases de données. Cette recherche est faite selon les approches interprétée ou compilée en fonction du type de la requête dans SYCSLOG.
- un traitement cohérent des axiomes disjonctifs par leur exploitation en tant que contraintes d'intégrité à vérifier sur les BD^+ et BD^- selon que les prédicats qui interviennent s'évaluent sous un monde fermé ou ouvert.

4.1. L'évaluation de requêtes sous un monde fermé

Cas 1 : ***Requêtes de vérification de définition.***

L'évaluation de ce type de requêtes est très simple. L'évaluation faite sous l'approche interprétée consiste à vérifier que chaque argument instancié dans un littéral $P(t_1,\ldots, t_n)$ satisfait la clause de définition du prédicat P qui doit avoir été défini par un axiome de structure. Une requête de ce type est de la forme :

$$\leftarrow \text{ correcte } (P(t_1 ,\ldots, t_n)).$$

Cas 2 : ***Requêtes d'intérrogation positives.***

Les requêtes d'intérrogation positives permettent l'intérrogation de la BD^+. Elles peuvent être fermées ou ouvertes. Une requête de ce type est de la forme :

[1] Pour avoir plus de détails sur cette interface nous renvoyons le lecteur à [VILL 85].

$$\leftarrow P(t_1, \ldots, t_n).$$

P est un prédicat définit par un axiome de structure. Si tous les termes sont instanciés, la requête est fermée. Dans le cas contraire elle est ouverte.

Le moteur d'inférence traite cette requête comme suit : tout d'abord les termes instanciés, s'il y en a, sont vérifiés moyennant une requête de vérification de définition. Ensuite, la requête est traduite pour le SGBD (BD^+) qui finalement l'évalue.

Cas 3 : ***Requêtes déductives positives.***

Une requête de ce type est de la forme :

$$\leftarrow P(t_1, \ldots, t_n).$$

P est un prédicat définit par un axiome de déduction. L'évaluation de ce type de requêtes est faite selon une approche compilée et dépend de la caractéristique récursive de la définition du prédicat P. Si cette définition n'est pas récursive, la règle de définition du prédicat P est utilisée pour obtenir un chemin de dérivation qui finit par une conjonction de requêtes d'intérrogation positives portant sur des prédicats dont les faits élémentaires se trouvent dans la BD^+. Cette conjonction est traduite pour le SGBD et constitue la réponse à la requête originale. Une requête portant sur un prédicat dont la définition est récursive est traitée différemment selon que les règles de déduction sont *singulières* ou *non-singulières*. Cette caractérisation est dû à Minker et Nicolas [MINK 83].

" Une règle récursive de la forme $F \wedge R_1 \wedge \ldots\ldots \wedge R_n \rightarrow R$ où F est une conjonction (qui peut être vide) de relations de base, et R_1, R_2, ..., R_n, R des littéraux qui correspondent à une même relation, est singulière si et seulement si :

(1) toute variable qui apparaît dans un littéral R_i et qui n'apparaît pas dans R, apparaît seulement dans R_i.

(2) toute variable dans R apparaît comme argument dans la même position pour tout littéral R_i où elle apparaît, sauf au plus pour un littéral R_i contenant toutes les variables dans R."

Minker et Nicolas ont montré que pour les règles singulières, tout processus de dérivation peut être coupé en conservant la complétude de la réponse à la requête. Les chemins infinis de dérivation sont coupés et la requête de déduction peut ainsi être compilée par une procédure complètement syntaxique (i.e indépen-

dante de l'état de la BD). Si la règle de définition d'un prédicat P est singulière, le moteur d'inférence trouve donc un chemin de dérivation fini qui arrive à une conjonction de littéraux à répondre sur la BD.

Les règles de déduction non-singulières sont traitées en génération. Elles sont activées lors de modifications des extensions des prédicats apparaissant soit dans la règle soit dans des règles de déduction pour d'autres prédicats dans la règle.

4.2. L'évaluation de requêtes sous un monde ouvert.

Cas 1 : ***Requêtes d'intérrogation négatives.***

Une requête de ce type est de la forme :

$$\leftarrow P(t_1, \ldots, t_n).$$

P est un prédicat dont le monde est " ouvert ". La requête peut être fermée ou ouverte. Leur évaluation est faite comme suit :

a) la requête est évaluée comme s'il s'agisait d'une requête d'intérrogation positive.

b) si l'évaluation du pas a) est positive (si la requête est fermée la réponse est "vrai", si la requête est ouverte la réponse est un ensemble de faits) alors l'évaluation de la requête est considérée finie. Par contre, si l'évaluation du pas a) est négative alors la requête :

$$\leftarrow P(t_1, \ldots, t_n).$$

est envoyée au SGBD (BD^-) sous la forme qui correspond. Si le résultat est positif la réponse alors à la requête originale est " faux " suivi des faits négatifs s'il y en a lieu. Si le résultat est négatif, la réponse à la requête originale est " **inconnu** ".

Cas 2 : ***Requêtes déductives négatives.***

Une requête de ce type est de la forme :

$\leftarrow P(t_1, \ldots, t_n)$.

P est un prédicat définit par un axiome de déduction et dont la règle de définition porte sur au moins un littéral dont le monde est égal à " ouvert ".

Supposons que la règle de définition du prédicat P soit la suivante :

$P(t_1,\ldots,t_n) \leftarrow P_1(\ldots.), P_2(\ldots.), P_3(\ldots.)$.

et supposons également que P_2 est évalué sous un monde ouvert et que P_1 et P_3 le sont sous un monde fermé.

L'évaluation de la requête est alors effectuée comme suit :

1) le littéral $P_1(\ldots)$ est évalué sous un monde fermé. Si l'évaluation est positive, le moteur d'inférence passe à la résolution du deuxième littéral. Si par contre l'évaluation est négative, le moteur essaie une autre règle de définition alternative du prédicat P, s'il y en a.

2) le littéral $P_2(\ldots.)$ est évalué comme une requête d'intérrogation négative. Si la réponse est " faux ", le moteur d'inférence essaye alors une autre règle de définition alternative du prédicat P, s'il y en a. Si la réponse est " vrai ", le moteur passe alors à la résolution du littéral $P_3(\ldots.)$. Si la réponse est " inconnu ", le littéral $P_2(\ldots.)$ est rangé pour participer en tant que condition d'une réponse conditionelle à la requête originelle et ensuite le moteur d'inférence passe à la résolution de $P_3(\ldots.)$.

3) le littéral $P_3(\ldots.)$ est évalué sous un monde fermé. Si l'évaluation est positive alors la réponse à la requête originale est :

   - " vrai " : si la réponse à $\leftarrow P_2(\ldots)$ est " vrai "

   - " si $P_2(\ldots.)$ alors $P(t_1,\ldots,t_n)$ " : si la réponse à $\leftarrow P_2(\ldots.)$ est " inconnu ".

4) s'il n'y a pas d'autre règle de définition alternative pour le prédicat P la réponse à la requête originale est donc " faux ".

En résumant, une requête déductive négative à une des trois réponses possibles :

1) *" vrai "*

2) *" faux "*

3) *Si* $C_1,\ldots,C_n$ *alors* $P(t_1,\ldots,t_n)$

Exemple 8

Supposons que dans une application pour un hôpital nous ayons défini les prédicats suivants :

patient : détermine la classe de tous les patients de l'hôpital. L'hôpital suppose que tout patient est connu comme tel, c'est à dire qu'il n'y a pas de patients inconnus à l'hôpital. Chaque patient est caractérisé par un numéro de sécurité sociale, un age, un sexe et une date de naissance.

maladie : détermine la classe de toutes les maladies répertoriées.

maladies_eues : associe à chaque patient les maladies qu'il a eues et à quelle date. Il est normal de supposer que l'hôpital ne connaît pas toutes les maladies d'un patient donné.

symptômes : détermine la classe de tous les symptômes considérés par un médecin de l'hôpital.

symptômes_patient : lorsqu'un patient arrive à l'hôpital on lui associe les symptômes qu'il dit sentir. Le médecin sait qu'il peut y avoir des symptômes inconnus du patient mais qui pourraient servir pour établir un diagnostic. Il peut donc les lui démander.

diagnostic : définit les diagnostics possibles en fonction des symptômes du patient et des maladies qu'il a eu.

Supposons aussi que SYCSLOG fournit les prédicats suivants :

éval_MF et éval_MO : pour spécifier pour un prédicat, s'il est évalué sous un monde fermé ou ouvert.

mode : permet de spécifier le mode d'obtention des faits élémentaires du prédicat (*ext* ou *ded*, directement de la BD ou par déduction moyennant la règle de définition du prédicat).

extensions et compréhension : permettent respectivement de définir un prédicat P en extension ou en compréhension.

relation : permet de définir un prédicat de relation entre des ensembles de termes.

D'après la signification de chaque prédicat nous avons :

patient :

éval_MF(patient).
mode(patient,ext).
compréhensions(patient(#sec_coc, Age, Sexe, Date)) ←[1]
sec_soc(#sec-soc),age(Age),sexe(Sexe),date(Date).

maladie :

extensions(maladie(cancer)).
extensions(maladie(hépatite)).
.
.

maladies_eues :

éval_MO(maladies_eues).
mode(maladies_eues, ext).
relation(maladies_eues(#sec_soc, Maladie, Date)) ←
compréhensions(patient(#sec_soc,_,_,_)),
extensions(maladie(Maladie)),
compréhensions(date(Date)).

symptômes :

éval_MF(symptômes).
mode(symptômes, ext).
compréhensions(symptômes(Code, Desc)) ←
intéger (Code),
string (Desc, 30).

symptômes_patient :

éval_MO(symptômes_patient).
mode(symptômes_patient, ext).
relation(symptômes_patient(#sec_soc, Code)) ←
compréhensions(patient(#sec_soc,_,_,_)),
compréhensions(symptômes(Code, _)).

[1] Bien entendu, les prédicats sec_soc, age, sexe et date ont dû être définis avant le prédicat patient.

diagnostic :

```
mode(diagnostic, ded).
diagnostic(Pac, "infection-reins", 85%) ←
        symptômes_patient(Pac, 10),        /* 10 = avoir de la fièvre */
        maladies_eues(Pac, "hépatite",_),
        symptômes_patient(Pac, 12).        /* 12 = taux de transaminases
                                           dans le sang supérieur à la normal */
```

Une requête de déduction négative serait donc toute requête portant sur le prédicat "diagnostic" car les prédicats symptômes_patient et maladies_eues doivent s'évaluer sous un monde ouvert. Une requête de déduction négative possible est :

← diagnostic (17654, X, Y).

Quelles sont les maladies X que l'on peut diagnostiquer au patient 17654 et avec quelle probabilité ?. Pour l'évaluation de cette requête, supposons que la BD se trouve dans l'état qui suit :

BD^+	BD^-
symptômes_patient(17654,10)	symptômes_patient(17654,15)
symptômes_patient(17654,12)	
.	
.	
maladies_eues(17654,infarctus,10/85)	

l'évaluation est faite comme suit :

1) ← symptômes_patient(17654, 10).
   Cette requête est donc traitée comme une requête d'intérrogation négative puisque le prédicat symptômes-patient s'évalue sous un monde ouvert. La réponse est " vrai " et l'on continue.

2) ← maladies_eues(17654, "hépatite", _).
   Sous la BD^+ maladies_eues(17654, hépatite,_) est faux.
   Sous la BD^- maladies_eues(17654, hépatite,_) est faux.
   Donc, la réponse est " inconnu " et l'on continue l'évaluation de la requête.

3) ← symptômes_patient(17654, 12).
   Cette requête s'évalue à " vrai " sous la BD^+.

Donc, la réponse à la requête ← diagnostic (17654, X, Y) est :

si maladies_eues(17654, hépatite,_) **alors** diagnostic(17654, infection-reins,85%)

Ceci veut dire : *Si le patient 17654 a eu comme maladie l'hépatite alors on peut diagnostiquer une infection des reins avec une probabilité de 0.85.*

Bien entendu, d'autres réponses sont possibles par l'évaluation d'autres règles pour le prédicat " diagnostic ".

5. Le module logique de cohérence sémantique

Ce module est chargé d'assurer l'absence de contradiction à l'intérieur d'une base de connaissances. Pour cela, il utilise les connaissances dans META-LOG et un échantillon de la BD qu'il maintient pour assurer sa fonction. La communication avec ce module n'est possible que depuis l'interface d'accès aux BCs lors de traitements de connaissances. Nous dédions la section 1 du chapitre 5 à la description détaillée de ce module.

6. L'interface logique de vérification de la cohérence des données (LVD)

C'est moyennant cette interface que la communication entre SYCSLOG et un SGBD est possible. Cette communication est toujours établie pour que SYCSLOG effectue les contrôles d'intégrité sémantique explicités dans la BC. Ce sont donc les transactions sous le SGBD qui doivent appeler cette interface. Nous dédions la section 2 du chapitre 5 à la description détaillée de cette interface.

CHAPITRE 4

SYCSLOG

LE MODELE LOGIQUE DE DONNEES

CHAPITRE 4

SYCSLOG : LE MODELE LOGIQUE DE DONNEES

Nous dédions cette section à la description d'un modèle logique de données.[1] Ce modèle est utilisé par le système logique d'intégrité sémantique SYCSLOG proposé dans ce travail et que nous décrirons en détails par la suite.

Dans le domaine de la logique des prédicats, la réalité est décrite par deux types de propositions : les propositions prédicatives (ou attributives) et les propositions relationnelles. Cette distinction n'apparaît pas dans le modèle relationel [CODD 70] bien qu'elle soit dans le modèle entité-relation [CHEN 76].

Une proposition prédicative ou attributive exprime une caractéristique ou propriété pour un sujet donné. Les propositions : "rouge est une couleur" et "Jean Paul est un employé dont le salaire est 6000 francs" sont prédicatives. La première exprime que l'objet ou sujet "rouge" a la propriété ou caractéristique d'être une couleur. La deuxième exprime que le sujet "Jean Paul" est caractérisé par "être un employé dont le salaire est de 6000 francs".

De leur côté, les propositions relationnelles expriment des liens sémantiques entre des objets. Lorsqu'on énonce : "Jean Paul est affecté au département des ventes", la proposition lie les objets "Jean Paul" et "département des ventes" à travers le lien "est affecté au".

Les propositions prédicatives sont représentées par des prédicats de propriété. Ces prédicats peuvent être définis en extension ou en compréhension. Les propositions relationnelles sont représentées par des prédicats de relation toujours définis en compréhension. Les connaissances pour ces prédicats sont considérées comme appartenant au monde ouvert ou au monde fermé selon que l'on représente ou non les connaissances négatives de façon explicite.

---

[1] Nous utilisons par la suite une notation très proche du langage Prolog, utilisé dans l'implémentation, où les connecteurs logiques $\wedge$ et $\vee$ sont représentés respectivement par la "," et le ";".

Par la suite, nous expliquons plus en détails la signification et la manière dont sont définis ces prédicats et leurs règles d'évolution. Nous considérons que les définitions générales des prédicats sont stockées sous forme de règles dans une base de méta-connaissances (META-LOG) et que ces règles permettent de générer des définitions de données spécifiques à une application, sous la forme de règles à stocker dans une base de connaissances (BC). Toute méta-règle définit ainsi la syntaxe et la sémantique d'une classe de prédicats et permet de traiter toute définition (axiome dans la théorie) appartenant à cette classe.

1. Les termes

Un terme est une entité réelle ou abstraite autonome. Par exemple, chaque caractéristique est en elle-même un terme, de même un individu ou un objet quelconque. Les termes peuvent être simples ou composés. Un terme simple est spécifié par une constante (ex. rouge). Un terme composé est représenté par une structure qui comprend les composants de l'entité.(ex. une plante est composée de feuille, tige, fleur et fruit). De même, chaque élément composant peut être un terme composé (ex. une fleur est composée d'étamine, pistil, calice et corolle).

Un terme simple est représenté par :

- une constante (ex. rouge) ou
- une variable (ex. X)

Un terme composé est représenté par :

- une structure ou arbre
  (ex.plante (feuilles, tige, racine
  fleur (étamine, pistil, calice, corolle),
  fruit))

On défini en général un terme par la méta-règle suivante :

[1]
```
terme (X)  ←  (atome (X) ; integer (X)).
terme (X)  ←  var (X).
terme (X)  ←  functor (X,Nom-objet,N-composants),
                  atome (Nom-objet),
```

integer (N-composants),
chaque-composant-est-un-terme (X).

2. Les propriétés de termes

Les termes peuvent être liés sémantiquement lorsqu'ils ont une ou plusieurs propriétés communes. Une propriété est représentée moyennant un prédicat de propriété. La définition d'une telle propriété correspond à un axiome d'unicité de propriété dans la théorie. Lorsque le prédicat de définition est interprété en extension il détermine une classe selon le principe d'abstraction. Lorsqu'il est interprété en compréhension c'est une caractéristique abstraite indépendante du sujet auquel il s'applique. Par exemple, la propriété "être une couleur" lie tous les termes qui ont la propriété d'être une couleur. On spécifie seulement la caractéristique qui lie les termes. Par contre, une interprétation en extension du prédicat unaire "couleur" détermine la classe ou ensemble de faits suivants :

couleur (rouge).
couleur (bleu).
couleur (verte).
.
.
etc

Une propriété peut être définie moyennant un prédicat exprimé en compréhension. Par exemple, l'age X d'un adulte est une valeur entière entre 18 et 80 ans.

age-adulte (X) ← integer (X), $X \geq 18$, $X \leq 80$.

2.1. Les prédicats unaires en extension

Un prédicat de propriété en extension est composé du nom du prédicat - nom de la propriété -, et de l'extension du prédicat, c'est à dire les termes caractérisés par la propriété. Une propriété est définie en extension lorsque la propriété caractérise des objets ou des termes simples dénombrables. Tout prédicat en extension correspond donc à un axiome d'unicité de propriété dans la théorie. Par exemple, la proposition :

rouge, blanc, jaune, noir sont des **couleurs**

est représentée par le prédicat de nom "couleurs" dont l'extension est l'ensemble des termes caractérisés par "être des couleurs", c'est à dire, l'ensemble des couleurs

rouge, blanc, jaune et noir.

La méta-règle de définition d'un prédicat de propriété en extension est la suivante :

[2] **pred-extension** (Op,Nom-prédicat,Ltermes) ←
opération-BC (Op),
atome (Nom-prédicat),
chaque-élément-est-terme-const (Ltermes),
vérif-sémantique-ext (Op,Nom-prédicat,Ltermes),
traiter-pred-ext (Op,Nom-prédicat,Ltermes).

Op est le type d'opération sur la BC, c'est à dire la création, l'adjonction ou l'élimination d'un énoncé de définition d'une propriété *(créer, ajouter et effacer)*.

Nom-prédicat est instancié par le nom de la propriété.

Ltermes est une liste qui doit être instanciée par une liste de termes simples constants, représentant les termes liés par la propriété en question. La signification de ces termes, de même que la vérification sémantique effectuée par la clause verif-sémantique-ext, dépendent de l'opération Op.
La clause de nom traiter-pred-ext exécute la mise à jour de la BC.

La création d'un prédicat en extension

La création d'un prédicat en extension signifie la mise en place d'un ensemble de faits qui constituent l'extension du prédicat et qui représentent, dans la BC, les termes caractérisés par la propriété représentée par le prédicat. Chaque création implique donc que la propriété n'a pas été définie auparavant, car il ne peut pas exister deux propriétés sémantiquement différentes définies sous un même prédicat. Ltermes dans [2] est une liste de termes simples constants, qui constitueront l'extension du prédicat représentant la propriété. Cette liste doit contenir au moins un terme pour définir la propriété.

La création d'un prédicat en extension génère dans la BC autant de faits de définition qu'il y a d'éléments dans la liste de termes Ltermes. Un seul fait est inséré dans le cas de termes répétés.

Exemple 9

Supposons qu'à un moment donné un concepteur veuille créer dans la BC d'une application l'énoncé de définition par extension de la propriété "être un jour de la semaine". Pour cela, un appel à la méta-règle [2] déclenche son exécution comme suit :

```
pred-extension (créer, jour-semaine, [lundi,...,dimanche]) ←
        opération-BC (créer),
        atome (jour-semaine),
        chaque-élément-est-terme ([lundi,...,dimanche]),
        vérif-sémantique-ext (créer,jour-semaine,
                              [lundi,mardi,...,dimanche]),
        traiter-pred-ext(créer,jour-semaine,
                              [lundi,mardi,...,dimanche]).
```

Puisque "créer" est une opération valide, "jour-semaine" est un atome et chaque jour de la semaine est un terme, alors la vérification sémantique peut être effectuée. Si l'énoncé est validé, la clause vérif-sémantique-ext autorise alors la création des faits suivants dans la BC de l'application :

```
extensions(jour-semaine(lundi)).
extensions(jour-semaine(mardi)).
.
.
extensions(jour-semaine(dimanche)).
```

si l'énoncé est invalide, la clause vérif-sémantique-ext échoue et rien n'est créé dans la BC.

L'élimination d'un prédicat en extension

L'élimination d'un prédicat en extension signifie l'effacement total ou partiel des faits qui constituent l'extension du prédicat. Chaque élimination implique donc que la propriété a été définie auparavant. Dans ce cas, Ltermes dans [2] est une liste de termes simples constants qui correspondent aux faits à éliminer de l'extension du prédicat. Chaque terme dans Ltermes correspond donc à l'élimination du fait respectif dans la BC. Une liste de termes vide signifie l'élimination totale de l'extension du prédicat.

La vérification sémantique effectuée par la clause verif-sémantique-ext consiste en :

- s'il s'agit d'une élimination partielle, vérifier que les termes à effacer existent dans l'extension du prédicat
- s'il s'agit d'une élimination totale (soit Ltermes=[] soit Ltermes constitue l'extension complète du prédicat), vérifier qu'il n'y a pas d'autres propriétés ou relations utilisant la propriété qu'on est en train d'éliminer.

La modification d'un prédicat en extension

La modification d'un prédicat en extension signifie l'adjonction de faits à l'extension du prédicat. Pour chaque terme dans Ltermes un fait est ajouté. La modification n'est possible que s'il existe déjà au moins un fait définissant la propriété. La liste de termes Ltermes ne doit pas être vide et aucun des termes à ajouter ne doit exister dans l'extension du prédicat.

2.2. Les prédicats en compréhension

Une propriété peut être représentée par un prédicat défini en compréhension. Un tel prédicat est composé du nom du prédicat - nom de la propriété -, et de la compréhension du prédicat. Celle-ci comprend un terme générique représentant les termes caractérisés par la propriété, et des contraintes ou des conditions sur ces caractéristiques, s'il y en a. Par exemple, le prédicat de propriété "salaire" est défini en compréhension par le terme générique salaire(s) et les contraintes de définition entier(s), $s \geq 1500$ et $s \leq 50000$.

Un prédicat défini en compréhension permet aussi de représenter un ensemble de termes possédant plusieurs propriétés communes. Par exemple, l'ensemble des employés peut être représenté par le prédicat de propriété "employé" dont le terme générique est :

employé (sec-soc, nom, salaire, catégorie)

Ceci veut dire que tout terme caractérisé par "être un employé" est représenté par les caractéristiques sec-soc, nom, salaire et catégorie.

La méta-règle de définition d'un prédicat en compréhension est la suivante :

[3] **pred-compréhension** (Op,Pred-Termes,Lcond,Lclés,Mode,Monde) ←
opération-BC (Op),
est-clause-comp (Op, Pred-Termes),
functor (Pred-Termes, Nom-préd, 1),
atome (Nom-préd),
chaque-élément-est-condition (Lcond),
vérif-sémantique-comp (Op, Pred-Termes, Lcond, Mode, Lclés),
traiter-pred-comp (Op, Pred-Termes, Lcond, Mode, Lclés, Monde).

Op est le type d'opération sur la BC, c'est à dire la création, l'adjonction ou l'élimination d'un prédicat en compréhension *(créer, ajouter et effacer)*.

Lcond est une liste qui devra être instanciée par une liste de conditions ou contraintes comportant la compréhension du prédicat.[1]

Monde est un paramètre qui désigne sous quelle hypothèse : monde fermé ou monde ouvert, devront être traité les connaissances du prédicat en train de définir.

Mode est un paramètre qui permet de déclarer le mode d'exploitation du prédicat.

- Si Mode = **type** on considère que la règle de définition pour la propriété est utilisée pour vérifier qu'un terme donné satisfait la propriété. Le terme générique doit être de la forme Pred-Termes = Nom-pred et Lclés doit être la liste vide. La conjonction de conditions dans Lcond définit les termes ayant la propriété Nom-pred. Dans ce cas, la règle de définition est un axiome d'unicité de propriété, le paramètre Monde prend donc automatiquement la valeur "fermé".
- Si Mode = **ext** on considère que les données correspondantes au prédicat se trouvent stockées explicitement dans la BD. Le terme générique est de la forme Pred-Termes = Nom-pred(p_1, ..., p_q). Lclés contient de façon optionnelle les caractéristiques clés[2] parmis p_1, ..., p_q. Si une caractéristique est entourée par des crochets, la liste fait référence à une liste de valeurs pour la caractéristique. Par exemple, une liste de salaires est représentée par [salaire]. La conjonction de p_1, ..., p_q et des conditions dans Lcond détermine le prédicat de définition

[1] Les conditions sont formées à partir de prédicats de base (entier, chaine), d'opérateurs de comparaison (<,>,=) et d'opérateurs arithmétiques (+,-,*,/).

[2] Les valeurs correspondantes aux attributs clés dans un n-uplet permettent de l'identifier de manière unique.

de l'ensemble des termes ayant les caractéristiques p_1, ..., p_q. Cette conjonction constituera donc la partie droite de la règle à générer pour le prédicat Nom_pred. La règle ainsi générée sert à vérifier pour le prédicat que tout fait explicite (positif si Monde=fermé, positif et négatif si Monde=ouvert) satisfait les restrictions de définition. Dans ce cas la règle de définition pour le prédicat est un axiome de structure dans la théorie.

- Si Mode = **ded**, le terme générique est de la forme Pred-Termes = Nom-Pred(p_1, ..., p_q). Les prédicats p_1, ..., p_q caractérisent les variables quantifiées universellement dans l'axiome de déduction de la théorie et qui quantifient le prédicat Nom_pred. Lcond est une liste de sous-listes qui contient l'antécédant de l'axiome de déduction. La liste est considérée comme la conjonction des sous-listes et chaque sous-liste comme une disjonction des éléments qui la forment. Les évaluations pour un prédicat Nom-Pred dont le Mode=ded sont faites toujours sous un monde fermé.

La création d'un prédicat en compréhension

La création d'un prédicat en compréhension signifie la mise en place d'une règle de définition pour le prédicat. Chaque création implique donc que le prédicat n'a pas été défini auparavant. Les conditions peuvent être représentées par des prédicats prédéfinis dans le langage ou par des prédicats définis obligatoirement auparavant.

La clause vérif-sémantique-comp dans [3] vérifie donc :

- qu'il n'y a aucun prédicat défini sous ce nom
- que les termes dans Pred-Termes sont des prédicats définis dans la BC et qu'ils ne se repètent pas
- que les conditions dans Lcond portent uniquement sur les prédicats spécifiés dans Pred-Termes
- si Lclés n'est pas vide et Mode=ext, que les prédicats spécifiés pour les clés appartiennent à l'ensemble de propriétés spécifié dans Pred-Termes.
- que l'adjonction de l'énoncé de définition de la propriété n'est pas en contradiction avec les énoncés déjà existants dans la BC.

La clause traiter-pred-comp dans [3], outre la création physique de la règle de définition pour le prédicat, stocke :

- pour chaque terme Ti dans Pred-Termes, un fait de la forme compréhension(dépends(Nom-Pred(Ti))). Ce fait exprime la dépendance du prédicat Nom-pred de la caractéristique représentée par le terme Ti,
- si Mode=ext deux faits de contrôle d'unicité des valeurs de clés. Un fait contenant les caractéristiques clés pour le prédicat et un autre pour la vérification de l'unicité des valeurs de clés. Ce dernier fait est détaillé au paragraphe "La contrainte générale d'unicité des valeurs de clé" à la fin de ce chapitre. Ces faits ont un intérêt purement opérationnel au moment des vérifications sémantiques.
- un fait indiquant pour le prédicat le monde sous lequel il est traité. Le fait est de la forme éval_MF(Nom-pred) ou éval_MO(Nom-pred) selon la valeur du paramètre Monde.
- un fait indiquant le mode du prédicat. Selon la valeur du paramètre Monde le fait a l'une des formes suivantes : mode(Nom-pred,type), mode(Nom-pred,ext) et mode(Nom-pred,ded).
- si Mode=ext un ensemble de règles de dérivation de contenu, correspondant aux axiomes de contenu dans la théorie, est créé pour le prédicat Nom-pred. Pour chaque caractéristique $p_i \notin$ Lclés la règle suivante est générée :

$$p_i\text{-Nom-pred}(Lc_1,\ldots,Lc_n, VP_i) \leftarrow \text{Nom-pred}(p_1,\ldots,p_q)$$

où Lclés = $[Lc_1, \ldots, Lc_n]$ et Lclés $\cup$ $VP_i \subseteq \{ p_1, \cdots, p_q \}$.

Exemple 10

On veut définir l'ensemble des intervalles par un prédicat de nom "intervalle". Un intervalle est un terme composé d'un nom, d'une borne inférieure et d'une borne supérieure. On impose comme condition sur ces termes que la borne inférieure doit être inférieure à la borne supérieure. On suppose que borne_inf et borne_sup ont été définis auparavant comme des entiers par des prédicats en compréhension et que "nom" a été défini comme une chaine de caractères.

← pred-compréhension (créer,
intervalle(nom, borne_inf, borne_sup),
[borne_inf<borne_sup], [], ext, fermé).

L'exécution de cette requête par la règle [3] génère dans la BC la règle suivante :

compréhension (intervalle(A_nom,A_borne-inf,A_borne-sup)) ←
compréhension(nom(A_nom)),
compréhension(borne-inf(A_borne-inf)),
compréhension(borne-sup(A_borne-sup)),
A_borne-inf < A_borne-sup.

et les faits :

compréhension (dépends(intervalle(borne_inf))).
compréhension (dépends(intervalle(borne_sup))).

Puisque l'on n'a pas spécifié des attributs clés, tous les attributs comporteront la clé, c'est à dire :

key (intervalle(nom, borne_inf, borne_sup)).

et : eval_MF (intervalle).
mode (intervalle, ext).

Exemple 11

Si maintenant nous voulons définir une règle de dérivation pour les intervalles dont la magnitude - la différence entre les deux bornes - est inférieure à 10. L'appel à formuler est le suivant :

← pred-compréhension (créer,
intervalle_inf10(nom, borne_inf, borne_sup),
[[intervalle(nom, borne_inf, borne_sup),
borne_sup - borne_inf ≤ 10]],
ded, fermé).

La règle de dérivation générée par cet appel est :

```
compréhension (intervalle_inf10(A_nom,A_borne_inf,A_borne_sup)) ←
        compréhension (intervalle(A_nom, A_borne_inf,
                                  A_borne_sup) ),
        ≤( - (A_borne_sup,A_borne_inf),10).
```

et les faits :

```
compréhension (dépends(intervalle_inf10(intervalle))).
eval_MF (intervalle_inf10).
mode (intervalle_inf10, ded).
```

L'élimination d'un prédicat en compréhension

L'élimination d'un prédicat en compréhension signifie l'effacement de(s) la règle(s) de définition du prédicat. Chaque élimination implique donc que le prédicat a été défini auparavant.
Pour éliminer toutes les règles de définition du prédicat la liste Lcond dans [3] doit être vide. Si l'on ne veut éliminer qu'une règle de définition, Lcond doit contenir les conditions qui la déterminent. Une élimination partielle n'est possible que si la règle à éliminer n'est pas la seule règle de définition pour le prédicat. Pour autoriser une élimination totale il faut qu'il n'y ait pas de propriétés ni de relations utilisant la propriété qu'on est en train d'éliminer.

Lorsque le mode du prédicat est "ext", l'élimination de sa règle de définition signifie implicitement que la BD ne contient plus l'extension du prédicat. De même, si Lcond n'est pas vide ou ne correspond pas complètement aux conditions qui déterminent la règle de définition du prédicat, l'élimination est considérée comme une élimination virtuelle des propriétés explicitées dans Lcond. Cela veut dire qu'à l'ensemble de propriétés groupées par le prédicat et dont l'extension se trouve dans la BD, on en enlève une ou plusieurs propriétés. L'extension correspondante reste sans modification. L'élimination virtuelle est matérialisée en éliminant les règles de dérivation de contenu pour les propriétés à éliminer virtuellement. Une élimination virtuelle peut correspondre aussi à l'élimination d'une propriété virtuelle (créée par une modification virtuelle).

La modification d'un prédicat en compréhension

La modification d'un prédicat en compréhension dont le Mode ≠ ext correspond à l'adjonction d'une règle de définition pour le prédicat. Cette nouvelle règle est une définition alternative ou vue différente pour le prédicat. La modification est possible s'il existe au moins une règle de définition pour le prédicat. Le terme générique pour la nouvelle règle n'est pas modifiable, seule les contraintes ou conditions sur ces caractéristiques sont changées. En ce cas, la clause vérif-sémantique-comp dans [3] vérifie donc :

- qu'il y a au moins une règle de définition pour la propriété
- que Lcond ne contient que des conditions valides
- que l'adjonction de l'énoncé de définition de la propriété n'est pas en contradiction avec les énoncés déjà existants dans la BC.

Dans ces conditions, la clause traiter-pred-comp a les mêmes fonctions que la création.
La modification d'un prédicat en compréhension dont le Mode = ext signifie l'adjonction virtuelle d'une propriété à l'ensemble de termes groupés par le prédicat. La liste Lcond contient le nom de la nouvelle propriété et une valeur pour la propriété. Si Pred-Termes est un terme générique non instancié, la valeur dans Lcond est associée à tous les termes groupés par le prédicat. Si par contre Pred-Termes a les caractéristiques clés instanciées, la valeur dans Lcond est associée au terme désigné par la valeur de clé. La modification virtuelle est matérialisée par l'adjonction d'une règle de dérivation de contenu pour la nouvelle propriété.

3. Les relations entre termes

Les relations entre des ensembles de termes sont représentées par des prédicats de relation à n arguments et avec $n=q+r$, $q \geq 2$. Les q premiers termes sont des prédicats de propriété ou de relation représentant les termes mis en relation. Les r termes qui restent sont des caractéristiques de la relation représentées par des prédicats de propriété. Une relation peut ou non avoir des caractéristiques propres.

La méta-règle de définition est la suivante :

[4] **relation** (Op, Nomrel-Termes, Lpred, Mode, Monde) ←
opération-BC (Op),
est_clause_rel (Op, Nomrel-Termes),
functor (Nomrel-Termes, Nomrel,_),
atome (Nomrel),
chaque_feuille_est_atome (Nomrel-Termes),
vérif-sémantique-rel (Op, Nomrel-Termes, Lpred, Mode),
traiter-relation (Op, Nomrel-Termes, Lpred, Mode, Monde).

Op est le type d'opération sur la BC.

Mode représente les modes d'exploitation possibles pour un prédicat de relation : *ext* et *ded*. Ceux-ci ont la même signification que pour les prédicats de propriété en compréhension.

Monde représente les mondes d'évaluation possibles : *ouvert* et *fermé*.

Nomrel-Termes est un arbre Prolog dont la racine est le nom de la relation, Nomrel, et les feuilles sont des prédicats représentant les termes liés. Lpred est une liste de prédicats de propriété et de conditions spécifiant les caractéristiques associées à la relation et les contraintes sur ces caractéristiques. Si Mode=ded, les feuilles sont les noms des rôles que jouent dans la relation les termes mis en relation. Lpred est une liste tel que Lcond pour les prédicats en compréhension dont le mode est égal à "ded".

La création de relations

La création d'un prédicat de relation signifie la mise en place par la première fois d'une règle de définition pour la relation. Chaque création implique :

- vérifier qu'il n'y a aucun prédicat de relation défini sous ce nom dans la BC.
- si Mode = ext, vérifier que chaque argument dans Nomrel-Termes doit être défini dans la BC comme un prédicat de propriété ou de relation.
- si Mode = ext, vérifier que chaque élément dans la liste Lpred est, soit un prédicat de propriété, soit une condition sur les prédicats de propriété intervenant dans la relation.

- si Mode = ded, vérifier que chaque rôle dans nomrel_Termes apparait au moins une fois dans les feuilles des arbres dans Lpred. Chaque arbre dans Lpred doit correspondre à un prédicat déjà défini.

- vérifier que l'adjonction de l'énoncé de définition pour la relation n'est pas en contradiction avec la BC.

La clause traiter-relation crée la règle de définition pour la relation et stocke les faits correspondants au mode et au type d'évaluation pour le prédicat de relation (fermée ou ouverte).
Si Mode = ext, la clé de la relation est composée comme suit :

- s'il y a des prédicats dont l'extension se trouve dans la BD, parmi les prédicats qui participent à la relation, alors la clé de la relation est composée par les clés de ces prédicats.

- si aucun des prédicats qui participent à la relation n'a de clé, alors la clé de la relation est composée de tous les termes des prédicats qui participent à la relation. Ceci est le cas de prédicats en extension et de prédicats en compréhension dont le Mode = type.

Les règles de dérivation de contenu sont aussi générées tel que nous l'avons décrit pour les prédicats de propriété.

Exemple 12

Supposons qu'on a les définitions des employés et des départements dans la BC d'une application. Un employé est défini par un prédicat de propriété qui groupe les caractéristiques d'un employé. Un département est défini de façon similaire, mais avec les caractéristiques suivantes : un numéro de département Ndep, un Nom et un Budget, le numéro de département Ndep étant la caractéristique clé. Supposons que l'on veuille créer une relation de nom "affectation" entre des employés et des départements dont la signification est : des employés sont affectés aux départements, ou bien des départements ont des employés. Chaque fait de la relation ayant comme caractéristique la date à partir de laquelle l'employé a été affecté au département, il faudrait formuler l'appel suivant à la méta-règle [4] :

← relation (créer, affectation(employé,département), [date], ext, fermé).

Cet appel est donc unifié avec la méta-règle [4] qui effectue toutes les vérifications

syntaxiques et sémantiques et qui insère dans la BC la règle de définition suivante :

relation (affectation(Nemp, Ndep, Date)) ←
compréhension (employé(Nemp,_,_,_)),
compréhension (département(Ndep,_,_)),
compréhension (date(Date)).

L'exécution de la méta-règle [4] vérifie entre autres qu'il n'y a pas d'autre prédicat sous le nom affectation et que les prédicats "employé" et "département" ont été définis auparavant dans la BC. Il faut également vérifier que l'attribut Date est défini moyennant un prédicat en compréhension dont le Mode = type.
De même que pour les prédicats de propriété en compréhénsion un fait de contrôle d'unicité des valeurs de clés est généré. Pour cet exemple le fait généré est le suivant :

key (affectation(Nemp,Ndep,_), affectation(Nemp,Ndep,_)).

La clé de la relation affectation est composée des clés des prédicats "employé" et "département" participant dans la relation. Le fait généré pour ceci est :

key (affectation(sec-soc, num-dep)).

Puisque la relation affectation aura une extension dans la BD^+ et que leur évaluation sera sous un monde fermé, on a :

eval_MF (affectation).
mode (affectation,ext).

Une seule règle de dérivation de contenu est générée car il y a une seule propriété qui ne participe pas dans la clé de la relation. Donc :

date-affectation (Nemp, Ndep, Date) ←
affectation (Nemp, Ndep, Date) .

Exemple 13

Supposons que l'on définit la relation père par les faits suivants :

relation (père(Père, Fils)) ←
compréhension (nom(Père)),
compréhension (nom(Fils)).
eval_MF (père).
mode (père,ext).

Maintenant, nous définisons la relation ancêtre de façon à pouvoir dériver les faits de cette relation à partir de ceux de la relation père. L'appel à la méta-règle [4], en accord avec ceci, doit être :

← relation(créer, ancêtre(Pere,Fils), [[père(Père,Fils)],
[père(Père,Fils1),ancêtre(Fils1,Fils)]],
ded, fermé).

La première disjonction (ou première sous-liste) permet de générer la règle qui établit que tout père est l'ancêtre de ses fils :

relation (ancêtre(Père, Fils)) ← relation (père(Père, Fils)).

Le deuxième conjunct ajoute une règle de définition pour la relation ancêtre. Celle-ci établit que tout père est l'ancêtre des descendants de ses fils :

relation (ancêtre(Père, Fils)) ←
relation (père(Père, Fils1)),
relation (ancêtre(Fils1, Fils)).

les faits suivants sont aussi générés :

eval_MF (ancêtre).
mode (ancêtre, ded).

L'élimination de relations

L'élimination d'un prédicat de relation signifie l'effacement de(s) la règle(s) de définition pour la relation. Chaque élimination oblige donc que la relation ait été définie auparavant.
Dans ce cas, Nomrel-Termes dans [4] correspond au nom de la relation. Si Lpred est vide l'élimination est totale. Si par contre Lpred n'est pas vide et contient des conditions, l'alternative de définition pour la relation déterminée par ces conditions sera éliminée. De même que pour les prédicats de propriété, l'élimination de la règle de définition pour une relation entraîne l'élimination de toutes ses contraintes

d'intégrité et contraintes opérationnelles générées comme conséquences de ces contraintes d'intégrité. Si le Mode=ext et Lpred, en étant différente de vide, ne correspond pas exactement aux conditions qui déterminent la règle de définition de la relation, l'élimination est considérée comme une élimination virtuelle des propriétés spécifiées dans Lpred comme pour les prédicats de propriété en compréhension.

La modification de relations

La modification d'un prédicat de relation a la même signification que la modification dans le cas des prédicats de propriété en compréhension.

4. Les contraintes d'intégrité

Une contrainte d'intégrité restreint l'existence des faits de la BD qui constituent des extensions de prédicats. A chaque prédicat peut être associé un ensemble de contraintes d'intégrité. Elles sont vérifiées uniquement lors de mises à jour de l'extension du prédicat dans la BD.[1] Toute contrainte d'intégrité validée est traduite automatiquement dans une ou plusieurs contraintes opérationnelles à vérifier au moment de l'éxécution de mises à jour sur les données.

Comme dans une grand partie de la littérature sur les contraintes d'intégrité nous considérons les CIs classées en deux catégories : *statiques* et *dynamiques*. Les premières ne référencent qu'un seul état de la BD et les deuxièmes référencent deux états consécutifs.

4.1. Les CIs statiques

Une contrainte d'intégrité statique (ou plus simplement CI) est une f.b.f. fermée et libre de fonctions du calcul des prédicats du 1er ordre. Toute CI lors de sa création est récrite sous la forme *clausale*.

---

[1] Dans la théorie relationnelle, ces contraintes sont nommées "indépendantes du domaine" car une mise à jour ne change pas les domaines sous-jacents.

Ceci obéit à deux objectifs différents :

(1) Le premier objectif est de pouvoir vérifier la non-contradiction de la CI avec les autres CIs. Ceci est possible grâce au fait que la méthode de mise d'une formule en forme clausale (FC) préserve la satisfaisabilité (insatisfaisabilité) de la formule. Une fois que les CIs sont sous forme clausale, il est possible d'utiliser un démonstrateur de théorèmes basé sur la méthode de résolution pour en vérifier la satisfaisabilité de l'ensemble de clauses.

(2) Le deuxième objectif est de pouvoir vérifier la CI lors de toute requête de modification de données. L'ensemble des CIs doit maintenant être non-contradictoire sous le nouvel état de la BD.

Par la suite nous considérons une CI comme une f.b.f sous la forme clausale c'est à dire que nous supposons que la méthode de mise en FC a été appliquée à la CI sous forme standard pour obtenir un ensemble de clauses. De cette manière la détermination de la satisfaisabilité de la CI originale équivaut à déterminer la satisfaisabilité de chacune des clauses de l'ensemble.

Donc, une contrainte d'intégrité est une formule de la forme :

$$\sim A_1 \vee \sim A_2 \vee \ldots \vee \sim A_n \vee B_1 \vee \ldots \vee B_k$$

ou sous forme implicationnelle :

$$A_1 , \ldots , A_n \Rightarrow B_1 , \ldots, B_k$$

où les A_i et B_j sont des formules atomiques dont les prédicats représentent des propriétés, des relations ou des prédicats prédéfinis dans le langage. L'antécédant correspond à la conjonction des formules atomiques A_1, ..., A_n et le conséquent à la disjonction des formules atomiques B_1, ..., B_k.

Une CI est sous forme de clause de Horn lorsque au plus un littéral est positif. En particulier, pour les relations, on peut spécifier des contraintes exprimant des dépendances fonctionnelles, multivaluées, etc. De nombreux auteurs ont étudié les rapports existants entre les clauses de Horn et les dépendances portant sur des relations d'une BD [FAGI 82], [GRAN 82], [NICO 78], [REIT 79]. Notamment, les travaux de R.Fagin [FAGI 82] sont d'un grand intérêt. Il montre que toute dépendance est un cas particulier des dépendances d'implication (DI). Une DI n'est plus qu'une clause de Horn, c'est à dire un énoncé de la forme :

$$((A_1 \wedge .... \wedge A_n) \Rightarrow B)$$

où les Ai sont des formules relationnelles, c'est à dire des expressions de la forme

$p(T_1,\ldots,T_n)$ où p est un prédicat de relation et T_1, ..., T_n des termes ($n \geq 2$). B est soit une formule relationnelle soit une égalité.

Par défaut, une contrainte d'intégrité référence des faits appartenant à un même état de la BD. Cet état est, bien entendu, l'état actuel. L'état actuel est un état valide car il satisfait toutes les restrictions exprimées moyennant les contraintes. Le processus de vérification de contraintes assure que la BD passe toujours d'un état actuel cohérent dans un nouvel état cohérent. Donc, lorsqu'on énonce une contrainte d'intégrité sous forme de clause telle que celle décrite ci-dessus, on veut dire implicitement :

si les formules atomiques A_1, A_2,... et A_n sont vraies sous l'état actuel de la BD alors au moins une des formules atomiques B_i est aussi vraie sous cet état.

Pour vérifier que l'exécution d'une requête de modification n'entraîne pas la BD dans un état incohérent, on vérifie les CIs sur un nouvel état fictif obtenu en effectuant virtuellement les modifications de la requête sur l'état actuel. Toute contrainte d'intégrité est ainsi traduite dans des contraintes opérationnelles qui supposent que l'opération de mise à jour qui met en cause la contrainte a été effectuée et vérifient les conditions qui lui correspondent.

Exemple 14

Prenons comme exemple la contrainte d'intégrité suivante :

(affectation(Nemp,Ndep1,_), affectation(Nemp,Ndep2,_)) $\Rightarrow$ (Ndep1 = Ndep2)

Le symbole de variable " _ " représente une variable anonyme quantifiée implicitement de façon universelle. La contrainte exprime implicitement que si un employé Nemp est actuellement affecté dans deux départements Ndep1 et Ndep2 alors les deux départements sont obligatoirement le même. Autrement dit, tout employé se trouve actuellement affecté dans un seul département.

4.2. Les CIs dynamiques

Les contraintes d'intégrité dynamiques sont des clauses dont les formules atomiques ont une signification implicite. Si nous voulions exprimer une contrainte entre deux ensembles de faits appartenant à deux états consécutifs de la BD, il faudrait spécifier la contrainte comme suit :

Opent(FMAJ) , FA , FAN1 ⇒ FAN2 (1)

Opent est une des opérations possibles pour un fait appartenant à l'extension d'un prédicat dans la BD. C'est l'opération à effectuer sur l'état actuel (insérer, modifier, effacer).

FMAJ est une formule atomique dont le prédicat est soit un prédicat de propriété soit un prédicat de relation ayant une extension dans la BD. Elle représente le fait résultant de l'opération Opent sous l'état nouveau. Lorsque l'opération Opent est égal à modifier, le prédicat Opent a deux arguments. La première formule atomique représente le fait à modifier et la deuxième le fait modifié.

FA est une conjonction de formules atomiques. Chacune porte sur **l'état actuel** de la BD. Les prédicats de ces formules sont des prédicats de propriété, des prédicats de relation ou des prédicats prédéfinis restreignant les termes variables dans les formules. Ces prédicats définent bien entendu le fait à mettre à jour et les conditions qui doivent exister sur l'état actuel pour que la contrainte soit appliquée.

FAN1 est une conjonction de formules atomiques, dont les prédicats sont des prédicats prédéfinis restreignant les termes variables dans les formules FMAJ et FA. Ces formules explicitent des conditions entre l'état actuel et le fait résultat sur l'état nouveau (FMAJ) pour que la contrainte soit appliquée.

FAN2 est une conjonction de formules atomiques. Chacune porte sur **les états nouveau et actuel** de la BD. Les prédicats de ces formules sont des prédicats de propriété, des prédicats de relation ou des prédicats prédéfinis dans le langage. Ces formules explicitent des conditions qui devront se satisfaire, si la contrainte est appliquée, entre l'état actuel et le fait résultat sur l'état nouveau.

La sémantique d'une contrainte d'intégrité référencant deux états consécutifs d'une BD est la suivante :

Si, avant d'effectuer l'opération Opent pour produire le fait représenté par FMAJ, les conditions FA et FAN1 référencant l'état actuel et l'état nouveau sont vraies, alors il faudrait que les conditions FAN2 contraignant ces deux états soient aussi vraies pour que la contrainte soit validée.

Notre approche est similaire à celle de *"actions-relations"* dans Nicolas et al. [NICO 77] pour les lois de transition. La partition qu'ils font des arguments de la relation contrainte par la loi, en invariants, variants et "irrelevants" est implicite

dans notre cas. Les arguments invariants sont représentés par les mêmes variables dans les formules FMAJ et FA. Les arguments variants sont explicités par les formules dans FAN1 et les arguments "irrelevants" sont des variables dans FMAJ et FA qui n'apparaissent pas conditionnées dans FAN1 et qui restent anonymes dans l'implementation. Par contre, nous considérons qu'il peut y avoir une autre relation, participant dans la loi de transition, différente de celle sur laquelle a été imposée la dite loi. Ceci est manifesté par la possibilité d'inclure la dite relation en tant que formule dans FA. En contre partie, l'idée d'avoir une relation définissant des transitions valides pour un argument, permettant ainsi d'exprimer plusieurs règles de transition dans une seule expression, nous semble très importante à exploiter. Dans notre approche elle peut être mise en oeuvre par des prédicats définis en compréhension et formant une partie de FAN2.

Exemple 15

Considérons comme exemple la contrainte dynamique suivante : *" si un programmeur a la catégorie "junior" alors leur prochaine catégorie doit obligatoirement être la "senior""*. L'opération mise en question étant la modification de la catégorie d'un programmeur, Opent est égal à modifier. Le fait résultant de l'opération "modifier" sous l'état nouveau est un fait de l'extension de la propriété "programmeur" (FMAJ = programmeur(X,Y)). Le fait à modifier est, bien entendu, un fait de l'extension de la propriété "programmeur" dont la catégorie est la "junior", c'est à dire : programmeur(X,junior). L'état actuel n'est pas référencé par d'autres propriétés ou relations, FA est donc la formule "vrai". Puisqu'il n'y a plus de conditions à exprimer entre l'état actuel du programmeur X et son état futur, FAN1 est aussi la formule "vrai". Enfin, si la catégorie à modifier d'un programmeur X est la catégorie "junior", la condition qui doit se satisfaire est que leur nouvelle catégorie soit égal à "senior". Donc, en résumant, la contrainte est spécifiée comme suit:

modifier(programeur(X, junior), programmeur(X, Y)) ⇒ Y=senior
- Opent- ---- état actuel ----- --- état nouveau ---

Cette contrainte peut être lue comme : si un programmeur X a à présent la catégorie "junior" et s'il doit changer de catégorie il faut que la nouvelle catégorie soit la "senior". Une autre façon de formuler cette contrainte est la suivante :

modifier (programeur(X,junior),programmeur(X,senior)) ⇒ vrai .

Exemple 16

Nous pouvons définir la contrainte dynamique antérieure d'une façon plus générale. Supposons que les changements possibles de catégorie pour un programmeur sont uniquement les suivants : junior-senior1, junior-senior2, senior1-senior2 et senior2-senior3. Ces changements sont définis par le prédicat changecat comme suit :

mode (changecat, ext).

compréhension (changecat(C1, C2)) ←
extensions(catégorie(C1)),
extensions(catégorie(C2)).

La contrainte dynamique définissant ces changements possibles est alors :

modifier(programeur(X,C1), programmeur(X,C2)) ⇒ changecat(C1,C2) .

4.3. La définition des contraintes d'intégrité

La base META-LOG contient deux méta-règles de définition de contraintes d'intégrité. Ces méta-règles permettent la vérification syntaxique et sémantique d'une contrainte lors de son insertion ou de son élimination d'une BC donnée. La méta-règle contrôle aussi les traitements implicites pour les contraintes opérationnelles déclenchées par le traitement de la contrainte d'intégrité.

[5] **contrainte-statique** (Op, CIFS, message(M)) ←
transforme (CIFS, CIFC),
vérif-syntaxe (CIFC),
vérif-sémantique-cont (Op, CIFC),
traiter-contrainte (Op, CIFS, CIFC).

[6] **contrainte-dynamique** (Op, CIDFC, message(M)) ←
vérif-syntaxe (CIDFC),
vérif-sémantique-cont (Op, CIDFC),
traiter-contrainte (Op, CIDFC).

Op est le traitement à effectuer sur la contrainte : *ajouter* ou *effacer*.

CIFS est une chaine de caractères contenant la contrainte d'intégrité sous la forme standard.

M est une chaine de caractères contenant le message à envoyer lors de violations de la CI.

CIFC est une liste de sous-listes contenant la CI sous la forme clausale. Chaque sous-liste est considérée comme la disjonction des littéraux qui la forment donc chacune représente une clause.

$$CIFC = [\ [A_{11},A_{12},\ldots,A_{1l1}]\ ,\ldots,\ [A_{m1},A_{m2},\ldots,A_{mlm}]\ ]$$

CIDFC est une liste contenant la CI dynamique sous la forme de clause implicationnelle :

$$CIDFC = [\ [A_1,\ ..,\ A_n],\ [B_1,\ ..,\ B_q]\ ]$$

Chaque A_i, B_i est un arbre Prolog dont la racine doit être le nom d'un prédicat de propriété, d'un prédicat de relation ou d'un prédicat prédéfini dans le langage. La sous-liste $[A_1, \ldots, A_n]$ est interprétée comme la conjonction des formules A_i. Celle-ci constitue l'antécédant de la contrainte. La sous-liste $[B_1, \ldots, B_q]$ est interprétée comme la conjonction des formules Bi constituant le conséquent d'une contrainte sur deux états consécutifs de la BD.

La clause "transforme" dans [5] utilise un ensemble de règles qui permettent de récrire une f.b.f en forme standard (CIFS) dans la forme clausale (CIFC). La clause "vérif-syntaxe" vérifie que les prédicats dans les littéraux ont été définis dans la BC ou correspondent aux prédicats prédéfinis dans le langage. La clause "vérif-sémantique-cont" vérifie que la CI sous forme clausale est cohérente avec les CIs déjà existantes dans la BC. La clause "traiter-contrainte" stocke dans la BC la contrainte d'intégrité sous les formes standard et clausale et, génère les contraintes opérationnelles qui ont pu être impliquées par la contrainte d'intégrité.

En général, la contrainte d'intégrité sous la forme standard est stockée comme suit :

contrainte-statique(Idenc, CIFS).

le premier argument du fait est instancié par une valeur croissante dans l'ordre d'insertion des contraintes. Cette valeur indentifie la contrainte d'intégrité. Le deuxième argument est instancié à la chaine de caractères contenant la CI sous forme standard.

La CI sous forme clausale est stockée moyennant un ensemble de règles Prolog comme suit :

contrainte (Idenc) ← (2)

A_{11} ; ....; A_{1l1}.

.

.

.

contrainte (Idenc) ←

A_{m1} ; ....; A_{mll}.

Enfin, pour tout prédicat de propriété ou de relation apparaissant dans la CI sous forme clausale, un fait de la forme suivante est généré :

contraintes-par-pred (Idenc, P).

le fait exprime que le prédicat P est restreint par la CI identifiée par la valeur Idenc.

Si la contrainte est de la forme Opent(FMAJ) , FA , FAN1 ⇒ FAN2 alors le fait suivant est généré :

contrainte-dynamique(Idenc, Opent(FMAJ), [LFA,LFAN1], LFAN2).

où LFA, LFAN1 et LFAN2 sont des listes contenant les formules de FA, FAN1 et FAN2, respectivement.

Les faits ainsi créés ont un intérêt purement déclaratif. Ils servent à informer et permettent de déterminer quelles sont les contraintes qui ont été imposées sur un prédicat donné ou dans quelles contraintes apparaît par exemple un prédicat donné. La vérification d'une contrainte d'intégrité est faite en vérifiant les contraintes opérationnelles qui lui sont associées.

Exemple 17

Supposons que l'on veuille imposer la contrainte suivante sur la relation "affectation" : *"Tout employé E affecté à un département D1 ne peut pas être affecté à un autre département D2"*.

Cet énoncé exprime une dépendance fonctionnelle pour la relation "affectation" entre le Nemp et le Ndep (Nemp → Ndep). Cette contrainte est écrite sous forme de clause comme suit :

(affectation (Nemp,Ndep1,_), affectation (Nemp,Ndep2,_)) ⇒ Ndep1=Ndep2

Pour insérer cette contrainte dans la BC, l'interface d'accès aux connaissances effectue l'appel suivant :

```
← contrainte-statique (insérer, "(∀Nemp)(∀Ndep1)
        [affectation (Nemp, Ndep1, _) ∧
         affectation (Nemp, Ndep2, _) ⇒
         =(Ndep1, Ndep2)] ").
```

Cette requête est alors unifiée avec la méta-règle [5] et s'il n'y a aucune incohérence les faits et la règle suivants sont insérés dans la BC :

```
contrainte-statique ("Idenc", "(∀Nemp)(∀Ndep1).....").
contrainte-par-pred ("Idenc", affectation).
contrainte ("Idenc") ←
        not(affectation(Nemp,Ndep1,_));
        not(affectation(Nemp,Ndep2,_));
        =(Ndep1,Ndep2).
```

Exemple 18

Supposons que l'on a défini l'ensemble des propriétés des "programmeurs" par un prédicat de propriété qui groupe les caractéristiques suivantes : être un employé de numéro Nemp dont la catégorie = "programmeur", et avoir une classification C (=junior, senior..). Sur cette définition, on veut ensuite imposer la contrainte suivante :

Tout programmeur doit être un employé.

dans ce cas, l'appel à faire par l'interface d'accès aux BCs est :

```
← contrainte-statique (insérer, "(∀X)(∃V1)(∃V2)(∃V3)
        [programmeur(X,V1) ⇒ employé(X,V2,programmeur,V3)]")
```

dont l'évaluation génère dans la BC les faits et la règle suivants :[1]

[1] L'exécution du pas 6 de la méthode de mise en forme clausale 6 remplace toute variable quantifiée existentiellement par une variable anonyme. Par exemple $(\forall Y)(\exists Z)P(Y,Z)$ est substitué par $(\forall Y)P(Y,\_)$.

```
contrainte-statique ("Idenc", "(∀X)(∃V1)(∃V2)(∃V3)
        [programmeur(X,V1) ⇒ employé(X,V2,programmeur,V3)]").
contrainte-par-pred ("Idenc", programmeur).
contrainte-par-pred ("Idenc", employé).
contrainte("Idenc") ←
        not(programmeur(X,_)) ;
        employé(X,_,programmeur,_).
```

Exemple 19

On veut maintenant restreindre la modification du salaire d'un employé selon la contrainte suivante :

Le nouveau salaire d'un employé doit être égal à son salaire actuel augmenté de 10% du salaire moyen des employés de leur catégorie.

Cette contrainte porte sur deux états consécutifs de la BD et fait en même temps référence à un agrégat : la moyenne. Les agrégats dans SYCSLOG sont des prédicats prédéfinis profitant d'un traitement spécial lors de leur évaluation. Des valeurs cumulatives, pour tout agrégat, sont maintenues par SYCSLOG. La contrainte peut être écrite sous forme implicationnelle comme suit :

```
modifier(employé(X,_,C,S1), employé(X,_,C,S2)) ,
        'MOY'(employé(_,_,C,S), S, MOY)  ⇒  S2 = S1 + MOY/10
```

dans ce cas, l'appel fait par l'interface d'accès aux BCs est :

```
← contrainte-dynamique (insérer,
        [ [modifier(employé(X,_,C,S1), employé(X,_,C,S2)),
          'MOY'(employé(_,_,C,S), S, MOY) ],
          [ S2 = S1 + MOY/10 ] ] ).
```

dont l'évaluation génère dans la BC le fait suivant :

```
contrainte-dynamique ("Idenc",
        modifier(employé(X,_,C,S1), employé(X,_,C,S2) ),
        [ [ 'MOY'(employe(_,_,C,S), S, MOY) ], [] ],
        [ S2 = S1 + MOY/10 ] ).
```

L'adjonction de contraintes d'intégrité

La sémantique des opérations pour les contraintes d'intégrité est légèrement différente de celle des prédicats de propriété et de relation. Etant donné qu'un prédicat peut être associé à une ou plusieurs contraintes, le concepteur dispose de deux opérations **ajouter et effacer** pour ajouter et effacer respectivement une contrainte d'intégrité. Un effacement total de toutes les contraintes d'intégrité d'une BC est aussi possible.[1] Dans tous les cas, la clause vérif-syntaxe vérifie que chaque prédicat de la liste CIFC qui ne peut pas être vide, doit avoir été défini auparavant dans la BC ou être un prédicat prédéfini dans le langage. La clause vérif-sémantique-cont vérifie que l'adjonction ou l'élimination d'une CI ne met pas en contradiction les énoncés contenus dans la BC. Enfin, la clause traiter-contrainte génère (ou élimine) dans la BC la (les) contrainte(s) d'intégrité sous la forme (1) ou (2) et génère (ou élimine) en même temps les contraintes opérationnelles déduites par les dépendances implicites contenues dans la contrainte.

L'élimination de contraintes d'intégrité

L'élimination, comme nous l'avons expliqué ci-dessus, signifie l'élimination soit d'une contrainte d'intégrité (CIFC≠ vide), soit de toutes les contraintes d'intégrité créées jusque là. Bien entendu, la contrainte d'intégrité à éliminer, dans le cas d'une élimination partielle, doit exister dans la BC. La clause vérif-sémantique-cont vérifie que l'élimination de la (des) contrainte(s) n'amène pas la BC dans un état incohérent. Enfin, la clause traiter-contrainte élimine de la BC la (les) contrainte(s) d'intégrité et élimine en même temps toutes les contraintes opérationnelles qui ont pu être générées par l'insertion de cette contrainte d'intégrité.

[1] L'intérrogation d'une BC dans le seul but de recherche d'informations est gérée complètement par l'interface d'accès aux connaissances, indépendamment des règles de META-LOG.

5. Les contraintes opérationnelles

Les contraintes opérationnelles permettent de représenter des politiques d'opération spécifiques à chaque application. Certaines de ces politiques (ex. autoriser l'insertion d'un fait dans l'extension d'un prédicat donné, autoriser l'exécution des mises à jour à propager comme conséquence de l'insertion de faits, etc) sont générées à partir des contraintes d'intégrité. D'autres sont fixées directement par le concepteur de l'application. Le contrôle de la cohérence au moment de la mise à jour des données explicites (extension) dans la BD est effectué par SYCSLOG en fonction de ces politiques. Chaque donnée modifiable doit donc avoir une politique d'opération bien déterminée. Si l'évaluation d'une contrainte opérationnelle échoue, un message de violation de la cohérence de la BD est envoyé au SGBD hôte.

Les contraintes opérationnelles sont créées dans SYCSLOG selon la fonction qu'elles auront dans le système. Ces fonctions peuvent être clasées en 3 catégories différentes :

- le contrôle de propagations des mises à jour à effectuer à l'intérieur d'une transaction. Dans ce cas, les contraintes opérationnelles sont créées explicitement par le concepteur.
- le contrôle de la cohérence des données en relation à une contrainte d'intégrité. Dans ce cas, les contraintes opérationnelles correspondantes à la CI sont générées automatiquement à partir de la CI.
- les contrôles généraux, tel que le contrôle d'unicité des valeurs de clé pour les faits dans la BD et les contrôles référentiels entre les extensions différentes dans la BD. Les contraintes opérationnelles générales sont créées automatiquement lors de la création d'une BC.

Dans cette section nous nous dédions exclusivement à la définition de contraintes opérationnelles. Leur traitement lors de mises à jour de données est décrit dans la section 2 du chapitre 6.

La méta-règle de définition de contraintes opérationnelles est la suivante :

[7] **c-opérationnelle** (Op, Opent(Nompred-Termes), Type-cond, LCond,message(M)) ←
(Op=ajouter ; Op=effacer),
opérationsBD (Opent),
(Type-cond=avant ; Type-cond=après),
vérif-conditions (LCond),
vérif-sémantique-contop (Op, Opent(Nompred-Termes), Type-cond, LCond),
traiter-c-opér (Op, Opent(Nompred-Termes), Type-cond, LCond,message(M)).

Un appel à cette méta-règle exprime qu'il s'agit de l'adjonction ou de l'élimination d'une contrainte opérationnelle sur un prédicat. La contrainte est vérifiée lors des opérations du type indiqué par Opent pour les faits représentés par Nompred-Termes. Si tous les termes dans Nompred-Termes sont des variables non instanciées alors la contrainte porte sur toute l'extension du prédicat Nompred. Si parmi les termes, certains sont instanciés, alors la contrainte opérationnelle restreint uniquement la partie, dans l'extension du prédicat Nompred, définie par les termes instanciés.

opérationsBD : un ensemble de faits dont le prédicat est "opérationsBD" définit dans la BC l'ensemble des opérations possibles sur la BD. L'ensemble de base initial est :

opérationsBD(insérer).
opérationsBD(modifier).
opérationsBD(effacer).

Cet ensemble peut être augmenté ou diminué en fonction de l'application et des possibilités offertes par le SGBD hôte.

Nompred : le prédicat sur lequel porte la contrainte doit être, bien entendu, défini dans la BC.

LCond est une liste de conditions [C1a, ..., Cna] à vérifier d'après le paramètre Type-cond. Si Type-cond = avant, les conditions dans LCond sont vérifiées dans la transaction avant l'exécution de l'opération Opent sur la BD. Si Type-cond = après, elles sont vérifiées après.

M est un message à envoyer en cas de violations de la contrainte opérationnelle, c'est à dire lorsqu'au moins une des conditions dans LCond est fausse.

Lors de la vérification sémantique de la requête Opent(Nompred-Termes) toutes les contraintes opérationnelles avant sont vérifiées. Leur validité autorise l'exécution de l'opération Opent sur la BD par le SGBD. La vérification des contraintes opérationnelles après respectives est rétardée jusqu'à la fin de la transaction.[1]

```
  transaction
-----------------------------------------
DEBUT
        .
        .
        .
        <---------------------------- avant
   Opent(Nompred-Termes)
        .
        .
        .
        <---------------------------- après
FIN
-----------------------------------------
```

Les règles à insérer dans la BC par les clauses traiter-c-opér ont d'une façon générale la forme suivante :

c-opérationnelle (Id,Opent,Nompred-Termes,avant) ← C1a, ..., Cna.

c-opérationnelle (Id,Opent,Nompred-Termes,avant) ← message(M), faux.

c-opérationnelle (Id,Opent,Nompred-Termes,après) ← C1z, ..., Cqz.

c-opérationnelle (Id,Opent,Nompred-Termes,après) ← message(M), faux.

La clause traiter-c-oper affecte une identification Id à la contrainte. Si la contrainte opérationnelle est générée en conséquence de la création d'une CI, la valeur de Id est la même que pour la contrainte d'intégrité (Idenc) associant ainsi à une CI ses contraintes opérationnelles.

[1] Une transaction est une suite finie d'actions sur les objets BD qui fait passer la BD d'un état cohérent dans un autre état cohérent [ADIB 82].

5.1. Les contraintes opérationnelles créées explicitement

Ces contraintes permettent la propagation des mises à jour à l'intérieur d'une transaction. A cette fin, SYCSLOG fournit un ensemble de conditions qui sont spécifiées, lors de la création de la contrainte opérationnelle, dans la liste Lcond dans [7].

Les conditions sont vérifiées du point de vue syntaxique et sémantique par la clause vérif-conditions. Celle-ci vérifie que chacune correspond à une des catégories suivantes :

- conditions *d'existence* de données,
- conditions *d'insertion* de données,
- conditions de *suppression* de données,
- conditions de *modification* de données et
- conditions *prédéfinies*.

Les conditions d'existence de données

Ces conditions portent sur l'existence de données, qui devront exister sur l'état actuel (avant) ou sur l'état de la BD après la mise à jour (après). Une condition d'existence est écrite :

exister (Nompred-Termes)

Nompred-Termes est un arbre Prolog dont la racine est le nom d'un prédicat. Les feuilles sont des termes instanciés ou non. Au moment d'une vérification d'existence de données, la recherche de celles-ci est faite sur la BD selon les termes instanciés et provoque l'instanciation des termes libres. La recherche de données correspondantes aux contraintes opérationnelles (avant) est faite sur l'état actuel de la BD. Tandis que, celle qui correspond aux contraintes opérationnelles après l'est sur l'état de la BD juste avant la fin de la transaction.

Exemple 20

Considérons comme exemple d'utilisation des conditions d'existence de données la contrainte opérationnelle suivante :[1]

[1] La valeur qui identifie la contrainte opérationnelle est parfois omise par des raisons de commodité.

c-opérationnelle (insérer, employé(Nsc,S,C,Nom), après) ←
exister (affectation(Nsc,_,_)).

Elle établit que tout employé doit obligatoirement être affecté à un département. La vérification de cette contrainte est faite comme suit :

1) la requête d'insertion de l'employé Nsc est autorisée,

2) la condition exister(affectation(Nsc,_,_)) est vérifiée juste avant la fin de la transaction qui contient la requête. Donc, si l'employé a été affecté dans un département pendant le déroulement de la transaction, le fait affectation(Nemp,_,_) se trouve dans la BD, la condition exister(affectation(Nsc,_,_)) s'évalue à vrai et la contrainte opérationnelle est validée. Dans le cas contraire, elle est invalidée et la transaction est informée des motifs de l'échec.

Les conditions d'insertion de données

Ces conditions portent sur l'insertion de données. Une condition de ce type fait référence aux insertions qui devraient être faites par l'usager après l'exécution de la requête et avant la fin de la transaction. Une condition d'insertion est écrite :

insérer(Nompred-Termes)

Nompred-Termes est décrit tel que précédemment. Une condition d'insertion permet de :

(1) générer une requête d'insertion de données. Cette requête est récrite dans le LMD hôte et envoyée à la transaction. C'est la transaction qui décide de l'exécuter ou non. L'exploitation de cette facilité dépend du SGBD hôte qui peut gérer ou non des valeurs nulles. D'une façon générale, la plupart de ces requêtes contiennent dans leurs arguments des valeurs nulles car il est normal qu'on ne retrouve pas toute l'information nécessaire dans la requête de mise à jour initial.

(2) vérifier si des insertions de données ont été faites avant la fin de la transaction. Pour cela, l'interface LCD maintient un registre des actions exécutées après la requête qui met en cause la contrainte contenant la condition d'insertion.

Exemple 21

Pour ce type de condition, considérons l'exemple précédent mais substituons la condition d'existence par une condition d'insertion.

c-opérationnelle (insérer, employé(Nsc, S, C, Nom), après) ←
insérer (affectation(Nsc,_,_)).

Du point de vue sémantique, la contrainte est la même, c'est à dire qu'elle exprime que tout employé doit obligatoirement être affecté dans un département. Par contre, du point de vue syntaxique il y a une différence. La vérification de cette contrainte est faite comme suit :

1) comme précédemment la requête d'insertion de l'employé Nsc est autorisée,

2) une requête d'insertion du fait affectation(Nsc,_,_) dans le LMD est envoyée à la transaction. Celle-ci décide de l'exécuter ou non.

3) Juste avant la fin de la transaction, la condition insérer(affectation(Nsc,_,_)) est vérifiée. Mais, dans ce cas, le fait affectation(Nsc,_,_) n'est pas recherché dans la BD. Au contraire, le fait insérer(affectation(Nsc,_,_)) est recherché dans la transaction de vérification. Si la transaction de l'usager a effectué l'insertion explicite dans la BD du fait affectation(Nsc,_,_), il est certain que le fait insérer(affectation(Nsc,_,_)) se trouve dans la transaction de vérification (TV), donc la condition s'évalue à vrai et la contrainte est validée. Dans le cas contraire, le fait insérer(affectation(Nsc,_,_)) ne se trouve pas dans la TV, la condition s'évalue à faux et la contrainte opérationnelle est invalidée.

Les conditions d'élimination de données

Ces conditions portent sur l'élimination de données. Une condition de ce type fait référence aux éliminations qui devraient être faites par l'usager après l'exécution de la requête et avant la fin de la transaction. Une condition d'élimination est écrite :

éliminer (Nompred-Termes)

Nompred-Termes est décrit tel que précédement. Une condition de ce type est vérifiée de la même manière que les conditions d'insertion de données.

Les conditions de modification de données

Ces conditions portent sur des modifications de données. Ces sont des modifications qui doivent être réalisées par l'usager après l'exécution d'une requête donnée et avant la fin de la transaction.

modifier (Fait1, Fait2)

Fait1 est le fait à modifier suite à l'exécution de la requête de mise à jour. Fait2 est le fait modifié. De même que pour les conditions d'insertion et d'élimination, une requête de modification dans le LMD hôte est envoyée à la transaction lors du pré-traitement de la contrainte opérationnelle (après). Pour la vérification d'une condition de modification de données, l'interface logique de cohérence des données (LCD) stocke dans la transaction TV (associé à la transaction qui contient la requête qui met en cause la contrainte opérationnelle à laquelle appartient la condition de modification) un fait de la forme modifier(Terme1,Terme2). Terme1 est unifié avec Fait1. Terme2 est unifié avec Fait2. Ceci est fait lors de toute requête de modification de données. Au moment de la vérification de la contrainte opérationnelle (juste avant la fin de la transaction) le fait modifier(Terme1,Terme2) doit avoir été stocké en conséquence d'une requête de modification du fait Terme1. Si cela est, la contrainte opérationnelle est validée si la condition s'unifie avec le fait stocké. En cas contraire, elle est invalidée.

Exemple 22

Supposons que l'élimination d'un programmeur impose comme contrainte la modification de la catégorie du fait correspondant dans la classe des employés, la nouvelle catégorie étant "analyste". Cette contrainte opérationnelle est écrite comme suit :

```
c-opérationnelle (effacer, programmeur(Npgr,_), après) ←
                    modifier (employé(Npgr,_,programmeur,_),
                              employé (Npgr,_,analyste,_)).
```

Après l'élimination d'un programmeur Npgr et avant la fin de la transaction il faudrait que la transaction modifie la catégorie "programmeur" de l'employé Npgr dans la catégorie "analyste". Au moment du pré-traitement de cette contrainte, l'équivalent de la requête

```
modifier (employé(Npgr,_,programmeur,_), employé(Npgr,_,analyste,_)).
```

dans le LMD hôte est envoyé à la transaction. Dans la requête, Npgr est instancié au numéro de l'employé. Supposons d'abord que la prochaine requête dans la transaction est la requête de modification de la catégorie de l'employé Npgr. Puisqu'il s'agit d'une requête de modification, l'interface LCD stocke dans la transaction de vérification le fait suivant :

modifier(employé(Npgr,_,programmeur,_), employé(Npgr,_,analyste,_)).

Lors de la vérification de la contrainte opérationnelle :

← c-opérationnelle (effacer, programmeur(Npgr,_), après).

le seul antécédent de la contrainte s'évalue à vrai et donc la contrainte opérationnelle aussi. Supposons maintenant que la transaction n'exécute pas de modification de la catégorie de l'employé Npgr. Le fait modifier(employé(Npgr,_,programmeur,_), Fait2) n'existe pas et lors de la vérification de la contrainte, l'antécédent échoue et donc la contrainte opérationnelle aussi. Enfin, si la transaction exécute la modification de la catégorie de l'employé Npgr mais dans une catégorie différente de "analyste", l'évaluation de l'antécédent échoue et en conséquence la contrainte opérationnelle échoue aussi.

Les conditions prédéfinies

Ce sont des conditions sur les variables qui interviennent dans la définition de la contrainte. Ces conditions utilisent des prédicats prédéfinis pour les opérateurs de comparaison ($=,<,\leq,>,\geq$), agrégats ('MOY','MIN',..), etc. Les prédicats "faux" et "message" sont aussi considérés commme des conditions prédéfinies. Le premier permet d'exprimer l'interdiction absolue d'exécuter une opération donnée sur un prédicat donné dans la BD. Le deuxième permet, en présence d'une opération donnée sur l'extension d'un prédicat, d'envoyer un message à la transaction. La sémantique des opérateurs de comparaison est la même que sous Prolog sauf pour le prédicat d'égalité. Pour l'évaluation de celui-ci, il faut que les deux termes à comparer soient instanciés, sinon l'évaluation échoue.

Exemple 23

Supposons que, en attente d'une augmentation du budget du département de ventes (Ndep=3), le chef du département interdise l'arrivée de nouveaux employés dans son département. Cette contrainte peut être exprimée comme suit :

c-opérationnelle (insérer, affectation(Nsc,3,_), avant) ←
message("Interdiction temporaire de mise à jour sur
la relation affectation"),
faux.

En même temps, lors de l'augmentation du budget du département de ventes par le département du budget, on peut le notifier au concepteur par un message pour qu'il enlève l'interdiction. Puisque la contrainte fait référence à deux états de la BD, on pourrait l'exprimer en tant que contrainte d'intégrité dynamique comme suit :

modifier(département(3, ventes, Budget_act,_),
département(3, ventes, Budget_nou,_))
⇒ Budget_act < Budget_nou, message("....") .

Cette contrainte est traduite de façon automatique lors de son insertion dans la BC en :

c_opérationnelle (modifier, département(3, ventes, Budget_act,_),
département(3, ventes, Budget_nou,_), avant) ←
not(exister(département(3, ventes, Budget_act,_)));
(Budget_act < Budget_nou,
message("L'interdiction ..... doit être enlévée")).

L'interprétation de cette contrainte est la suivante : en présence d'une requête de modification du fait correspondant au département de ventes dans l'extension du prédicat département, le département de ventes doit, avant d'autoriser la modification, avoir actuellement un budget, Budget_act, lequel doit être inférieur au budget nouveau, Budget_nou. Si ces deux conditions sont vérifiées, le message est envoyé à la transaction en même temps que l'exécution de la requête est autorisée.

5.1.1. L'adjonction de contraintes opérationnelles

L'adjonction d'une contrainte opérationnelle signifie la création d'une règle pour la contrainte. Dans [7] si la liste LCond est vide, il s'agit d'une erreur. Chaque élément dans LCond est une formule atomique dont le nom du prédicat est un prédicat déjà défini dans la BC ou un prédicat prédéfini dans le langage. Ceci est vérifié par la clause vérif-conditions. La clause vérif-sémantique-contop vérifie que la nouvelle contrainte opérationnelle n'est pas en contradiction avec les contraintes opérationnelles de la BC. Pour ceci la contrainte opérationnelle est récrite sous

forme de clause comme une contrainte d'intégrité qui ne fait pas de référence aux opérations sur la BD. La clause traiter-c-oper insère la contrainte opérationnelle dans la BC avec une identification Idop. Celle-ci permet d'identifier la contrainte d'une façon unique et directe. En outre, le coût C associé à la contrainte est calculé et on stocke :[1]

coût (Idop,C).

De même, les antécédents d'une contrainte opérationnelle (avant) qui entraînent des recherches de données dans la base sont déterminés et ordonnés selon une stratégie.[2] Si A_1, A_2, ...,A_p sont ces antécédents ordonnés, le fait suivant est créé dans la BC :

antécédents-BD (Idop, [A_1, A_2, ..., A_p]).

Enfin, un dernier fait est créé (ou mis à jour s'il existe déjà) pour l'opération Opent et le fait Nompred-Termes. Ce fait met en relation les contraintes opérationnelles portant sur Nompred-Termes à vérifier avant l'exécution de l'opération Opent. Chaque contrainte opérationnelle est représentée par la valeur Idop et toutes ces valeurs sont ordonnées dans l'ordre ascendant des coûts des contraintes. Le fait est de la forme :

contrainte-oper (Opent, Nompred-Termes, [$Idop_1$,$Idop_2$,..,$Idop_n$], avant).

où, si coût(Idopi,Ci) alors $C1 \leq C2 \leq .... \leq Cn$.

5.1.2. L'élimination de contraintes opérationnelles

L'élimination d'une contrainte opérationnelle signifie l'élimination d'une ou plusieurs contraintes opérationnelles pour un prédicat donné.

Seules les contraintes opérationnelles qui ont été créées explicitement par le concepteur peuvent etre éliminées. Les contraintes opérationnelles générées en conséquence des contraintes d'intégrité ne peuvent être éliminées qu'à travers l'élimination des dites contraintes d'intégrité.

---

1 Le calcul des coûts des contraintes opérationnelles est détaillé à la section 5.2.

2 La stratégie d'ordonnancement des antécédents d'une contrainte opérationnelle (avant) est décrite à la section 5.3.

Dans [7], si la liste LCond est vide et le paramètre Type-cond=avant alors toutes les contraintes opérationnelles (avant) qui ont été insérées pour Opent(Nompred_Termes) sont éliminées, de même si Type-cond=après. Par contre, si LCond n'est pas vide, alors la contrainte définie par les éléments de la liste et le paramètre Type-cond est éliminée. Enfin, la clause vérif-sémantique-contop vérifie que les éliminations possibles n'amènent pas la BC dans un état incohérent. Si la cohérence de la BC est maintenue, la clause traiter-c-oper effectue les éliminations démandées.

5.2. Génération automatique de contraintes opérationnelles

Certaines contraintes opérationnelles sont générées automatiquement comme conséquence de dépendances créées entre des propriétés et des relations. Ces dépendances sont mises en évidence de façon implicite pour les contraintes d'intégrité référencant uniquement l'état actuel d'une BD. D'autres sont générées comme conséquences des restrictions dynamiques spécifiées par des contraintes d'intégrité référencant deux états consécutifs de la BD. Lorsqu'on définit des relations, on génère des contraintes opérationnelles pour exprimer de façon opérationnelle les liens entre le fait qui exprime la relation et les faits mis en relation. Une contrainte opérationnelle d'unicité de clé est aussi générée pour toute BC pour les ensembles de termes et pour les relations. Les contraintes opérationnelles ainsi générées ne pourront être éliminées par le concepteur que par l'élimination de la CI que leur a donné naissance.

5.2.1. Les contraintes opérationnelles référencant un seul état de la BD

Pour préciser, considérons une contrainte d'intégrité sous la forme de clause implicationnelle en général, et référencant uniquement l'état actuel de la BD :

$$A_1 , A_2 , \dots , A_n \Rightarrow B_1 , \dots , B_q$$

Soit :

- $F = \{ A_1, A_2, \dots, A_n \}$ l'ensemble de n formules atomiques qui forment l'antécédant de la contrainte d'intégrité

- $B = \{ B_1, ..., B_q \}$ l'ensemble de q formules atomiques qui forment le conséquent de la contrainte d'intégrité

- P est une partition de F, telle que :

  $P = \text{partition}(F) = \{ P_0, P_1, ..., P_k \} ; 1 \leq k \leq n$

  P_0 groupe les formules atomiques dans F dont le prédicat ne représente ni de propriétés ni de relations ;
  P_i, $(1 \leq i \leq k)$ groupe les formules atomiques dans F qui ont un même prédicat. Celui-ci représente une propriété ou une relation. C'est à dire :

  $$P_i = \{ A_j / A_j = p_i(T_1,...,T_{qi}) \text{ et } 1 \leq j \leq n \}$$

  donc :

  $$\bigcup_{i=1}^{k} P_i = F = \{ A_1, ..., A_n \} \text{ et } P_i \cap P_j = \varnothing ; \forall i \neq j$$

- Q est une partition de B faite de la même manière que P.
  Q_0 groupe les formules atomiques dans B dont le prédicat ne représente ni des propriétés ni des relations.
  Q_i , $(1 \leq i \leq w)$ groupe les formules atomiques dans Q qui ont un même prédicat. Celui-ci représente une propriété ou une relation. C'est à dire :

  $$Q_i = \{ B_j / B_j = q_i(T_1, .., T_{qi}) \text{ et } 1 \leq j \leq q \}$$

- la formule atomique $p_i(T_1,...,T_{qi})$ est équivalente à $\text{exister}(p_i(T_1,...,T_{qi}))$. Le but du prédicat exister est d'identifier les formules atomiques portant sur un prédicat de propriété ou sur une relation.

- Si S est un ensemble de formules atomiques, CS et DS dénotent respectivement la conjonction et la disjonction de ces formules atomiques, respectivement.

Critère de génération

(1) Pour tout P_i, $(1 \leq i \leq k)$ et par groupe de formules atomiques **unifiables** dans P_i générer :

c-opérationnelle (insérer, $p_i(T_1,\ldots,T_{qi})$, avant) ← (3)
not(exister(CP_1)) ; ... ; not(exister(CP_{i-1})) ;
not(exister(C(P_i - { $p_i(T_1, .., T_{qi})$ }))) ; ... ;
not(exister(CP_k)) ; not(CP_0) ;
DB_0 ; exister(DQ_1) ; ... ; exister(DQ_w).

Le nombre de contraintes à générer pour P_i est toujours inférieur ou égal à sa cardinalité. Pour éviter des rédondances une seule contrainte opérationnelle doit être générée par groupe de formules atomiques unifiables. Donc, à la limite, si toutes les formules atomiques dans P_i sont unifiables on génère une seule contrainte opérationnelle (3).

(2) Pour tout Q_i, ($1 \leq i \leq w$) et par groupe de formules atomiques unifiables dans Q_i générer :

c-opérationnelle (effacer, $q_i(T_1,\ldots,T_{qi})$, après) ← (4)
not(exister(CP_1)) ; ... ; not(exister(CP_k)) ;
not(CP_0) ;
DB_0 ; exister(DQ_1) ; ... ; exister(DQ_{i-1}) ;
exister(D(Q_i - { $q_i(T_1, .., T_{qi})$ })) ; ... ;
exister(DQ_w).

c-opérationnelle (modifier,$q_i(T_1^1,\ldots,T_{qi}^1)$,$q_i(T_1^2,\ldots,T_{qi}^2)$,après) ←
c-opérationnelle (effacer, $q_i(T_1^1,\ldots,T_{qi}^1)$, après),
c-opérationnelle (insérer, $q_i(T_1^2,\ldots,T_{qi}^2)$, avant).

Justification : La contrainte d'intégrité

$$A_1 , A_2 , \ldots , A_n \Rightarrow B_1 , \ldots, B_q$$

est équivalente à :

$$CP_1 , CP_2 , \ldots , CP_k , CP_0 \Rightarrow DQ_0 , \ldots , DQ_W$$

et en notation disjonctive à :

$$\sim CP_0 \vee \ldots \vee \sim CP_k \vee DQ_0 \vee \ldots \vee DQ_w$$

Cette expression est mise en question lorsqu'on insère un fait dans une propriété ou relation p_i ($1 \leq i \leq k$) ou lorsqu'on élimine ou qu'on modifie un fait dans une propriété ou relation q_j ($1 \leq j \leq w$). L'élimination ou modification d'un fait d'une propriété ou relation p_i ne met pas en question la contrainte puisque CP_i

devient faux donc not(CP_i) est vrai et l'expression disjonctive est vrai (vrai $\vee$? $\equiv$ vrai). De même, l'insertion d'un fait dans une propriété ou relation q_j puisque DQ_j devient vrai et donc l'expression disjonctive aussi.

Puisque $CP_i \equiv p_i(T_1,\ldots,T_{qi}) \wedge C(P_i - \{ p_i(T_1,\ldots,T_{qi}) \})$, lorsqu'on insère par exemple le fait $p_i(T_1,\ldots,T_{qi})$, la formule atomique correspondante devient vraie.

Donc, $CP_i \equiv$ vrai $\wedge$ $C(P_i - \{ p_i(T_1,\ldots,T_{qi}) \}) \equiv C(P_i - \{ p_i(T_1,\ldots,T_{qi}) \})$ et

not(CP_i) $\equiv$ not($C(P_i - \{ p_i(T_1,\ldots,T_{qi}) \})$).

Pour que l'expression disjonctive soit vraie, il faudrait que :

$$\text{not}(CP_1) \vee \ldots \vee \text{not}(CP_{i-1}) \vee \text{not}(C(P_i - \{ p_i(T_1,\ldots,T_{qi}) \})) \vee$$
$$\text{not}(CP_{i+1}) \vee \text{not}(CP_k) \vee \text{not}(CP_0) \vee DQ \equiv \text{vrai}$$

Autrement dit :

$$\begin{aligned} &\text{c-opérationnelle (insérer, } p_i(T_1,\ldots,T_{qi}), \text{ avant)} \leftarrow \\ &\quad \text{not(exister}(CP_1)) ; \ldots ; \text{not(exister}(CP_{i-1})) ; \\ &\quad \text{not(exister}(C(P_i - \{ p_i(T_1, \ldots, T_{qi}) \}))) ; \ldots ; \\ &\quad \text{not(exister}(CP_k)) ; \text{not}(CP_0) ; \\ &\quad DB_0 ; \text{exister}(DQ_1) ; \ldots ; \text{exister}(DQ_W). \end{aligned} \qquad (3)$$

La contrainte opérationnelle est vérifiée avant l'exécution de la requête d'insertion pour assurer un ordre dans la création d'informations dépendantes les unes des autres. Un fait d'une classe S inclue dans une autre classe C doit être créé si et seulement si le fait correspondant dans la classe C existe déjà.

Si l'on élimine un fait dans un prédicat q_i ($1 \leq i \leq w$),

$DQ_i \equiv q_i(T_1,\ldots,T_{qi}) \vee D(Q_i - \{ q_i(T_1,\ldots,T_{qi}) \})$ donc

$DQ_i \equiv$ faux $\vee$ $D(Q_i - \{ q_i(T_1,\ldots,T_{qi}) \})$
$\equiv D(Q_i - \{ q_i(T_1,\ldots,T_{qi}) \})$

Pour que l'expression disjonctive totale soit vraie, il faudrait que :

$\text{not}(CP_0) \vee \ldots \vee \text{not}(CP_K) \vee$
$DQ_0 \vee \ldots \vee D(Q_i - \{ q_i(T_1,\ldots,T_{qi}) \}) \vee \ldots \vee DQ_W \equiv$ vrai

d'une autre manière :

c-opérationnelle (effacer, $q_i(T_1,\ldots,T_{qi})$, après) ← (4)
not(exister(CP_1)) ; ... ; not(exister(CP_k)) ;
not(CP_0) ;
DB_0 ; exister(DQ_1) ; ... ; exister(DQ_{i-1}) ;
exister($D(Q_i - \{ q_i(T_1, .., T_{qi}) \})$) ; ... ;
exister(DQ_W).

La contrainte opérationnelle est vérifiée après l'exécution de la requête d'élimination pour assurer l'élimination d'informations dépendantes les unes des autres. Un fait d'une classe C peut être éliminé à tout moment si et seulement si tous les faits correspondants dans les sous-classes S sont éliminés après l'élimination du fait concerné et avant la fin de la transaction qui exécute la requête (qui met en cause la contrainte opérationnelle).

La modification d'un fait dans un prédicat de propriété ou de relation q_i ($1 \leq i \leq w$) peut être considérée comme équivalente à l'élimination du fait à modifier suivie de l'insertion du fait modifié. Donc, si $q_i(T_1^1,\ldots,T_{qi}^1)$ est le fait à modifier et $q_i(T_1^2,\ldots,T_{qi}^2)$ est le fait modifié :

c-opérationnelle (modifier, $q_i(T_1^1,\ldots,T_{qi}^1)$, $q_i(T_1^2,\ldots,T_{qi}^2)$, après) ← (5)
c-opérationnelle (effacer, $q_i(T_1^1,\ldots,T_{qi}^1)$, après),
c-opérationnelle (insérer, $q_i(T_1^2,\ldots,T_{qi}^2)$, avant).

Exemple 24

Soit la contrainte d'intégrité sur "programmeur" sous la forme implicationnelle :

programmeur(X,_) ⇒ employé(X,_,programmeur,_)

donc, F = { programmeur(X,_) }
P_0 = ∅
P_1 = { programmeur(X,_) }
B = { employé(X,_,programmeur,_) }
B_0 = ∅ et B_1 = B

L'application du critère provoque la création des contraintes opérationnelles suivantes :

c-opérationnelle (insérer, programmeur(X,_), avant) ←

not(exister(programmeur(X,_)));
exister(employé(X,_,programmeur,_)).

Ceci veut dire qu'avant toute insertion d'un fait dans l'extension du prédicat programmeur, il faut qu'il existe le fait correspondant dans l'extension des employés. Autrement dit, l'existence d'un fait dans programmeur n'est pas possible sans l'existence d'un fait correspondant dans employé.

c-opérationnelle (effacer, employé(X,_,programmeur,_), après) ←
not(exister(programmeur(X,_)).

De même, après toute élimination d'un fait de l'extension du prédicat employé et dont la catégorie est égal à "programmeur", il faut qu'avant la fin de la transaction (qui contient la requête de mise à jour qui met en cause la contrainte) le fait correspondant à l'employé dans l'extension du programmeur n'existe pas. Autrement dit, si celui-ci existe il faut l'éliminer avant la fin de la transaction.

Bien entendu, après la modification de la catégorie de 'programmeur' pour un employé X, le programmeur X ne doit pas exister comme tel dans la BD.

c-opér(modifier,employé(X,_,programmeur,_),employé(X,_,analyste,_),après) ←
c-opérationnelle (effacer, employé(X,_,programmeur,_), après),
c-opérationnelle (insérer, employé(X,_,analyste,_), avant).

Exemple 25

Considérons maintenant la contrainte d'intégrité qui exprime une dépendance fonctionnelle pour la relation "affectation" entre le Nemp et le Ndep.

(affectation(Nemp,Ndep1,_) , affectation(Nemp,Ndep2,_)) ⇒ Ndep1 = Ndep2

Donc, F = { affectation(Nemp,Ndep1,_),affectation(Nemp,Ndep2,_) }
P_0 = ∅
P_1 = { affectation(Nemp,Ndep1,_),affectation(Nemp,Ndep2,_) }
B = B_0 = { Ndep1 = Ndep2 }

L'application du critère provoque la création d'une seule contrainte opérationnelle :

c-opérationnelle (insérer, affectation(Nemp,Ndep1,_), avant) ←
not(exister(affectation(Nemp,Ndep1,_))) ;
not(exister(affectation(Nemp,Ndep2,_))) ;
= (Ndep1,Ndep2).

Cette contrainte échoue chaque fois qu'on veut affecter un employé Nemp dans un département Ndep1 et que cet employé est déjà affecté dans un département Ndep2 différent de Ndep1.

5.2.2. Les contraintes opérationnelles pour les restrictions dynamiques

Les contraintes opérationnelles générées dans ce cas permettent de vérifier les contraintes imposées entre deux états consécutifs d'une BD. Pour préciser, considérons une contrainte d'intégrité sous la forme de clause telle que nous l'avons décrit dans (1) à la section 4.2 :

$$\text{Opent}(\text{prédicat}(T_1,\ldots,T_n)) \wedge (FA_1 \wedge \ldots\ldots \wedge FA_m) \wedge (FAN_{11} \wedge \ldots\ldots \wedge FAN_{1q}) \Rightarrow FAN_{21} \wedge \ldots\ldots \wedge FAN_{2p}$$

La signification d'une telle contrainte est la suivante :

avant d'effectuer l'opération Opent pour le fait prédicat($T_1,\ldots,T_n$), si les conditions $FA_1,\ldots,FA_m$ et $FAN_{11},\ldots,FAN_{1q}$ portant sur les états actuel et nouveau respectivement sont vraies, alors les conditions $FAN_{21},\ldots,FAN_{2p}$ qui restreignent le changement d'état doivent être vraies aussi.

Supposons que les i premières FA portent sur des ensembles de propiétés ou sur des relations et que les (m-i) restantes ont comme prédicat des prédicats prédéfinis dans le langage restreignant les variables dans ces formules atomiques. Effectuons les mêmes suppositions pour les k premières FAN_2.

Pour la vérification d'une telle contrainte on suppose que l'opération Opent pour le fait prédicat($T_1,\ldots,T_n$) a été exécutée. Donc, la formule atomique Opent(prédicat($T_1,\ldots,T_n$)) est toujours vraie et représente l'état nouveau dans la vérification de la contrainte. Autrement dit, vu la sémantique des contraintes opérationnelles et de l'implication logique :

c-opérationnelle (Opent, prédicat(T_1,..,T_n), avant) ← (6)
not(exister(FA_1)) ;....; not(exister(FA_i)); not(FA_{i+1}) ;
..... ; not(FA_m) ;
not($FAN1_{11}$) ;...; not(FAN_{1q}) ;
(exister(FAN_{21}) ,..., exister(FAN_{2k}), exister(FAN_{2k+1}) ,
....., FAN_{2p}).

Lorsque l'opération Opent est une modification, cette règle a quatre arguments à la place de trois. Le deuxième et troisième argument correspondent au fait à modifier et au fait modifié respectivement. La règle permet aussi de vérifier que le fait à modifier existe sur l'état actuel.

Exemple 26

Supposons que sous une application dans laquelle des employés font partie de l'environnement, on veut établir la politique suivante par rapport aux salaires des employés :

Le salaire d'un employé doit toujours augmenter entre 5% et 25% du salaire actuel.

D'un point de vue opérationel, cette politique peut se traduire par : premièrement, l'employé doit avoir actuellement un salaire susceptible d'être modifié, et deuxièmement, le nouveau salaire doit être égal à l'actuel augmenté d'un facteur compris entre 5% et 25%. La contrainte est écrite comme suit :

modifier (employé(N,_,_,Sa), employé(N,_,_,Sn))
⇒ Sn > 1.05*Sa ∧ Sn < 0.26*Sa

laquelle est traduite pour sa vérification opérationnelle dans :

c-opérationnelle (modifier,employé(N,Sa,_,_),employé(N,Sn,_,_), avant) ←
not(exister((employé(N,Sa,_,_)));
(>(Sn,*(1.05,Sa)) ,
<(Sn,*(0.26,Sa))).

Au moment de la vérification de cette contrainte, les variables N (le numéro de l'employé) et Sn (son salaire) se trouvent instanciées respectivement au numéro de l'employé et à la valeur du nouveau salaire. La clause "exister" déclenche la recherche de données pour l'employé N est c'est ainsi que la variable Sa est

instanciée au salaire actuel de l'employé. Une fois Sn et Sa instanciées, les conditions sont vérifiées. Si elles évaluent à "vrai" la modification est autorisée, en cas contraire, la modification est interdite par SYCSLOG.

5.2.3. La contrainte opérationnelle générale d'unicité de clé

Cette contrainte est une contrainte générale d'unicité de clé pour toutes les propriétés et relations dont les extensions se trouvent stockées dans la BD. Elle est générée automatiquement lors de la création de toute BC.

```
c-opérationnelle (insérer, Pred_Termes, avant)  ←                (7)
                    functor (Pred_Termes,Nompred,_),
                    key (Pred_Termes, Fclé),
                    not(exister(Fclé)).
```

La clause de nom "key" comporte un ensemble de faits qui sont générés à chaque insertion de prédicats dans la BC. Le but de chaque fait ainsi généré est de permettre l'identification des valeurs clés d'un fait pour un prédicat donné. Par exemple, pour le prédicat employé, le fait généré est le suivant :

key(employé(Nsc,_,_,_),employé(Nsc,_,_,_)).

La contrainte (7) est vérifiée lors de l'insertion d'un fait dans l'extension du prédicat "employé". Le premier terme du fait est ainsi unifié avec le fait employé à insérer (Pred_Termes). La variable clé Nsc est instanciée et les autres attributs non clés restent des variables anonymes sans importance. Le deuxième terme Fclé est le résultat de l'appel, c'est à dire le fait à rechercher dans la BD.

5.2.4. Les contraintes opérationnelles générées pour les relations

On met en relation des ensembles de termes. Ces ensembles sont représentés dans la BC par des prédicats de relation. Au moment de mettre en relation des termes ayant été liés par un prédicat, il faut non seulement assurer que les ensembles existent mais également que les termes leurs appartiennent. Les contraintes opérationnelles générées dans ce cas ont, en général, la même signification que les contraintes référentielles dans une base de données relationnelle. L'intégrité référentielle a été largement étudiée par Codd [CODD 79] et par Date [DATE 81]. D'une façon générale une contrainte référentielle, dans le modèle relationel, est

présente lorsqu'une relation fait référence à une autre relation. Par exemple, la relation EMPLOYE fait référence à la relation DEPARTEMENT via l'attribut #dep. Toute valeur de l'attribut #dep dans la relation EMPLOYE doit apparaître dans un n-uplet de la relation DEPARTEMENT.

Supposons qu'on ait une relation qui associe deux ensembles de termes. (De la même façon ce qu'on établit est étendu pour plus de deux ensembles de termes).

La clause suivante doit se vérifier pour la relation :

$$\text{relation}(K_1,K_2,....) \Rightarrow \text{prédicat}_1\ (K_1,....) \wedge \text{prédicat}_2\ (K_2,....)$$

laquelle est logiquement équivalente à :

$$\text{relation}\ (K_1,K_2,\ ...) \Rightarrow \text{prédicat}_1\ (K_1,\ ...) \quad \text{et}$$
$$\text{relation}\ (K_1,K_2,\ ...) \Rightarrow \text{prédicat}_2\ (K_2,\ ...)$$

où K_1 et K_2 sont les caractéristiques clés pour les faits des extensions des prédicats 1 et 2. La clause exprime ce qui suit : si les faits représentés par les valeurs K_1 et K_2 des caractéristiques clés sont mis en relation (moyennant la relation de nom "relation"), alors des faits correspondants doivent avoir une existence indépendante dans les extensions des prédicats qui les définissent.
Si l'on applique le critère de génération de contraintes opérationnelles pour ces clauses, on obtient pour la première :

$$\text{c-opérationnelle (insérer, relation}(K_1,K_2,...)\text{, avant)} \leftarrow$$
$$\text{exister (prédicat}_1(K_1,...)).$$

Cette contrainte établi qu'avant de mettre en relation deux faits représentés par K_1 et K_2, il faut qu'il existe le fait correspondant à K_1 dans l'extension du prédicat 1. De façon similaire, on obtient pour la deuxième clause :

$$\text{c-opérationnelle (insérer, relation}(K_1,K_2,...)\text{, avant)} \leftarrow$$
$$\text{exister (prédicat}_2(K_2,...)).$$

L'autre contrainte générée est :

$$\text{c-opérationnelle (effacer, prédicat}_1(K_1,...)\text{, après)} \leftarrow$$
$$\text{not(exister(relation}(K_1,K_2,...)).$$

Celle-ci établit qu'après l'élimination d'un fait de l'extension du prédicat$_1$ il faut qu'il n'existe pas (avant la fin de la transaction) le fait correspondant à relation$(K_1,K_2,..)$.

On procède de la même manière pour la deuxième clause.
La contrainte opérationnelle correspondante à l'opération "modifier" n'a pas de sens car on ne peut pas modifier les valeurs clés d'un fait.

Exemple 27

Considérons la relation affectation de l'exemple précédent. Pour cette relation on a les clauses :

affectation(Nemp,Ndep,_) ⇒ employé(Nemp,_,_,_)
affectation(Nemp,Ndep,_) ⇒ département(Ndep,_,_)

Donc, pour cette relation les instances des contraintes opérationnelles ci-dessus sont :

c-opérationnelle (insérer, affectation(Nemp,Ndep,_), avant) ←
exister (employé(Nemp,_,_,_)).

Cette contrainte exprime que l'on ne peut pas affecter un employé dans un département si cet employé n'a pas été déclaré comme tel.

c-opérationnelle (effacer, employé(Nemp,_,_,_), après) ←
not(exister(affectation(Nemp,Ndep,_))).

Celle-ci exprime qu'un employé ne peut pas être affecté dans un département s'il n'est plus un employé de l'entreprise.
De même pour la deuxième clause, les contraintes opérationnelles correspondantes expriment :
1) que l'on ne peut pas affecter un employé dans un département si celui-ci n'a pas été déclaré comme tel; et
2) qu'un département ne peut pas être éliminé tant qu'il y a des employés affectés au département.

- En résumé, les contraintes opérationnelles permettent, comme leur nom l'indique, de vérifier des contraintes au niveau opérationnel. Elles permettent de traiter d'une façon uniforme toute sorte de dépendance - implicite ou explicite -, des restrictions dynamiques et enfin des restrictions d'unicité de clé pour l'ensemble des propriétés et les relations d'une application donnée. En outre, elles évoluent directement ou indirectement en fonction de l'évolution des contraintes d'intégrité

qui leur ont partiellement donné naissance.

5.3. Calcul des coûts

Lors d'une vérification de contraintes opérationnelles, il est utile de commencer par vérifier les contraintes les moins coûteuses de manière à provoquer plus rapidement un échec en cas de violation de la cohérence. Dans ce but, un facteur de coût est affecté à chaque contrainte opérationnelle lors de sa création en fonction d'une connaissance empirique des coûts de chacune des opérations élémentaires qu'elle doit effectuer. Les opérations choisies dans notre cas sont le transfert de données vers la mémoire centrale et le temps calcul nécessaire au traitement sur ces données.

On peut remarquer que les contraintes opérationnelles (après) n'effectuent pas de vérifications sur les données de la BD. Le calcul de leur coût n'est donc pas nécessaire pour réaliser une optimisation importante. Nous ne traitons donc de cette façon que les contraintes opérationnelles avant.

Soit $S = \{ C_1, C_2, \ldots, C_n \}$ l'ensemble des contraintes opérationnelles mises en cause par une requête donnée R. Si R est valide, le coût de S est égal à l'addition des coûts de chaque contrainte, puisque dans ce cas, on évalue toutes les contraintes.

$$\text{coût}(S/R=\text{valide}) = \text{coût}(C_1) + \ldots. + \text{coût}(C_n)$$

mais par contre, si la requête est invalide le coût de S est égal à l'addition des coûts des k contraintes vérifiées jusqu'à la détection de l'invalidité. Autrement dit :

$$\text{coût}(S/R=\text{invalide}) = \sum_{j=1}^{k} \text{coût}_j \quad \text{et} \quad \forall j \; \exists! i \; / \; \text{coût}_j = \text{coût}(C_i) \quad \text{et} \quad 1 \leq i \leq n$$ [1]

Considérons le cas où la requête R est valide. Pour minimiser le coût de S il suffirait de minimiser isolémment le coût de chaque contrainte. Comme on l'a signalé ci-dessus, ce coût dépend directement du nombre de faits N_i à transférer en mémoire principale et du temps de CPU T_i d'évaluation de la contrainte :

$$\text{coût (Ci)} = (M^*N_i) + (R^*T_i)$$

- M est le coût de transfert un fait en mémoire principale,

[1] $\exists! i$ = existe une seule valeur i.

- R est le coût d'utilisation d'une seconde de CPU
- T_i, le temps de CPU d'évaluation de C_i, dépend du nombre de faits examinés NF_i pendant la résolution, et de la compléxité p_i de la contrainte.

  $T_i = T * NF_i * p_i$ où $p_i \geq 1$ et $0 \leq NF_i \leq N_i$
- T est le temps en secondes pris par le moteur d'inférence pour examiner un fait.

Si nous reprenons la formule du coût d'une contrainte :

$$\text{coût (Ci)} = (M * N_i) + (R * T * p_i * NF_i)$$

puisque la compléxité pi de la contrainte C_i est intrinsèque à celle-ci, c'est à dire, qu'on ne peut pas la diminuer (cela relève du concepteur de l'application lors de la conception), on atteint le coût minimal lorsque le nombre de faits à amener en mémoire est juste la quantité minimal de faits nécessaires à l'évaluation de la contrainte. C'est à dire :

$$\min(\text{coût}(C_i)) = K * (M+R+T+p_i) \quad \text{et} \quad K = \min(N_i) = \min(Nf_i) \qquad (8)$$

Une partie de notre stratégie devrait donc consister à atteindre ce nombre minimal de faits nécessaires à l'évaluation de chaque contrainte. Nous verrons que ceci est assuré en déterminant un ordre pour les antécédents de la contrainte Ci.

Maintenant, nous allons analyser le cas où la requête R est invalide. Le coût minimal est atteint en fixant un ordre d'évaluation des contraintes (puisqu'on ne sait pas à priori que la contrainte C_k sera invalidée), et cet ordre "coûte" le moins cher possible. Ceci est atteint lorsque l'ordre est ascendant par coût de contrainte et le coût de chaque contrainte est aussi minimal. C'est à dire :

$$\text{coût (S/R=invalide)} = \sum_{j=1}^{k} \text{cout}_j \; ; \; 1 \leq k \leq n \quad \text{et} \quad \text{coût}_1 \leq \text{coût}_2 \leq \ldots \leq \text{coût}_k$$

où $\forall j \; \exists! i \; / \; \text{coût}_j = \min(\text{coût}(C_i)) \quad \text{et} \quad 1 \leq i \leq n$

Ceci est cohérent avec notre résultat, dans le cas où R est valide. La stratégie en sa totalité doit assurer que les contraintes opérationnelles à évaluer pour une requête donnée sont classées et par la suite évaluées dans l'ordre ascendant du coût.

Donc, le coût minimal d'évaluation d'une contrainte est donné par la formule (8) :

$$\min(\text{coût}(Ci)) = K * (M+R+T+pi) \quad \text{et} \quad K = \min(Ni) = \min(NFi)$$

Les seuls facteurs qui dépendent de la contrainte sont N_i ou NF_i, le nombre de faits à amener en mémoire principale et le nombre minimal de faits nécessaires à l'évaluation de la contrainte. L'autre facteur est, bien entendu, la compléxité pi de la contrainte. Le premier dépend de l'état sur lequel se trouve la BD lors d'une vérification de la contrainte. Nous prenoms comme état représentatif de la BD un échantillon de la BD (décrit au chapitre suivant) et ce facteur est considéré comme le nombre maximal de faits nécessaires à la vérification de la contrainte. En conséquence, les coûts ainsi obtenus ne correspondent pas aux coûts réels d'évaluation mais seul l'ordre des coûts intéresse.

Le deuxième facteur, pi, est estimé en fonction du nombre de prédicats impliqués dans la contrainte. La compléxité d'évaluation d'un prédicat prédéfini est insignifiante près de celle du reste de prédicats. La compléxité d'évaluation d'un prédicat j peut être évaluée en considérant ses termes. Si NV_j est le nombre de termes variables, NVA_j le nombre de variables anonymes et NTI_j le nombre de termes instanciés, la compléxité d'évaluation du prédicat j peut être mesurée par la formule :

$$p_j = \alpha_1 * NV_j + \alpha_2 * NVA_j + \alpha_3 * NTI_j$$

où $1 > \alpha_1 > \alpha_2 > \alpha_3 > 0$. La compléxité de la contrainte Ci est donc déterminée par l'addition des facteurs de compléxité d'évaluation pour les prédicats intervenant dans Ci.

Les paramètres M, R et T dépendent du système d'exploitation hôte et du moteur d'inférence. Ils doivent être initialisés lors de la création d'une BC. C'est ainsi qu'une fois initialisée une BC elle contient des faits de la forme :

paramètre (m,M).
paramètre (r,R).
paramètre (t,T).

où les variables M, R et T seront instanciées respectivement par le coût de transfert d'un fait en mémoire principale, le coût d'utilisation d'une seconde de CPU et le temps en secondes pris par l'interpréteur pour examiner un fait. Le coût ainsi obtenu, est le coût minimal d'évaluation de la contrainte sur l'échantillon à un moment donné. Il est stocké dans la BC sous la forme :

coût (Idop,Ci).

lors de la création de toute contrainte opérationnelle (avant) Ci.

5.4. La stratégie d'ordonnoncement d'antécédents d'une contrainte

Lorsrqu'une contrainte opérationnelle a besoin de données de la base pour son évaluation, elle est traitée lors de sa création. Tout d'abord, les instanciations possibles avec les données de la requête à vérifier sont effectuées symboliquement à l'intérieur de la contrainte. Ensuite, les antécédents de la contrainte qui entraînent une recherche de données sont déterminés. S'il y en a quelques uns redondants, on ne les prend pas en compte pour la recherche de données. On utilise la stratégie suivante pour déterminer dans le LMD hôte l'ordre de réécriture de ces antécédents.

(1) un antécédent A1 avec l'attribut clé instancié est prioritaire sur un antécédent A2 dans lequel il ne l'est pas. Si les deux antécédents ont les attributs clés instanciés ils ont la même priorité.

(2) un antécédent A1 qui a moins d'attributs non instanciés qu'un autre antécédent A2 est prioritaire. S'ils ont le même nombre d'attributs non instanciés ils ont la même priorité.

Ainsi, le fait antécédants_BD(Idop,[A_1,A_2,...,A_p]) est stocké pour la contrainte Idop. Il contient l'ordre déterminé pour les antécédants.

La réécriture de la requête de recherche de données est faite comme suit :

- les antécédents sont récrits dans l'ordre déterminé.
- si un antécédent Ai a des attributs à instancier avec des données de la base et que cet attribut a déjà été instancié dans une recherche antérieure Aj (j<i), alors l'antécédent Ai est récrit en requête de recherche de données avec la restriction déterminée par l'instanciation faite par Aj.

Une fois la requête récrite elle est stockée sous forme d'un fait de la forme : réécrire_antécédants_BD(Idop,RV). Celui-ci est utilisé lors de la vérification de la contrainte opérationnelle.

CHAPITRE 5

LE MODULE LOGIQUE DE VERIFICATION SEMANTIQUE

CHAPITRE 5

LE MODULE LOGIQUE DE VERIFICATION SEMANTIQUE

La vérification sémantique est effectuée par SYCSLOG à deux niveaux. Le premier niveau, qui assure la cohérence des définitions de données, est mis en oeuvre par un ensemble de procédures logiques chargées de traiter les opérations de modification des BCs des applications. Le deuxième niveau, le niveau de cohérence de données, est assuré par des procédures logiques qui, en utilisant les informations des BCs, vérifient toute requête de modification des données propres aux applications.

1. Vérification sémantique des définitions de données

Du point de vue de la logique, une BC comporte un ensemble de clauses obtenu par :

1) L'écriture sous forme de clauses :
   a) des axiomes d'unicité de propriété, de structure, de contenu, de déduction et de base disjonctifs dans la théorie du 1^{er} ordre de l'application.
   b) des f.b.f fermées sous la forme clausale ou contraintes d'intégrité.
   La cohérence d'une BC traite de la cohérence de cet ensemble de clauses.

2) La génération de contraintes opérationnelles à partir des contraintes d'intégrité. Lorsque l'ensemble de clauses correspondant à une CI sous la forme standard est cohérent avec la BC, la CI est traduite dans un ensemble de contraintes opérationnelles logiquement équivalentes à la CI sous forme standard.

Pour assurer que les énoncés propres à l'application seront satisfaits sur des états de la BD, il faut assurer la cohérence de sa BC. Ces états sont nommés, par définition, des **états valides** de la BD.

1.1. La cohérence d'une base de connaissances

Un *ensemble de clauses est cohérent* s'il y a au moins une interprétation sous laquelle toutes les clauses de l'ensemble sont vraies [KOWA 79]. Une interprétation est faite par rapport à un domaine qui définit l'ensemble de termes pour lesquels les clauses ont un sens. D'après ceci, nous pouvons définir une BC cohérente comme suit :

Définition 1

Une *BC est sémantiquement cohérente sous D* $\Leftrightarrow$

$$\exists I(D)\ \forall \text{clause } w \{ w \in BC \Rightarrow (D_I \neq \varnothing \wedge v_I(w)) \}$$

D est le domaine constitué par l'ensemble de termes constants définis par les axiomes d'unicité de propriété dans la BC. $I(D)$ est une interprétation dont le domaine est l'ensemble D. Une interprétation sous un domaine D est la paire (D, Mc), où D est le domaine d'interprétation et Mc sont les assignations aux constantes de w. $v_I(w)$ est l'évaluation de la formule w sous l'interprétation I •

Dans la logique des prédicats, une formule w est *satisfaisable (réalisable)* sous un domaine D si et seulement s'il existe une interprétation du domaine D sous laquelle l'évaluation de w est vraie. Donc, nous pouvons dire d'une façon équivalente qu'une BC est sémantiquement cohérente si et seulement si toutes ses clauses sont satisfaisables sous un domaine D.

Nous remarquons que l'ensemble des constantes de n'importe quel état de la BD est inclus dans le domaine D car il est complètement défini par les axiomes d'unicité de propriété de la BC.

L'explicitation d'un domaine D n'est pas toujours possible. Celui-ci peut être de grand taille ou même infini, par exemple lorsque le prédicat *entier* intervient dans la définition d'une propriété. Notre intérêt est donc de trouver un mécanisme qui permette de déterminer, lors de l'insertion d'un énoncé W dans une BC cohérente, s'il est ou non satisfaisable sur le domaine actuel D. Pour ceci, un domaine D^* est explicité de façon à ce que la BC est satisfaisable sur D si et seulement si elle l'est sur D^*.

1.2. Théorème de l'Echantillon

Une formule w est satisfaisable sous le domaine D associé à la BC si et seulement s'il existe une interprétation I^* sous le sous-domaine D_{Echant} déterminé par l'ensemble de constantes qui apparaissent dans un échantillon de la BD ou qui sont générées pendant l'interprétation, telle que l'évaluation de la formule sous cette interprétation est vraie. C'est à dire :

w est satisfaisable sous $D(BC)$

$$\Leftrightarrow \exists I^*(D_{Echant})[D_{Echant} \neq \varnothing \wedge D_{Echant} \subseteq D(BC) \wedge v_{I^*}(w)\ ] \bullet$$

La validité de cette théorème est basée sur la construction d'un échantillon de la BD et sur la possibilité de regrouper en classes des éléments d'un certain domaine D tel que des éléments appartenant à une même classe soient équivalents, du point de vue logique, pour l'évaluation d'une formule w. L'échantillon ainsi construit détermine un domaine D_{Echant} contenant un ou plusieurs éléments représentatifs par classe.

Avant de donner la démonstration du théorème de l'échantillon nous allons d'abord détailler la méthode de construction et d'exploitation d'un échantillon de la BD et l'algorithme pour déterminer si une formule w est satisfaisable sur un échantillon de la BD.

1.3. Construction de l'échantillon de la BD (Echant)

L'échantillon d'une BD comporte un ensemble de faits positifs et négatifs représentatifs respectivement des BD^+ et BD^-. Ces faits satisfont les énoncés de définition donnés par le concepteur.

L'échantillon d'une BD est construit par l'interface d'accès aux connaissances (vue concepteur) au fur et à mesure que la BC évolue.

1) Lors de l'insertion de prédicats de propriété définis en compréhension (lesquels auront une extension dans la BD), l'interface génère un (1) fait qui constituera la classe pour le prédicat et qui est inséré dans l'échantillon. Un fait est le nombre minimun mais cet ensemble peut être augmenté selon le nombre de relations définies en fonction du prédicat. Les valeurs pour les caractéristiques clés uniquement sont générées, les autres caractéristiques correspondantes

aux faits restent variables.

2) Lors de l'insertion d'une contrainte d'intégrité, l'interface détermine pour chaque prédicat qui intervient dans la contrainte, le nombre minimal de faits qui permettra de vérifier la satisfaisabilité de la contrainte (si elle est satisfaisable bien entendu). Si ce nombre est supérieur au nombre de faits existants dans l'échantillon pour le prédicat, le système génère la différence. S'il est inférieur le système ne génère rien. Ainsi, au moment de la vérification de la contrainte il y a dans l'échantillon autant de faits qu'il est nécessaire pour que les contraintes existantes et celle qu'on est en train d'insérer puissent être vérifiées.

3) Dans tous les autres cas il n'y a pas de génération de faits car ils peuvent être déduits à tout moment de l'échantillon de la BD et de la BC.

Pour la construction de cet échantillon, l'interface d'accès aux BCs utilise un ensemble de règles de dérivation. Celles-ci sont exploitées en génération pour générer explicitement les faits de l'échantillon.

Le contrôle du nombre de faits par prédicat est maintenu grâce à l'utilisation des clauses suivantes :

(1) quantfaits ($Idop_i$, $[p_1^i(N_1^i),\ldots,p_n^i(N_n^i)]$).

(2) quantfaits ('Max', P, Np).

Une clause du type (1) groupe, pour toute contrainte identifiée par $Idop_i$, les différents prédicats de propriété (P_j^i) apparaissant dans la contrainte de façon directe ou indirecte (par une relation) et le nombre minimal de faits N_j^i nécessaires pour le prédicat pour la vérification de la contrainte. Ce nombre est déterminé syntaxiquement à partir des variables clés qui apparaissent dans la contrainte. Chaque variable différente représente un fait dans l'extension du prédicat de propriété correspondant. Par exemple, pour une contrainte de transitivité pour une relation R pour les termes définis par P_1 ($R(X,Y) \wedge R(Y,Z) \Rightarrow R(X,Z)$), le nombre minimal de faits pour P_1 est 3 (dû aux variables X, Y et Z). Une clause du type (2) maintient le nombre maximum de faits Np pour un prédicat P entre toutes les quantités minimales qui se trouvent dans les clauses du type (1).
Lors de modifications de contraintes ces deux faits sont mis à jour.

1.4. Exploitation de l'échantillon de la BD (Echant)

Un échantillon ainsi construit contient à tout moment la quantité minimal de faits nécessaires à la vérification des énoncés de définition de la BC. Il est intérrogé lors de toute modification de la BC.

Soit :

E(P,Mode) : la clause de définition en compréhension du prédicat de propriété ou de relation P dont le Mode est égal à *ext* ou *ded.*

$P(t_1,\ldots,t_n)$: le littéral positif dans E(P,Mode). Celui-ci est toujours précédé par les mots *compréhension* ou *relation* selon le type du prédicat P.

C_i : la contrainte i-ème sous forme de clause.

{ Echant } : représente la conjonction de toutes les clauses dans l'échantillon.

{ BC } : représente la conjonction de toutes les clauses dans la base de connaissances.

La vérification sémantique est effectuée comme suit :

a) si **inserer**(E(P,Mode)) alors effectuer l'insertion de E(P,Mode) et montrer qu'il existent des termes $t_1, \ldots, t_n$ qui permettent que le prédicat P s'évalue à vrai. Du point de vue du démonstrateur de théorèmes il s'agit de montrer l'insatisfaisabilité de l'ensemble de clauses :

$$\{ \text{Echant} \} \cup \{ \text{BC} \} \cup \{ \text{E(P)} \} \cup \{ \sim P(t_1,\ldots,t_n) \}$$

Si l'ensemble de clauses est insatisfaisable alors l'insertion est confirmée, en cas contraire elle est annulée.
La vérification effectuée est indépendante du monde du prédicat P (fermé ou ouvert). Elle est faite toujours sur les faits positifs et dans le cas d'un monde ouvert la satisfaisabilité de l'énoncé est étendue aux faits négatifs. Si l'énoncé est satisfaisable pour le prédicat P il le sera également pour un prédicat Q = not_P.

b) si **inserer**(C_i) alors effectuer l'insertion de la Ci (sous la forme $C_i \Rightarrow Q_i$, où Q_i = contrainte(I_i)) et montrer qu'il existent des termes, correspondants aux variables dans les contraintes, qui permettent que la conjonction de toutes les contraintes s'évalue à vrai. Du point de vue de la preuve il s'agit de montrer l'insatisfaisabilité de l'ensemble de clauses :

$\{ \text{Echant} \} \cup \{ \text{BC} \} \cup$

$$\{ \sim C_1 \vee Q_1 , \ldots, \sim C_i \vee Q_i \} \cup \{ \sim Q_1 \vee \ldots \vee \sim Q_{i-1} \vee \sim Q_i \}$$

Si l'ensemble est insatisfaisable, l'insertion est confirmée. Dans le cas contraire, c'est à dire, lorsque il y a au moins un fait dans la BD tel que $\sim C_i$, ces faits sont éliminés pour assurer la satisfaisabilité de la contrainte. Si aucune extension des prédicats impliqués dans la contrainte n'est vide, alors l'insertion est confirmée. Dans le cas contraire, elle est annulée.
Bien entendu, si la contrainte contient un prédicat dont le monde est ouvert et qu'elle fait référence explicitemment aux faits négatifs (sous la forme not_P), alors les faits négatifs (not_P) dans l'échantillon sont pris en compte pour la vérification.

c) si **delete**(C_j) ($1 \leq j \leq i$) alors effectuer l'élimination de la clause $C_j \Rightarrow Q_j$ puis montrer l'insatisfaisabilité de l'ensemble de clauses :

$$\{ \text{Echant} \} \cup \{ \text{BC} \} \cup$$
$$\{ \sim C_1 \vee Q_1 , .., \sim C_{j-1} \vee Q_j, \sim C_{j+1} \vee Q_{j+1} , .., \sim C_i \vee Q_i \} \cup$$
$$\{ \sim Q_1 \vee .. \vee \sim Q_{j-1} \vee \sim Q_{j+1} \vee \ldots \vee \sim Q_i \}$$

Si l'ensemble est insatisfaisable, l'élimination est confirmée. Dans le cas contraire il existe au moins un fait dans la BD tel que $\sim C_{j+1} \vee \ldots \vee \sim C_i$ (une contrainte C_k contredit C_j si $C_k \Rightarrow C_j$ et donc nécessairement $k \geq j+1$). Ces faits sont éliminés pour assurer la satisfaisabilité des contraintes. Si aucune extension des prédicats impliqués dans la contrainte n'est vide alors l'élimination est confirmée. Dans le cas contraire elle est annulée et la contrainte qui a été contredite est déterminée parmi $C_{j+1}, \ldots, C_i$.

Exemple 28

Supposons qu'à un moment donné le concepteur insère d'abord les énoncés de définition d'un "humain" et ensuite les propriétés : "appartenir au sexe masculin" et "appartenir au sexe féminin" en fonction du prédicat "humain". Les clauses générées dans la BC sont les suivantes :

```
compréhension (humain(Ident, Nom, Sex, Age)) ←
        integer (Ident),
        compréhension (nom(Nom)),
        extension (sexe(Sex)),
        compréhension (age(Age)).
```

Le prédicat "sexe" étant défini par extension par :

```
extension (sexe("F")).
extension (sexe("M")).
```

Les prédicats "masculin" et "féminin" définis par compréhension par :

```
compréhension (masculin(X)) ← compréhension (humain(X,_,"M",_)
compréhension (féminin(X))  ← compréhension (humain(X,_,"F",_)).
```

Supposons que par la suite on définisse la relation "enfant" entre deux humains par :

```
éval_MF (enfant).
mode (enfant, ext).
relation (enfant(Fils, Parent)) ←
        compréhension (humain(Fils,_,_,_)),
        compréhension (humain(Parent,_,_,_)).
```

dont l'existence est restreinte par les contraintes suivantes :

1) Tout enfant E a deux parents P1 et P2, l'un de sexe "M" et l'autre de sexe "F".
2) La relation enfant est non-réflexive.
3) La relation enfant est non-symétrique.

Ces contraintes sont écrites sous forme de clause comme suit :

1) humain(P1,_,'F',); humain(P2,_,'F',_) ←
   enfant(E,P1), enfant(E,P2), P1≠P2
   humain(P1,_,'M',); humain(P2,_,'M',_) ←
   enfant(E,P1), enfant(E,P2), P1≠P2

Pour la contrainte de non-réflexivité on a :

2) not (enfant(E, E)) ←

Et pour la contrainte de non-symétrie :

3) not (enfant(P1, E)) ← enfant(E, P1)

On définit la relation "père" en fonction des prédicats enfant et masculin par :

éval_MF (père).
mode (père, ded).
relation (père(P,F)) ←
relation (enfant(F,P)),
compréhension (masculin(P)).

Supposons qu'à la fin, le concepteur veuille insérer la relation "grand-père" comme suit :

← relation(insérer, grand-père(GP,F),
[père(GP,P), père(P,F), féminin(GP)], ded).

Laquelle est bien entendu contradictoire car un grand-père ne peut pas appartenir au sexe féminin.

Voyons d'abord ce qui se passe pour l'échantillon lors de la création de ces énoncés. L'insertion du prédicat humain provoque l'insertion dans l'échantillon du fait suivant :

compréhension (humain (1,X1,X2,X3)).

L'insertion des énoncés par compréhension correspondants aux prédicats masculin et féminin ne provoque rien car ce sont des prédicats qui n'auront pas d'extension dans la BD. De même pour l'énoncé de définition de la relation enfant car les faits pour cette relation peuvent être déduits à partir de "humain".

L'insertion de la contrainte 1) sur enfant entraîne la détermination du nombre minimal de faits pour l'ensemble de propriétés "humain" qui permettra de vérifier la contrainte. Ce nombre est égal à 3 car trois variables différentes portent sur le prédicat "humain". L'interface génère donc deux faits puisqu'il en existe déjà un :

compréhension (humain(2,_,_,_)).
compréhension (humain(3,_,_,_)).

l'interface modifie les faits quantfaits pour le prédicat "humain" et crée le fait respectif pour la contrainte. Puisque la relation enfant intervient dans la contrainte, on génère l'ensemble des faits qui correspondent à la relation d'après sa définition. Dans ce cas, le produit cartésien des faits du prédicat humain. De cette manière les faits suivants sont générés[1] :

[1] Par commodité nous omettons par la suite les préfixes "relation" et "compréhension".

```
enfant(1,1).  enfant(1,2).  enfant(1,3).
enfant(2,1).  enfant(2,2).  enfant(2,3).
enfant(3,1).  enfant(3,2).  enfant(3,3).
```

L'insertion des prédicats père et grand_père ne génère aucun fait dans l'échantillon car ils sont des relations.

Voyons maintenant comment les vérifications de ces énoncés sont effectuées à partir de l'échantillon. L'insertion du prédicat masculin est vérifiée par la dérivation de son énoncé de définition, c'est à dire ← compréhension(masculin(X)). La participation de l'échantillon permet d'instancier la variable X à 1 par l'unification du fait humain(X,X1,'M',X3) avec humain(1,X1,X2,X3). Donc, l'insertion du prédicat masculin est confirmée. De la même façon le prédicat féminin et la relation enfant sont vérifiés.

Etant donné qu'il n'y a pas de contraintes définies sur l'entité humain, on procède à la vérification de la contrainte 1) en formulant l'appel : ← contrainte(X). Cet appel déclenche l'exécution des règles suivantes :

```
contrainte ( 1 ) ←
            (not(relation(enfant(E,P1))) ;
             not(relation(enfant(E,P2))) ;
             not(P1=P2)) ;
             (humain(P1,_,'F',_) ; humain(P2,_,'F',_))).
contrainte ( 1 ) ←
            (not(relation(enfant(E,P1))) ;
             not(relation(enfant(E,P2))) ;
             not(P1=P2)) ;
             (humain(P1,_,'M',_) ;humain(P2,_,'M',_))).
```

lesquelles sont équivalentes à la contrainte 1). S'il y avait eu des contraintes déjà définies sur la relation enfant, on les revérifie sur l'échantillon puisqu'il a été modifié par l'insertion de la contrainte. On procède de la même façon pour les contraintes 2) et 3) en déclenchant l'exécution des règles suivantes :

```
contrainte ( 2 ) ←
                not(relation(enfant(E,E))).
contrainte ( 3 ) ←
                not(relation(enfant(E,P1))) ;
                not(relation(enfant(P1,E))).
```

L'exécution des contraintes 1) donne vrai pour les valeurs E=1, P1=1 et P2=2. Il faut remarquer que l'humain 1 prend, au même temps, les rôles de fils et de parent. Ceci est tout à fait logique car jusqu'à présent on n'a pas restreint la relation enfant par une contrainte de non-réflexivité. L'exécution de la contrainte 2) donne faux car les faits enfant(1,1), enfant(2,2) et enfant(3,3) ne satisfont pas la contrainte. Donc ces faits sont éliminés, les contraintes 1) sont à nouveaux validées et la contrainte 3) peut ainsi être vérifiée sur cette nouvelle extension pour le prédicat de relation enfant car la cardinalité(enfant)=6.
L'exécution de la contrainte 3) donne faux car le fait enfant(2,1) ne satisfait pas la contrainte. De même, pour les faits enfant(3,1) et enfant(3,2). Donc, ils sont éliminés, les contraintes 1) et 2) sont réévaluées et, puisque la cardinalité(enfant) = 1, l'insertion est confirmée.

L'insertion de la relation père est vérifiée par l'appel ← compréhension(père(P,F)). Le fait de l'échantillon enfant(1,2) permet d'instancier les variables P et F respectivement à 2 et 1 et le fait de l'échantillon humain(2,X1,X2,X3) est instancié à humain(2,_,'M',_). L'insertion de l'énoncé de définition pour le prédicat père est donc confirmée.

Enfin, l'insertion de l'énoncé de définition de façon implicite pour le prédicat grand-père est vérifiée également en déclenchant l'appel ← grand-père(GP,F). L'insertion ne peut pas être autorisée. Logiquement, un grand-père ne peut pas appartenir à la classe de féminins. La contradiction est présente au moment de la dérivation du fait humain (2,_,'F',_) de l'échantillon, et le fait humain(2,X1,X2,X2) de l'échantillon à déjà été instancié à humain(2,_,'M',_) par la dérivation du fait masculin(GP). Donc, l'insertion de la classe grand_parent est annulée.

1.5. Démonstration du théorème de l'échantillon

La preuve du théorème de l'échantillon ne peut être que constructive car le théorème s'appuie sur la méthode de construction et d'exploitation de l'échantillon.

Si w est une clause de définition en compréhension du prédicat P dont le mode d'obtention de ses faits élémentaires est égal à *ext*, et si w est satisfaisable sous *D(BC)*, il est certain qu'il existe au moins un ensemble de termes $\{t_1,\ldots,t_n\} \subset D(BC)$ tel que $P(t_1,\ldots,t_n) \equiv$ vrai. Par construction de l'échantillon, ce fait identifié par t_1 a son correspondant t_1' dans l'échantillon, et des valeurs $t_2',\ldots,t_n'$ sont générées pour que $P(t_1',\ldots,t_n') \equiv P(t_1,\ldots,t_n) \equiv$ vrai.

Si w est une contrainte d'intégrité et si elle est satisfaisable sous *D(BC)*, il est vrai qu'il existent des faits élémentaires pour les prédicats dans la contrainte, tels que $v(w) \equiv$ vrai. Par construction de l'échantillon le nombre minimal de faits nécessaires à l'évaluation de la contrainte se trouvent dans l'échantillon et ceux qui ne satisfont pas la contrainte sont éliminés de l'échantillon. Donc, ceux qui restent satisfont, bien entendu, la contrainte.

Dans l'autre sens du théorème, si $D_{Echant} \subseteq D(BC)$, alors tout $X \in D_{Echant}$ appartient aussi à *D(BC)*. Donc, si I est une interprétation sous D_{Echant} elle l'est également sous *D(BC)*. En conséquence, si w est satisfaisable sous D_{Echant} elle l'est également sous *D*.

2. L'interface logique de cohérence de données (LCD)

La fonction de l'interface LCD est d'effectuer tous les contrôles sémantiques définis dans la base de connaissances d'une application donnée. A cette fin, l'interface LCD utilise un mécanisme déductif qui exécute plusieurs tâches sous son contrôle pendant le processus de vérification sémantique d'une transaction.

La vérification sémantique des données est donc effectuée au niveau des transactions. Chaque transaction déclenchée sous le SGBD hôte fait appel à l'interface LCD. Le contrôle de transactions, d'accès concurrents, etc reste à la charge du SGBD hôte.

Le contrôle effectué par l'interface LCD est mis en marche par un appel situé au début de toute transaction. Cet appel est généré soit par le pré-compilateur du LMD, dans le cas de couplage du LMD avec des langages de programmation hôte,

soit par l'interpréteur ou le compilateur du LMD. L'interface LCD vérifie la syntaxe et la sémantique des requêtes Ri de la transaction. Comme résultat de cette vérification, si la cohérence est maintenue (c'est à dire, si l'exécution des requêtes n'amène pas la base de données dans un état incohérent), l'interface LCD initialise à "vrai" une variable VC commune. Dans le cas contraire, la variable VC est initialisée à "faux". L'interface LCD peut autoriser l'exécution d'une requête R_i sous réserve que la transaction exécute une série d'opérations après l'exécution de la requête R_i. Cette série d'opérations est communiquée par l'interface à la transaction. Le contrôle sur l'exécution de ces opérations est effectué par l'interface LCD juste avant le point de confirmation de la transaction (voir figure 2).

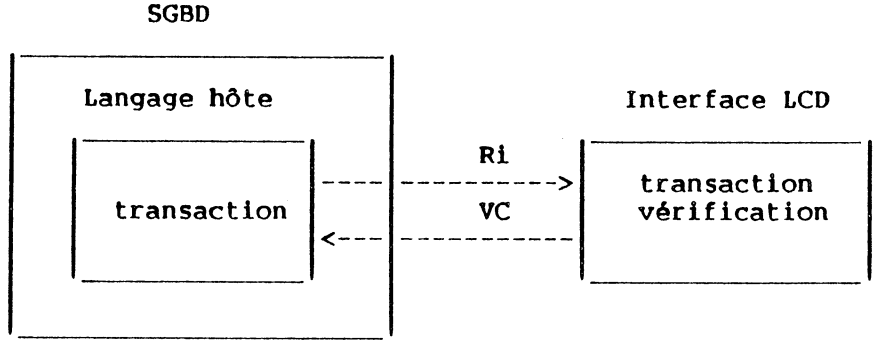


Figure 2 :

Diagramme de communication, sous un SGBD, entre une transaction de données et l'interface logique de cohérence de données. Ri est une requête en LMD et VC est la variable commune aux deux transactions.

2.1. Fonctionnement général de l'interface LCD.

Le traitement effectué par l'interface LCD est le suivant (voir figure 3) :

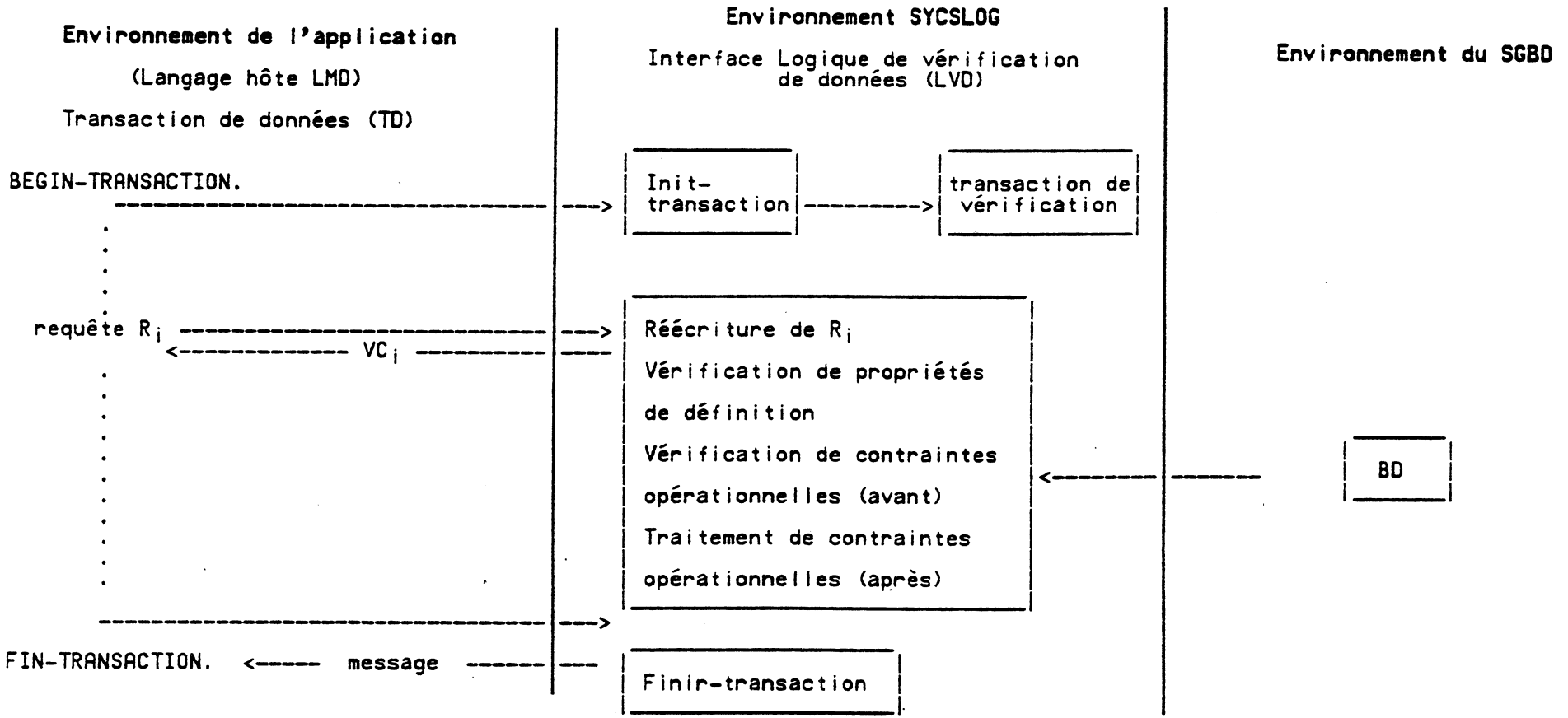


Figure 3 : Schéma de vérification sémantique pour une transaction.

- Pour chaque transaction (TD), l'interface LCD génère une transaction de vérification sémantique (TV) qui s'exécute sous l'environnement de SYCSLOG. L'interface LCD lui assigne un nom qui identifie la transaction.
- L'interface LCD contrôle l'exécution synchronisée des deux transactions (TD et TV).

Chaque transaction de vérification TV effectue à son tour:

1) la vérification de la cohérence de la BD par rapport à la BC actuelle et l'initialisation d'un module dont le nom est le nom associé à la TD par l'interface LCD.

2) Pour chaque requête Ri de mise à jour dans TD, un appel situé juste avant la requête Ri réveille la transaction de vérification TV qui exécute :
   - la réécriture de la requête Ri du LMD sous forme de clause (PR_i),
   - la vérification des propriétés de définition des entités impliquées dans la requête,
   - la recherche de contraintes opérationnelles qui devront être vérifiées avant l'exécution de la requête,
   - la vérification de contraintes opérationnelles (avant) dans l'ordre ascendant du coût,
   - le traitement de contraintes opérationnelles qui doivent être vérifiées après l'exécution de la requête.

3) Si la transaction TD s'est déroluée sans problèmes, la TV est réveillée juste avant la fin de la transaction pour vérifier que toutes les mises à jour qui étaient sous la responsabilité de l'usager ont été exécutées au cours de la transaction. Ceci accompli, les messages correspondants sont envoyés à la transaction TD.
   Si la variable VC a été instanciée à "vrai" la transaction est "confirmée". Dans le cas contraire, la transaction doit être annulée et le mécanisme de récupération du SGBD doit restituer la BD dans l'état où elle se trouvait au début de la transaction.

2.2. Communication avec les SGBDs

Chaque transaction TD communique au SGBD pour exécuter des traitements de données selon leurs droits d'accès respectifs. SYCSLOG utilise cette communication pour effectuer des recherches de données nécessaires au contrôle sémantique à l'intérieur des transactions.

Lors de la phase de l'édition de liens de la transaction TD, les modules suivants doivent être ajoutés :

INIT-TRANSACTION : celui-ci se charge d'initialiser la transaction de vérification pour la transaction TD. Il reçoit en paramètre d'entrée le nom de la base de données qui identifie la base de connaissances ("BC/"DBNAME) et sert à construire l'identificateur pour la transaction TD. Pour les messages à envoyer à la transaction TD un fichier IDENTRANS"/MSGS" est créé.

VERIFREQ : est le module d'accès aux connaissances générales de l'application. Celui-ci vérifie les requêtes en LMD de la transaction TD. Il reçoit en paramètre d'entrée une requête R_i et il envoi une variable VC_i instanciée selon les résultats de la vérification. Une zone de l'espace de travail de la transaction TD est partagée avec ce module pour communiquer les requêtes de mise à jour à propager par la transaction TD.

FINIR-TRANSACTION : ce module autorise ou interdit la confirmation de la transaction TD. Il n'a pas de paramètre d'entrée. La sortie est une variable VC-fin instanciée selon que la cohérence de la BD a été maintenue ou non pendant toute la transaction TD.

Les modules VERIFREQ et FINIR-TRANSACTION sont compilés par le compilateur du langage logique.

Les appels à ces trois modules ainsi que les instructions qui confirment ou annulent la transaction TD selon la variable VC-fin sont générés lors de la compilation de la transaction.

2.3. Vérification des requêtes en LMD

La vérification sémantique d'une requête R_i en LMD dans une transaction TD commence par son analyse syntaxique et sémantique. Cette phase accomplie, la validation syntaxique de la requête R_i autorise sa mise sous forme de clause. Celle-ci est alors vérifiée par les modules logiques de la transaction de vérification associée à la TD.

2.3.1. Analyse syntaxique et réécriture d'une requête en LMD

Cette analyse est effectuée par l'analyseur syntaxique du LMD. Cet analyseur est un ensemble de règles Prolog qui définissent la syntaxe du LMD. Si la requête est valide syntaxiquement, elle est récrite sous forme de clause.

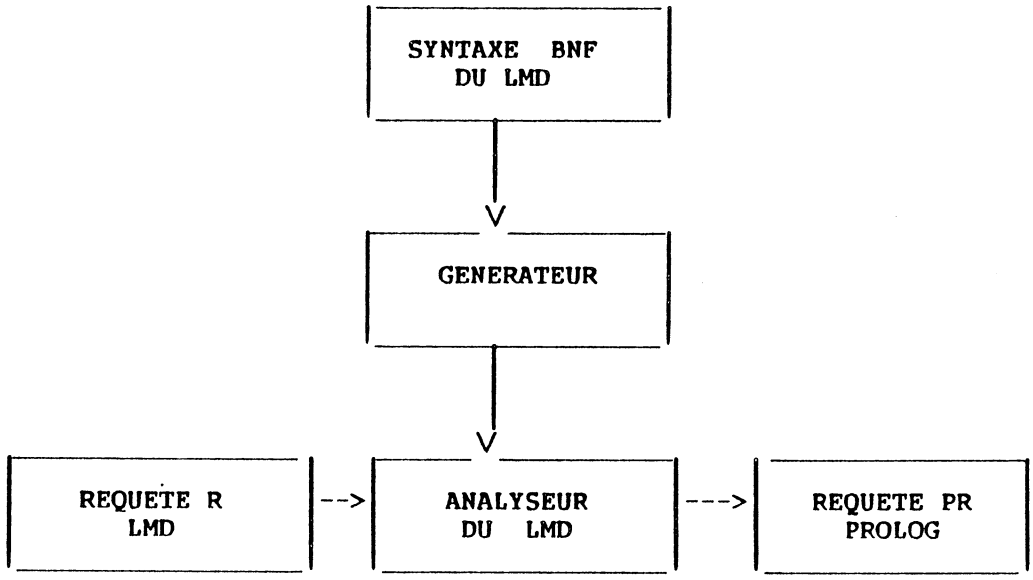


Figure 4 :

Diagramme de récriture d'une requête R en LMD dans une requête PR sous forme de clause.

L'analyseur syntaxique d'un LMD est généré automatiquement par un "Générateur automatique de l'analyseur syntaxique d'un langage". Celui-ci a été construit à partir d'un générateur implanté par P.H Donz comme un exemple d'utilisation de Foll-Prolog [DONZ 83]. Le générateur prend ainsi en entrée l'ensemble des règles qui décrivent la syntaxe du langage LMD. La sortie du générateur est l'analyseur du langage écrit en Prolog. La même procédure est

exécutée pour obtenir l'analyseur du LDD fournit par le SGBD.

2.3.2. Vérification de contraintes définitionnelles

Cette vérification est effectuée en utilisant les connaissances stockées dans la base de connaissances de l'application (BC). Une fois la requête R récrite en Prolog (PR), les contraintes définitionnelles des objets BD impliqués dans la requête sont vérifiées. Leur violation interdit l'exécution de la requête. La vérification porte sur les requêtes de recherche de données et de modification de données. Elle est guidée par une clause dans la BC qui est applicable de façon générale à tout prédicat ayant une extension dans la BD. Elle consiste à :

- vérifier d'abord que chaque objet BD a été défini dans la BC par l'intermédiaire de prédicats de propriété ou de relation.
- ensuite, pour chaque prédicat concerné dans la requête, l'évaluation par le mécanisme déductif de la règle qui le définit est déclenchée. On doit premièrement, vérifier que la requête PR s'unifie avec l'entête de la contrainte définitionnelle et deuxièmement, propager les substitutions obtenues dans les littéraux du corps de la contrainte (chaque littéral doit ainsi se vérifier). La contrainte définitionnelle est satisfaite si l'évaluation de chaque littéral donne "vrai". Suite à ceci la transaction TV procède à la recherche des contraintes opérationnelles qui devront être vérifiées avant l'exécution de la requête. Si l'évaluation d'un littéral au moins donne "faux", la transaction TV communique le résultat à la transaction TD.

Exemple 16

Supposons que pendant le déroulement d'une transaction donnée, on trouve une requête dont la réécriture en Prolog est :

insérer (humain(10, "J.Claude R", "M", 35)).

Premièrement, pour vérifier que "humain" a été défini dans la BC, l'interface LCD déclenche l'exécution de la clause générale suivante :

```
existe_entité (E) ←
        functor (E, Nom-pred, Nf),
        compréhensions (X),
        functor (X, Nom-pred, Nf),
        compréhension (E).

existe_entité (E) ←
        functor (E, Nom-rel, Nf),
        relations (X),
        functor (X, Nom-rel, 2),
        relation (E).
```

en faissant l'appel ← existe_entité(humain(10, "J.Claude R", "M", 35)). L'exécution de cette clause vérifie d'abord que la classe des humains a été définie comme un prédicat de propriété ou de relation. Elle vérifie aussi que la structure du terme humain(10, "J.Claude R", "M", 35) est en accord avec ce qu'a été défini dans la BC. Ensuite, pour vérifier la contrainte définitionnelle du prédicat "humain", la clause compréhensions(E), avec E instanciée à humain(10, "J.Claude R", "M", 35)), on déclenche l'exécution de la règle de définition de "humain"; c'est à dire :

```
compréhension (humain(Id, Nom, Sexe, Age)) ←
        integer (Id),
        compréhension (nom(Nom)),
        extension (sexe(Sexe)),
        compréhension (age(Age)).
```

L'exécution de cette clause vérifie que 10 est une valeur entière, que "J.Claude R" est un nom, et ainsi de suite.

2.3.3. Vérification des contraintes opérationnelles (avant)

Les contraintes opérationnelles à vérifier avant l'exécution d'une requête sont traitées dans l'ordre ascendant du coût. Le coût total d'évaluation de ces contraintes est minimal de même que le coût de chaque contrainte. Chaque contrainte peut entraîner la construction d'une requête de recherche de données de la base dans le LMD hôte. L'exécution des requêtes sur l'état actuel de la BD permet d'obtenir des données pour les évaluations des contraintes opérationnelles (avant).

Exemple 17

Parmi les contraintes opérationnelles à vérifier avant l'insertion d'un fait, on trouve les contraintes d'unicité de valeurs de clé. (section 5.2.3 du chapitre 4). Continuons avec l'exemple précédent d'insertion d'un fait dans l'extension du prédicat "humain". Pour vérifier l'unicité de la valeur de clé 10, l'interface LCD effectue l'appel à la contrainte opérationnelle général d'unicité de clé (7), l'identificateur Idop étant instancié à une valeur qui identifie la contrainte.

← c-opérationnelle (Idop, insérer, humain(10, "J.Clause R", "M",35), avant).

La clause functor(Pred-Termes,Nom-pred,_) dans (7) est unifiée avec functor(humain(10, "J.Clause R", "M", 35), humain,_) et évaluée à "vrai". Lors de la création de l'énoncé de définition pour l'ensemble de propriétés 'humain' le fait suivant a été inséré dans la BC :

key (humain(Ident,_,_,_), humain(Ident,_,_,_)).

La clause key(Pred-Termes,Fclé) est donc unifiée avec ce fait et instanciée à key(humain(10, "J.Claude R", "M", 35), humain(10,_,_,_)). La clause not(exister(Fclé)) est ainsi instanciée à not(exister(humain(10,_,_,_))), laquelle est la seule condition à vérifier qui nécessite de données de la BD. Elle sera donc traduite dans une requête de recherche pour l' humain d'identification égal à 10 dans le LMD hôte.

2.3.3.1. Stratégie de traitement

La stratégie utilisée pour le traitement des contraintes opérationnelles (avant) pour une requête R donnée en LMD est la suivante:

1) déterminer, dans l'ordre ascendant du coût, les contraintes opérationnelles (avant) mises en cause par la requête R.

2) pour chaque contrainte :

   - déclencher l'exécution de ses antécédents récrits dans le LMD hôte dans un ordre qui minimise leur coût,

   - évaluer la contrainte opérationnelle (avant).

La détermination de contraintes opérationnelles mises en cause par une requete R dans le LMD hôte

Dans le but d'assurer une recherche efficace des c-operationnelles (avant) mises en cause par une requête R, le fait suivant est créé lors de la définition des contraintes opérationnelles (ou mis à jour s'il existe déjà) pour chacune des paires Opent-Nompred-Termes :

contrainte-oper (Opent,Nompred-Termes, [Idop1,Idop2,....,Idéopn], avant).

Celui-ci groupe toutes les contraintes opérationnelles (avant) - identifiées par Idéop1,...,Idéopn - portant sur le prédicat Nom-pred (Nompred-Termes = Nom-pred(A1,...,Ap)) pour l'opération Opent sur la BD. Les valeurs Idop sont toujours ordonnées dans l'ordre ascendant des coûts des n contraintes. La détermination des contraintes opérationnelles (avant) mises en cause par une requête R LMD est donc faite d'une façon très simple. Une fois la requête R traduite en Prolog dans un littéral de la forme :

Opent (Nompred-Termes)

l'appel ← contrainte-oper(Opent,Nompred-Termes,Lcoper) est effectué par l'interface LCD et Lcoper est ainsi instancié à la liste de valeurs d'identificateurs des contraintes opérationnelles portant sur le prédicat Nompred pour l'opération Opent.

L'exécution des antécédents d'une c-opérationnelle dans le LMD hôte

Lors de la création d'une contrainte opérationnelle Ci (avant) un fait de la forme :

antécédents-BD (Idop, [A1, A2, ..., Ap]).

est inséré dans la BC. Ce fait groupe, pour une contrainte opérationnelle identifiée par Idop, les antécédents qui impliquent des recherches de données dans la base. Ces antécédents sont ordonnés dans la liste [A1, A2, ..., Ap] de façon à minimiser le coût d'évaluation de la contrainte Ci (section 5.4 du chapitre 4). De même, ce fait a un équivalent qui correspond à sa traduction dans le langage de manipulation de données hôte. Celui-ci est de la forme :

réécrire-antécédents-BD (Idop, RV).

où RV est la requête traduction des antécédents Ai dans le LMD hôte. Lors de la vérification de la contrainte, RV est modifiée avec les valeurs apparaissant dans la

mise à jour. La requête ainsi obtenue est RV'. L'exécution des antécédents d'une contrainte opérationnelle dans le LMD hôte est déclenchée en exécutant sous le SGBD hôte la requête RV' en LMD par un appel de la forme : ← communiquerSGBD(RV'). Les données résultantes de cette requête sont stockées dans la transaction de vérification pour l'évaluation proprement dite de la contrainte opérationnelle par le mécanisme déductif.

L'évaluation d'une contrainte opérationnelle (avant)

Une fois les données recherchées et stockées dans la transaction de vérification, l'exécution de la contrainte est déclenchée par l'interface LCD en faisant l'appel :

← c-opérationnelle (Idop, Opent, Nompred-Termes, avant).

Si l'évaluation donne vrai, la prochaine contrainte opérationnelle dans l'ordre est traitée. Par contre, si elle rend faux, le processus de vérification est arreté et le contrôle est retourné à la transaction avec les messages correspondants.

En résumé, le processus de vérification d'une requête R LMD est exécuté sous SYCSLOG par la clause suivante :

```
vérifier (R, VC)  ←
        analyser_réécrire (R, Opent, Nompred-Termes),
        contrainte_oper (Opent, Nompred-Termes, Lcoper),
        vérifier_contrainte (Opent, Nompred-Termes, Lcoper).

vérifier_contrainte (_, _, [])  ← .
vérifier_contrainte (Op, Npt, [Idop|RIdop])  ←
        réécrire_antécédents_BD (Idop, RVi),
        instancier_antécédents_BD (Idop, Npt, RVi, RVi'),
        communiquerSGBD (RVi'),
        c_opérationnelle (Idop, Op, Npt, avant),
        vérifier_contrainte (Op, Npt, RIdop).
```

2.3.4. Vérification de contraintes opérationnelles (après)

La vérification de ces contraintes est différente de celle des contraintes opérationnelles (avant). Les antécédents des contraintes opérationnelles (après) constituent soit des requêtes de recherche de données pour la vérification de la contrainte, soit de requêtes (insertion, modification et élimination) qui devront être exécutées par l'usager avant la fin de la transaction. Le processus de vérification est très simple, il suffit de contrôler que tous les antécédents ont leur équivalent dans la transaction de données.

Soit :

- R_i une requête en LMD et PR_i sa traduction en Prolog.
- A_j un antécédent d'une contrainte opérationnelle (après) mise en cause par la requête PR_i.

Si $PR_q = A_j$ $(q>i)$ cela signifie que l'usager a formulé la requête R_i nécessaire pour assurer la cohérence d'une mise à jour déjà effectuée. Autrement dit, une contrainte opérationnelle (après) possédant k antécédents (conditions de recherche, d'insertion, d'élimination et de modification de données) est validée pour une requête PR_i, si et seulement si, $\forall j$, $1 \leq j \leq k$

Si A_j = exister(E) $\Rightarrow$ exister(E) $\equiv$ vrai
Si A_j = insérer(E) $\Rightarrow$ $\exists R_q$ dans Ti $(q>i)$/ $PR_q = A_j$ = insérer(E)
Si A_j = éliminer(E) $\Rightarrow$ $\exists R_q$ dans Ti $(q>i)$/ $PR_q = A_j$ = éliminer(E)
Si A_j = modifier(E1,E2) $\Rightarrow$ $\exists R_q$ dans Ti $(q>i)$/ PR_q = modifier(F1,F2)
= modifier(E1,E2)

CHAPITRE 6

CONCLUSIONS

CHAPITRE 6

CONCLUSIONS

Suivant l'approche théorie de la preuve proposée par Reiter [REIT 84] pour formaliser une base de données, nous avons construit une théorie du 1er ordre dont les axiomes sont classés de façon précise et dont les conséquences immédiates se traduisent en une organisation productive et cohérente des connaissances d'une réalité ou d'un environnement. La catégorisation de la théorie est basée sur un modèle logique, implicite lors de la construction d'une théorie particulière. La théorie ainsi construite permet la formalisation d'environnements de nature différente : complète ou incomplète, statique ou dynamique.

En se basant sur cette approche, le système logique d'intégrité sémantique SYCSLOG permet, d'un point de vue plus pratique, l'implémentation de théories construites de cette manière. De ce fait, une base de connaissances, ou schéma logique de représentation d'un environnement, peut contenir sous SYCSLOG :

- des énoncés de définition des données servant à la vérification de la validité syntaxique et de la cohérence sémantique des données explicites
- des lois générales ou contraintes d'intégrité utilisées pour la vérification de l'intégrité sémantique des données explicites
- des lois générales servant à la déduction des données implicites
- des informations sur la nature de l'environnement, et en conséquence sur la manière d'évaluer ses données
- des informations sur la dynamique de l'environnement par la spécification de ses propres opérations.

A partir de ces connaissances, l'interface logique de cohérence des données (LCD) effectue un traitement de l'intégrité sémantique de l'application :

exhaustif, car il est capable de vérifier toute opération de modification de données définie dans la base de connaissances.

général, car la vérification et le maintien sont toujours effectués par l'intermé-

diaire des contraintes opérationnelles définies d'une façon déclarative, c'est à dire non-procédurale, et en conséquence à un niveau assez élevé.

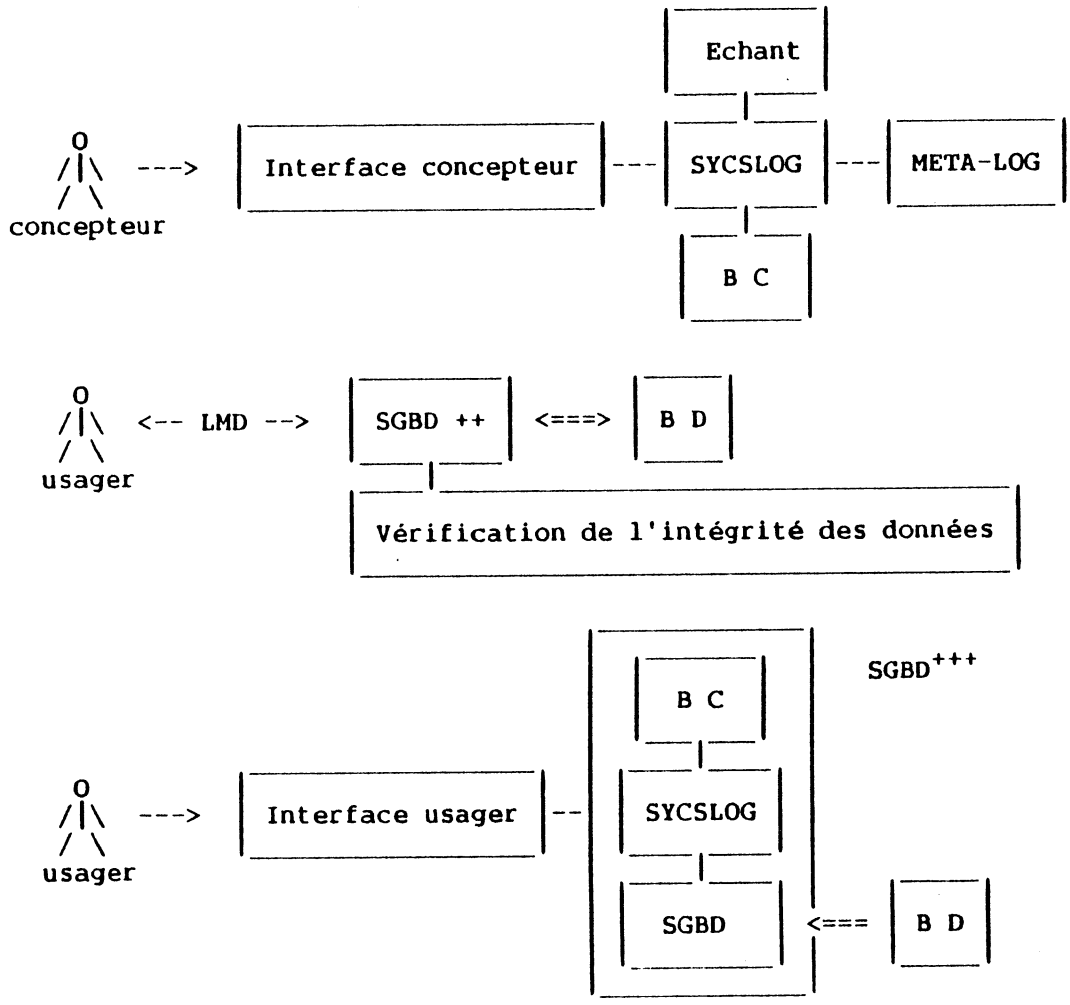
indépendant du SGBD, car les mécanismes logiques de vérification de l'intégrité sous SYCSLOG peuvent être implantés sous n'importe quel SGBD.

L'indépendance entre SYCSLOG et le SGBD est atteinte grâce au modèle logique qui permet l'établissement d'une correspondance entre ses propres objets et les objets conceptuels du modèle soutenu par le SGBD. De ce fait, des mécanismes automatiques de réécriture des langages de définition et de manipulation des données sous forme de clauses (et inversement) sont tout à fait concevables. Ces mécanismes permettent à SYCSLOG de connaître et d'accéder à toute donnée stockée sous le SGBD. En outre, il assure une assistance aux transactions par la communication des requêtes de mises à jour de données à propager à l'intérieur des transactions. Nous avons montré que ces requêtes sont définies par l'intermédiaire des contraintes opérationnelles, récrites dans le LMD hôte, puis envoyées à la transaction pour leur exécution.

L'utilisation de SYCSLOG peut se faire à plusieurs niveaux. Après que le concepteur ait défini un ensemble de connaissances cohérent pour l'environnement, l'usager peut intervenir sur deux plans :

- par l'intermédiaire du SGBD, après qu'il ait été étendu avec les procédures de vérification de l'intégrité des données ($SGBD^{++}$) obtenues à partir de la compilation des connaissances. Dans ce cas, l'accès aux données s'effectue dans le langage de manipulation de données fournit par le SGBD, de la même façon qu'en absence de SYCSLOG.

- par l'intermédiaire d'une interface utilisateur. Dans ce cas, l'usager interagit au niveau de SYCSLOG lui même, ce qui lui permet d'effectuer des recherches de données pouvant impliquer des processus déductifs. Cette facilité s'avère surtout intéressante lorsque le SGBD ne possède pas de capacités déductives qui permettent le traitement de vues récursives, ou la réponse aux questions sous les hypothèses d'un monde fermé ou d'un monde ouvert.



Une extension à cette dernière approche consiste à étendre les fonctionnalités offertes par l'interface utilisateur de façon à pouvoir modifier la base de données. De cette manière tout accès aux données est filtré par SYCSLOG qui constitue ainsi le niveau logique externe d'un SGBD offrant les fonctionnalités suivantes :

- le traitement uniforme de l'intégrité sémantique de l'information
- le traitement des informations sous les mondes fermé ou ouvert
- le traitement des données implicites (définies de façon récursive ou non).

Un prototype de SYCSLOG est en cours de réalisation sur VAX 780 sous le système d'exploitation UNIX Berckeley 4.2. L'interface concepteur ainsi que la base META-LOG sont écrites dans le langage C-Prolog [PERE 84]. Pour la deuxième facette, correspondante aux SGBD^{++}, les modules de communication entre une transaction donnée et SYCSLOG ont été mis en oeuvre par des fonctions

écrites en le langage C (au total trois fonctions). Ces fonctions sont donc facilement appelés depuis n'importe quel langage s'exécutant sous UNIX. Vu les nécessités de performance pour les mécanismes de vérification et de maintien de l'intégrité, l'exploitation de la BC est prévue dans une approche compilée. Les connaissances nécessaires à la vérification sémantique d'une transaction seront donc compilées lors de la compilation de la transaction (par un compilateur Prolog). Par rapport à un interpréteur, ceci offre l'avantage d'un gain substantiel en temps de résolution et en espace mémoire.

Extensions

Les extensions que nous envisageons à l'approche nommée $SGBD^{+++}$ peuvent être situées dans le but d'un enrichissement, au niveau logique, du modèle logique. Ceci se traduit directement dans des bases de connaissances plus riches pour les applications.

Pendant le développement de ce travail, nous avons limité l'établissement de contraintes d'intégrité sur des prédicats ayant une extension dans la base de données. Les propriétés ou relations définies par des axiomes de déduction peuvent être restreintes, de façon indirecte, en restreignant les prédicats de base qui participent dans leur définition. Ainsi, par exemple, si le prédicat de relation *grand-père* est défini en fonction de la relation *enfant* qui a une extension dans la BD, imposer une contrainte de non-réflexivité sur la relation *grand-père* n'est possible qu'en imposant la dite contrainte sur la relation *enfant*.

Il est clair que l'établissement indirect des contraintes n'est pas toujours évident ; parfois il peut être encombrant voire impossible. Par un autre côté, il faut considérer que les faits déduits par un axiome de déduction pour un prédicat donné, satisfont également dans la réalité des propriétés générales ou spécifiques qu'il faut vérifier lors de la déduction des dits faits.

Considérons par exemple la règle de déduction suivante pour un prédicat Q :

$$Q(x) \leftarrow P(x)$$

Cette règle est générale et elle permet de déduire Q(x) pour tout x tel que P(x) ≡ vrai. Une exception à cette règle pourrait, tout simplement, être par exemple :

$$not(Q(a)) \leftarrow P(b)$$

Ceci veut dire que le fait Q(a) ne peut en aucun cas être déduit si P(b) l'est. Il est

très naturel que l'on puisse exprimer cette restriction, dans notre base de connaissances, en tant que contrainte d'intégrité. Le système SYCSLOG déterminerait intelligemment quand et comment la vérifier.

Une extension possible consiste donc à permettre l'inclusion de contraintes d'intégrité portant sur des prédicats définis par des axiomes de déduction. Leur traitement pourrait être similaire à celui donné aux contraintes d'intégrité sur les prédicats ayant une extension dans la BD. Bien entendu, dans ce cas, les opérations *insérer*, *modifier* et *éliminer* n'ont aucun sens ; c'est l'opération *déduction* qu'il faut contrôler. Ainsi, les contraintes opérationnelles générées à partir de la restriction de l'exemple ci-dessus seraient :

c-opérationnelle (*deduction*, Q(a), après) ←
not(P(b)).

c-opérationnelle (*deduction*, P(b), après) ←
not(Q(a)).

La requête de déduction ←Q(a) entraînerait ainsi l'exécution de la règle Q(x) ← P(x), laquelle s'évaluera à vrai si le fait P(a) est vrai. La requête ←Q(a) n'aura une réponse affirmative que si le fait déduit (Q(a)) satisfait les contraintes opérationnelles qui lui ont été imposées, autrement dit, lorsque le fait P(b) ne peut pas être déduit.

Une autre extension possible consiste à augmenter progressivement la base META-LOG avec des prédicats de base et avec les opérateurs respectifs pour leurs termes, tel que des termes temporels, graphiques, etc. Ces prédicats seraient inclus dans les bases de connaissances sur demande du concepteur et en fonction des besoins de l'application.

BIBLIOGRAPHIE

[ADIB 82] M. Adiba, C. Delobel. *"Bases de Données et Systèmes Relationnels"*. Ed. Dunod, Paris 1982.

[ADIB 84] M. Adiba, G.T Nguyen. *"Information procesing for CAD/VLSI on a generalized data management system"*. Proc. 10th Very Large Data Bases Conference". Singapour, Aout 1984.

[BERN 80] P.A. Bernstein, B. Blaustein, E. Clarke. *"Fast Maintenance of Semantic Integrity Assertions using Redundant Aggregate Data"*. Proc. of Very Large Data Bases Conference, Montreal 1980,pp 126-136.

[BERN 81] P.A. Bernstein, B. Blaustein. *"A Simplification Algorithm for Integrity Assertions and Concrete Views"*. IEEE COMPSAC 1981.

[BERN 82] P. Bernstein, B. Blaustein. "Fast Methods for Testing Quantified Relational Calculus Assertions". Proc. ACM-SIGMOD. Conference of Orlando, Florida. June 1982.

[BROD 80] M.L. Brodie. *"The Application of Data Types to Database Semantic Integrity"*. Information Systems, Vol.5, pp 287-296.

[CASA 82] M.A. Casanova, J.M Castilho, A.L Furtado. *"A Temporal Framework for Database Specifications"*. Proc. of the 8th International Conference on Very Large Data Bases, Mexico City, September 1982, pp 280-291.

[CAFE 82] R. Caferra. *"Abstraction, Partage de Structure et Retour Arrière Non Aveugle dans la Méthode de Reduction Matricielle en Démonstration Automatique de Théorèmes"*. Thèse de 3ème cycle. Université Scientifique et Medicale de Grenoble. France.

[CHAM 75] D.D. Chamberlin, K.P. Eswaran. *"Functional Specifications of a Subsystem for Data Base Integrity"*. Proc. International Conference on Very Large Data Bases. Framington Mass. September 1975, pp 48-68.

[CHAN 78] C. L. Chang. *"DEDUCE 2 : Further Investigations of Deduction in Relational Data Bases"*. Logic and Data Bases, Plenum Press, 1978. pp 201-236.

[CHEN 74] F.Chenique. *"Comprendre la Logique Moderne"*. Série Loqique et Informatique 2. Ed. Dunod, Paris-Bruxelles-Montréal.

[CODD 70] E. F. Codd. *"A Relational Model for Data for Large Shared Data Banks"*. Communications of the ACM, Vol. 13, N. 6, pp 377-387

[CODD 79] E.F. Codd. *"Extending the Database Relational Model to Capture More Meaning"*. ACM TODS 4, No.4, December 1979.

[DATE 77] C.J. Date. *"An Introduction to Database Systems"*. Ed. Addison Wesley, 1977.

[DATE 81] C.J. Date. *"Referential Integrity"*. IBM General Products Division. California, USA 95150, January 1981.

[DEMO 83] R. Demolombe. *"Syntactical Characterization of a Subset of Domain Independent Formulas"*. ONERA-CERT, Touluse, France 1983.

[DONZ 83] Ph. Donz. *"Foll, Une extension au langage Prolog"*. CRISS-Université de Grenoble II, 1983.

[FAGI 82] R. Fagin. *"Horn Clauses and Database Dependencies"* Journal of the Association for Computing Machinery. Vol. 29, N.4, October 1982, pp 952-985.

[GALL 79] H. Gallaire, J. Minker, J.M. Nicolas. *"Background for Advances in Data Base Theory"*. Proc. Workshop on formal bases for Data Base. Toulouse 1979.

[GALL 81] H. Gallaire. *"Impacts of logic on Data Bases"*. VLDB 7th Cannes Sept. 1981.

[GALL 82] H. Gallaire, J. Minker, J.M. Nicolas. *"Logic and Data Bases: An overview and survey"*. Joint report CERT/CGE/University of Maryland. October 1982.

[GRAN 82] J. Grant, B. Jacobs. *"On the Family of Generalized Dependency Constraints"*. Journal of the Association for Computing Machinery. Vol. 29, N.4, October 1982, pp 986-997.

[HAMM 75] M. Hammer, D.J. Mcleod. *"Semantic Integrity in a Relational Data Base System"*. Proc. Very Large Data Bases Conference. Framington Mass. 1975. pp 25-47.

[HENS 83] L. Henschen, W. Mc Cune, S. Naqvi. *"Compiling Constraint-Checking Programs from First Order Formulas"*. Advances in Data Bases. Vol 2, 1984. Ed. H. Gallaire, J. Minker. Plenum Press.

[JOUV 79] M. Jouve, C. Parent. *"Qu'est ce qu'une base de données cohérente ?"*. Journées AFCET-IF. "Bases de Données Cohérentes". Paris (Mai 1979).

[KELL 78] C. Kellog, P. Klar and L. Travis. *"Deductive Planning and Pathfinding for Relational Data Bases"*. Logic and Data Bases, Plenum Press, 1978. pp 179-200.

[KOWA 78] R. Kowalski. *"Logic for Data Description"*. Logic and Data Bases. Plenum Press, 1978. pp 77-103.

[KOWA 79] R. Kowalski. *"Logic for Problem Solving"*. Artificial Intelligence Series. Nils J.Nilsson Editor.

[LEVE 84] H. J. Levesque. *"The Logic of Incomplete Knowledge Bases"*. On Conceptual Modeling. Springer-Verlag, 1984. (pp 165-189).

[LOPE 83] M. Lopez, J. Palazzo, F. Velez. *"The TIGRE Data Model"*. Rapport de recherche N.2. Nov. 1983. IMAG. BP 68. St. Martin d'Hères. France

[MINK 78] J. Minker. *"An Experimental Relational Data Base System Based On Logic"*. Logic and Data Bases, Plenum Press, 1978. pp 107-147.

[MINKB 82] J. Minker. *"On Indefinite Databases and the Closed World Assumption"*. Proc of 6th Conference on Automated Theorem Proving. Lecture Notes in Computer Science. No.138, New York 1982.

[MINKB 83] J. Minker. *"On Recursive Axioms in Deductive Databases"*. Information Systems, Vol 8, No 1,pp 1-13, 1983.

[NGUY 83] G.T. Nguyen, J. Olivares, P. Winninger. *"Programmation en Logique pour une Base de Données Généralisées"*. Rapport de recherche N.7, Nov. 1983. IMAG. BP 68. St.Martin D'Hères. France.

[NGUY 85]$_1$ G.T. Nguyen, J. Olivares. *"Semantic Data Organization on a Generalized Data Management System"*. Proc. International Conference on Foundations of Data Organization. Kyoto, Japon, Mai 1985.

[NGUY 85]$_2$ G.T. Nguyen, J. Olivares. *"L'Intégrité Sémantique dans une Bases de Données Généralisées"*. Journées Bases de Données Avancées. Mars 1985. S^t Pierre de Chartreuse, France.

[NGUY 85]$_3$ G.T. Nguyen, J. Olivares. *"Sycslog : Système Logique d'Intégrité Sémantique"*. Rapport de recherche N.26, Nov. 1985. IMAG. BP 68. St. Martin D'Hères. France.

[NICO 78]$_1$ J.M. Nicolas, K. Yazdanian. *"Integrity Checking in Deductive Data Bases"*. Logic and Data Bases. Plenum Press, New York 1978. pp 325-344.

[NICO 78]$_2$ J.M. Nicolas. *"First Order Logic Formalization for Functional, Multivalued and Mutual Dependancies"*. Proc. of the International Conference on Management of Data. ACM SIGMOD Mai-Juin 1978.

[NICO 78]$_3$ J.M. Nicolas et H. Gallaire. *"Data Base : Theory vs. Interpretation"*. Logic and Data Bases. Plenum Press, 1978. pp 33-54.

[NICO 79] J.M. Nicolas. *"Logique et Cohérence dans les Bases de Données"*. Journées AFCET-IP. "Bases de Données Cohérentes". Paris (mai 1979).

[NICO 82]$_1$ J.M. Nicolas. *"Logic for Improving Integrity Checking in Relational Data Bases"*. Acta Informatica 18, 1982, pp. 227-253.

[NICO 82]$_2$ J.M. Nicolas, K. Yazdanian. *"An Outline of BDGEN : A Deductive DBMS"*. ONERA-CERT. Dept. d'Informatique, 31055 Toulouse Cedex, France.

[NILS 71] N.J. Nilsson. *"Problem Solving methods in Artificial Intelligence"*. Mc. Graw Hill, 1971.

[PERE 84] F. Pereira. *"C-Prolog User's Manual"*. Version 1.5. Edinburgh Computer Aided Architectural Design. University of Edinburgh, February 1984.

[PORT 65] J. Porte. *"Recherches sur la Théorie Générale des Systèmes Formels et sur les Systèmes Connectifs"*. Collection de Logique Mathématique. Serie A.

[REIT 78]$_1$ R. Reiter. *"On Closed World Data Bases"*. Logic and Data Bases. Plenum Press, 1978. pp 55-76.

[REIT 78]$_2$ R. Reiter. *Deductive Question-Answering on Relational Data Bases".* Logic and Data Bases, Plenum Press, 1978. pp 149-177.

[REIT 79] R. Reiter. *"On the integrity of typed first order data bases".* Proc. Workshop on formal bases for Data Bases. Advances in Data Base Theory. Vol 1, Toulouse, Dec. 1979.

[REIT 84] R. Reiter. *"A Logical Relational Database Theory".* On Conceptual Modeling. Springer Verlag, 1984. pp 191-233.

[SANT 80] C.S. dos Santos, E.J. Neuhold, A.L Furtado. *"A Data Type Approach to the Entity Relationships Model".* Entity Relationships Approach to Systems Analysis and Design". P.P. Chen (ed). North Holland Publishing Company, 1980. pp 103-119.

[SIMO 84] E. SIMON, P. Valduriez. *"Design and Implementation of an Extendible Integrity Subsystem".* ACM-SIGMOD 74. Boston Mass. June 74.

[STON 75] M. Stonebraker. *"Implementation of Integrity Constraints and Views by Query Modification".* Proc. of the 1975 Sigmod Conference. San Jose, California, May 1975, pp 65-68.

[STON 84] M. Stonebraker. *"Adding Semantic Knodledge to a Relational Database System".* On Conceptual Moddelling. Springer-Verlag, 1984, pp 333-353.

[VELE 84] F. Velez. *"Définition formelle du langage LAMBDA".* Rapport de recherche N.11. Mars 1984. IMAG. BP 68. St Martin D'Hères. France.

[VILL 85] A. Villanueva. *"L'Interface d'accès aux Bases de Connaissances de SYCSLOG".* Rapport de DEA. IMAG BP 68. St Martin D'Hères, Septembre 1985.

[WEBE 83] W. Weber, W. Stucky, J. Karszt. *"Integrity Checking in Data Base Systems".* Information Systems. Vol. 7, N.2, pp 125-136.

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. J. KOULOUMDJIAN, Professeur
. J.M NICOLAS, Directeur de recherche

Madame OLIVARES Judith

est autorisée à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique".

Fait à Grenoble, le 30 mai 1986

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

RESUME

Ce travail présente une approche générale du traitement de *l'Intégrité Sémantique* des données gérées par les Systèmes de Gestion de Bases de Données (SGBD). Une organisation des connaissances (lois générales, données, etc) d'une réalité/environnement est également présentée, accompagnée d'une méthode permettant leur définition cohérente. Les idées proposées ont été formalisées au moyen du Calcul des Prédicats du 1er ordre puis mises en oeuvre par l'intermédiaire du langage Prolog.

Mots clés :

Logique et Bases de Données, Intégrité Sémantique, Contraintes d'Intégrité, Systèmes de Gestion de Bases de Données (SGBD) et Déduction, Modèle Logique, Information Incomplète.