MASTER EN SCIENCES INFORMATIQUES

Contribution à une méthodologie de développement d'applications d'aide à la décision

Hick, Jean-Marc; Fortemps, José

Award date:
1991

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'Informatique

Année académique 1990 -1991

Contribution à une méthodologie de développement d'applications d'aide à la décision

Mémoire de fin d'études présenté par

José FORTEMPS
Jean-Marc HICK

Pour l'obtention du grade de licencié et maître en informatique

Promoteur : J.- L. HAINAUT

Avant-propos

"Concevoir du neuf !" : voilà un objectif bien ambitieux. Pourtant, c'est la motivation qui nous a poussés à choisir ce sujet de recherche.

A la remise de ce manuscrit, avons-nous atteint cet objectif ?
Nous proposons dans ce mémoire de nombreuses idées neuves, mais nous devons bien admettre que certains concepts ne sont pas tout à fait stabilisés. Le sujet est tellement riche qu'il n'a pu être porté à maturité en une année. C'est pourquoi nous faisons appel à votre compréhension si certaines définitions restent imprécises.

Tout d'abord, nous tenons à remercier nos parents pour leur soutien affectueux tout au long de nos études.

Nous sommes reconnaissants au professeur Jean-Luc Hainaut pour sa constante disponibilité et ses conseils toujours judicieux qui nous ont permis d'éviter de nombreuses voies de recherche sans issue.

Nous remercions également les professeurs Jacques Kouloumdjan et André Flory, ainsi que toute l'équipe du laboratoire de recherche en système d'information de l'INSA de Lyon pour leur accueil chaleureux.

Enfin, nous pensons à tous ceux qui nous ont soutenus dans l'élaboration de ce travail.

Résumé

Le but de ce mémoire est de contribuer à l'élaboration d'une méthodologie de développement d'applications d'aide à la décision. Cette méthodologie doit être accessible à l'utilisateur final et pas seulement au développeur professionnel. Elle s'articule autour d'un modèle de spécification de bases de connaissances proposé par la professeur Jean-Luc Hainaut. La base de connaissances est décomposée en base de données, spécifiée par son schéma conceptuel entité/association, et par une base de règles qui décrit des grandeurs dérivables à partir de données externes, issues notamment de la base de données. On donne une définition détaillée des concepts du modèle de spécification, ainsi qu'un ensemble de règles de validation. Enfin, les deux derniers pôles de la méthodologie sont abordés de manière plus succincte : la démarche de conception et la mise en oeuvre.

Abstract

The purpose of this thesis is to contribute to the elaboration of a design methodology of decision support systems development. This methodology is intended for end-users as well as for professional developers. It is based on a specification model for decision support knowledge bases proposed by professor Jean-Luc Hainaut. The knowledge base is split into a data base and a rule base. The data base is described through its entity/relationship conceptual model, while the rule base consists of a set of variables and a set of rules that tell how to derive the values of the variables from external data (e.g. from the data base). The specification model concepts are analysed accurately and consistency rules are put forward. Finally, the two other parts of this methodology are examinated more briefly : design methodology and implementation of a supporting tool.

Table des matières

PARTIE I

INTRODUCTION

Partie I : Introduction

Fin des années 70, suite à l'échec relatif du concept de M.I.S. (Management Information System) et à l'apparition de nouveaux outils (tableurs, systèmes experts,...) sur micro-ordinateurs, une nouvelle façon de concevoir l'informatique dans les organisations est apparue : le "**end-user computing**". De manière étendue, le end-user computing englobe toute utilisation de l'ordinateur par tout membre d'une organisation extérieur au département informatique en tant que support aux tâches qu'il doit accomplir [MULQUIN,89].

Si on se place dans une perspective de gestion, ces tâches sont le plus souvent la prise de décision dans le but de déterminer quel ordre d'action sera suivi. Dans ce contexte, en se référant à la typologie de Keen [KEEN,78], nous aurons affaire à des processus de décision semi-structurés ou non structurés. Pour prendre ce type de décisions, le décideur doit pouvoir interagir avec les ressources informatiques pour y puiser des connaissances à jour sur son environnement. En outre, il doit être capable d'analyser, évaluer et raisonner de manière appropriée avec ces connaissances. Un **système d'aide à la décision** (SAD) est un logiciel qui aide le décideur[1] à traiter ces questions.

Cette introduction aux systèmes d'aide à la décision est relativement vague, mais le champ des SADs n'est pas clairement établi. Keen [KEEN,87] dit à ce propos : "Right from the start of the DSS movement, and, even now, there has been no established definition of DSS".

En 1978, Keen et Scott-Morton [KEEN,78] propose la définition suivante pour les SADs : "The application of available and suitable computer-based technology to help improve in semi-structured tasks". Cette définition reste cependant trop imprécise pour limiter la discipline de l'aide à la décision[2] .

A la manière de Sprague [SPRAGUE,87] ou Holsapple [HOLSAPPLE,87a], nous préférons souligner le mouvement de l'aide à la décision qui a poussé à l'**intégration** de divers outils informatiques (gestion de base de données, gestion de modèles,...). Cette tendance a permis une définition relativement stable de la structure d'un système d'aide à la décision.

---

[1] Pour éviter toute confusion, "**décideur**" désigne dans ce texte les cadres de gestion et non le "top-management". Ce dernier sera intéressé par les systèmes EIS (executive information system) plutôt que par les sytèmes d'aide à la décision.

[2] Le lecteur intéressé par ces questions de délimitation du champ de l'aide à la décision pourra consulter [KEEN,87], [SPRAGUE,87], [ER,88].

I.1. Structure d'un système d'aide à la décision

Nous pouvons représenter l'architecture classique d'un système d'aide à la décision de la manière suivante [HOLSAPPLE,87] :

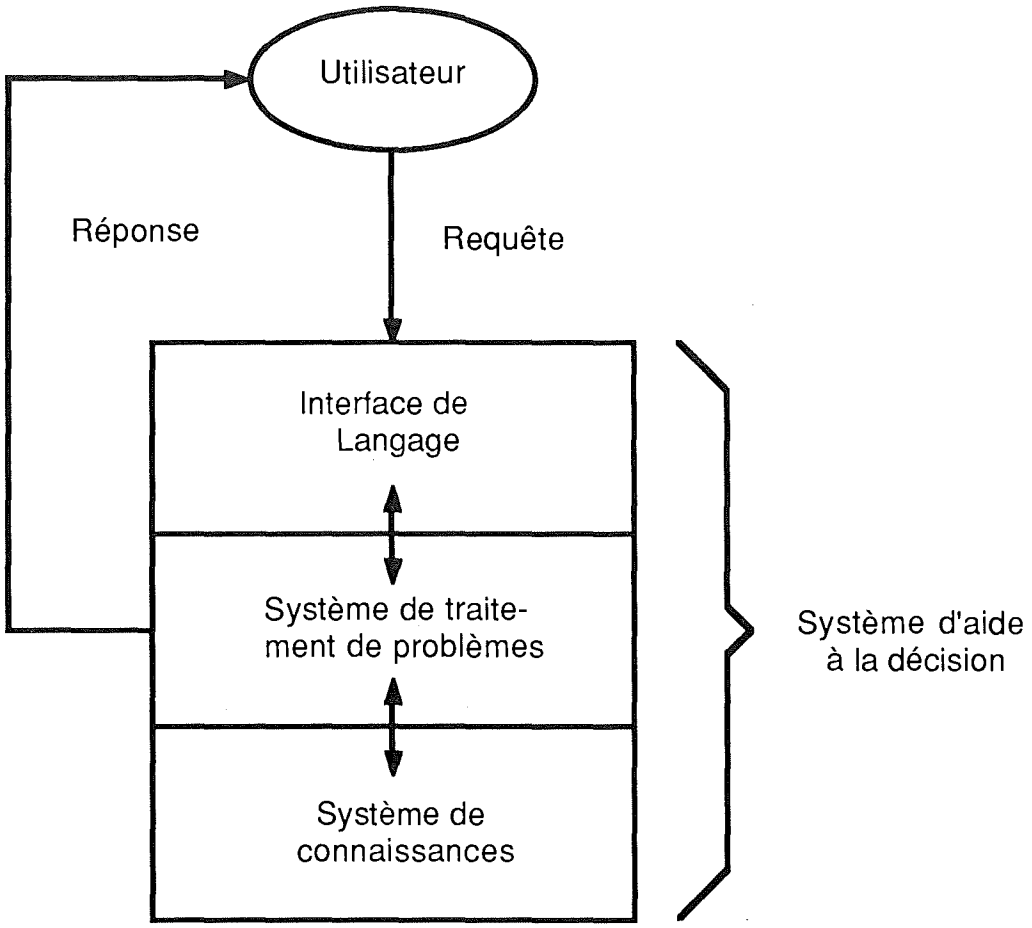


Figure 1.1. : Architecture d'un SAD

Un SAD comporte trois grands sous-systèmes : une interface de langage, un système de connaissances et un système de traitement de problèmes.

L'***interface de langage*** doit permettre au décideur d'exprimer des requêtes et de répondre à toutes les demandes que peut lui adresser le SAD.

Le ***système de connaissances*** consiste en connaissances sur l'environnement du décideur. Il peut contenir des connaissances sous formes très diverses : schémas de données, fichiers texte, modèles, bases de règles... Nous reviendrons sur ces diverses représentations des connaissances dans la suite de cette introduction.

Le ***système de traitement de problèmes*** repose au coeur du SAD. Il peut comprendre la requête d'un utilisateur via l'interface de langage et consulter le contenu du système de connaissances afin de satisfaire cette demande de recherche ou d'analyse.

A la recherche de généralité

Dans les années 70, on avait tendance à construire les SAD à partir de rien avec des langages de programmation classiques. Dans ce cas, leurs connaissances, généralement sous forme de modèles, étaient incorporées dans leur système de traitement de problèmes plutôt que dans leur système de connaissances.

Ce n'est que ces dernières années qu'apparaît le concept de **système de traitement de problèmes généralisé** (STPG). "Un STPG est un système de traitement de problèmes invariant et exploitable pour une large classe d'applications en aide à la décision." [HOLSAPPLE,87]

Le logiciel d'un STPG ne change jamais quelle que soit l'application d'aide à la décision où on l'utilise. Cela est possible parce qu'aucune connaissance spécifique n'est incorporée au STPG ; en effet, toutes les connaissances spécifiques à l'application sont rangées dans la base de connaissances.

Le problème principal pour la création d'un tel logiciel réside dans la capacité à prendre en main de nombreuses méthodes différentes de représentation des connaissances. Pour chaque méthode de représentation des connaissances utilisable dans un système de connaissances, un STPG doit posséder une capacité correspondante de traitement.

Pour créer un STPG, il faut dès lors combiner un grand nombre de capacités de traitement dans un seul programme invariant. Ces capacités devraient être combinées de telle façon qu'on puisse utiliser chacune individuellement. Permettre cette indépendance ne consiste pas à mélanger des organes séparés de traitement de connaissances. Une telle architecture ne permettrait pas à de multiples fonctions de travailler ensemble simultanément ; elle exigerait plutôt d'aller et venir parmi les différents organes.

Les logiciels intégrés

Début 83, apparaît le premier logiciel commercial fondé sur l'idée d'un STPG : Knowledgeman. En 1986, est mis en vente son prolongement, Guru. Avant de caractériser le logiciel Guru, il est intéressant de préciser la notion d'intégration logicielle qui va permettre la mise en oeuvre du concept de STPG.

Tout d'abord, il faut souligner qu'intégration ne signifie pas compatibilité. Si deux programmes peuvent avoir accès au même fichier, alors ils sont compatibles ; pourtant, leurs fonctionnalités respectives ne sont pas coordonnées.

On distingue trois types d'intégration :

1. L'environnement d'exploitation

Un tel logiciel consiste à regrouper un ensemble de programmes indépendants compatibles et d'y ajouter un logiciel pour coordonner leurs activités. Exemple : Windows, environnement Macintosch

Dans ce contexte, le transfert de données entre deux programmes s'effectue par la création de fichiers intermédiaires de transfert ou encore par ce qu'on appelle un procédé "couper/coller".

2. Le mode d'intégration imbriqué ou inclusif

Dans ce type d'intégration, on insère un ou plusieurs composants secondaires à l'intérieur d'un composant principal.

Exemple : - Lotus 123 avec comme composant principal le tableur et, comme composants secondaires, un gestionnaire de fichiers rudimentaire et un générateur de graphiques.
- One-up (COMSHARE) avec comme composant principal le tableur et la possibilité d'accéder à tous les types de bases de données.

Dans une telle configuration, tout le traitement doit être accompli par l'intermédiaire du composant principal. De plus, les capacités des composants secondaires sont restreintes par le composant principal.

3. Le mode d'intégration synergétique

Il s'agit de l'intégration de multiples composants de telle sorte qu'aucun n'en restreint aucun autre et que l'effet total du système soit beaucoup plus grand que la somme des effets individuels de ses composants.

L'impact principal de l'intégration synergétique est qu'elle a rendu les systèmes de traitement de problèmes généralisés concevables.

Exemple : Guru est un outil intégré de développement de base de connaissances. C'est donc un environnement de développement qui intègre, selon un couplage étroit, un tableur, un traitement de texte, un SGBD relationnel, un gestionnaire d'écrans, un gestionnaire de communication, un présentateur graphique, un interpréteur de langage procédural et un moteur d'inférence.

La représentation des connaissances

Dans l'architecture proposée, nous n'avons qu'esquissé le système de connaissances. Celui-ci contient l'ensemble des connaissances sur l'environnement du décideur. Ces connaissances se présentent sous des formes très diverses.

La ***base de données*** reste un élément essentiel d'un système de connaissances. Elle est une source d'information brute ou agrégée pour tous les traitements demandés par l'utilisateur. La capacité de traitement correspondante à la base de données est le système de gestion de bases données (SGBD).

La deuxième méthode d'expression qui émerge est celle de ***modèle***. Nous pouvons définir un modèle comme une "image formalisée des caractéristiques quantitatives de fonctionnement du réel perçu et des relations, le plus souvent quantitatives, elles aussi, qui les structurent" [HAINAUT,89]. Un modèle peut se présenter sous forme de connaissances procédurales (une séquence d'une ou plusieurs étapes pour spécifier comment générer de nouvelles connaissances à partir de celles qui existent). Cependant, dans de nombreux cas, un modèle se présente comme un ensemble de variables logiques et numériques (grandeurs caractéristiques du réel) et d'équations les définissant les unes par rapport aux autres. Des exemples de modèles courants sont la régression linéaire, les algorithmes d'optimisation, les procédures de contrôle d'inventaires, le calcul de taux de rentabilité,etc. L'outil de prédilection pour manipuler les modèles lorsqu'ils se présentent sous la forme d'un ensemble d'équations est le tableur.

Une troisième méthode qui prend de plus en plus d'importance dans les SADs est celle de la ***base de règles*** , c'est-à-dire une collection de règles représentant l'expertise dans un certain domaine. Cette méthode de représentation des connaissances fait typiquement appel aux techniques d'intelligence artificielle. L'outil d'utilisation d'une base de règles est le moteur d'inférence.

Il existe encore bien d'autres méthodes de représentation des connaissances telles que les fichiers texte, les graphiques, etc, mais celles-ci sont moins "opérationnelles" que les trois premières.

Un système de connaissances est donc **une base de connaissances hétérogènes**. Ces dernières sont constituées de règles, statiques ou dynamiques, et de faits. Les règles comprennent notamment le schéma de la base de données, des ensembles de règles de production, des ensembles de règles numériques et logiques.

Pour en terminer avec la description d'un système d'aide à la décision, nous pouvons intégrer dans un schéma général (figure 1.2.) l'ensemble des concepts proposés. Pour des futurs prolongements de cette architecture, on consultera [HOLSAPPLE,87a].

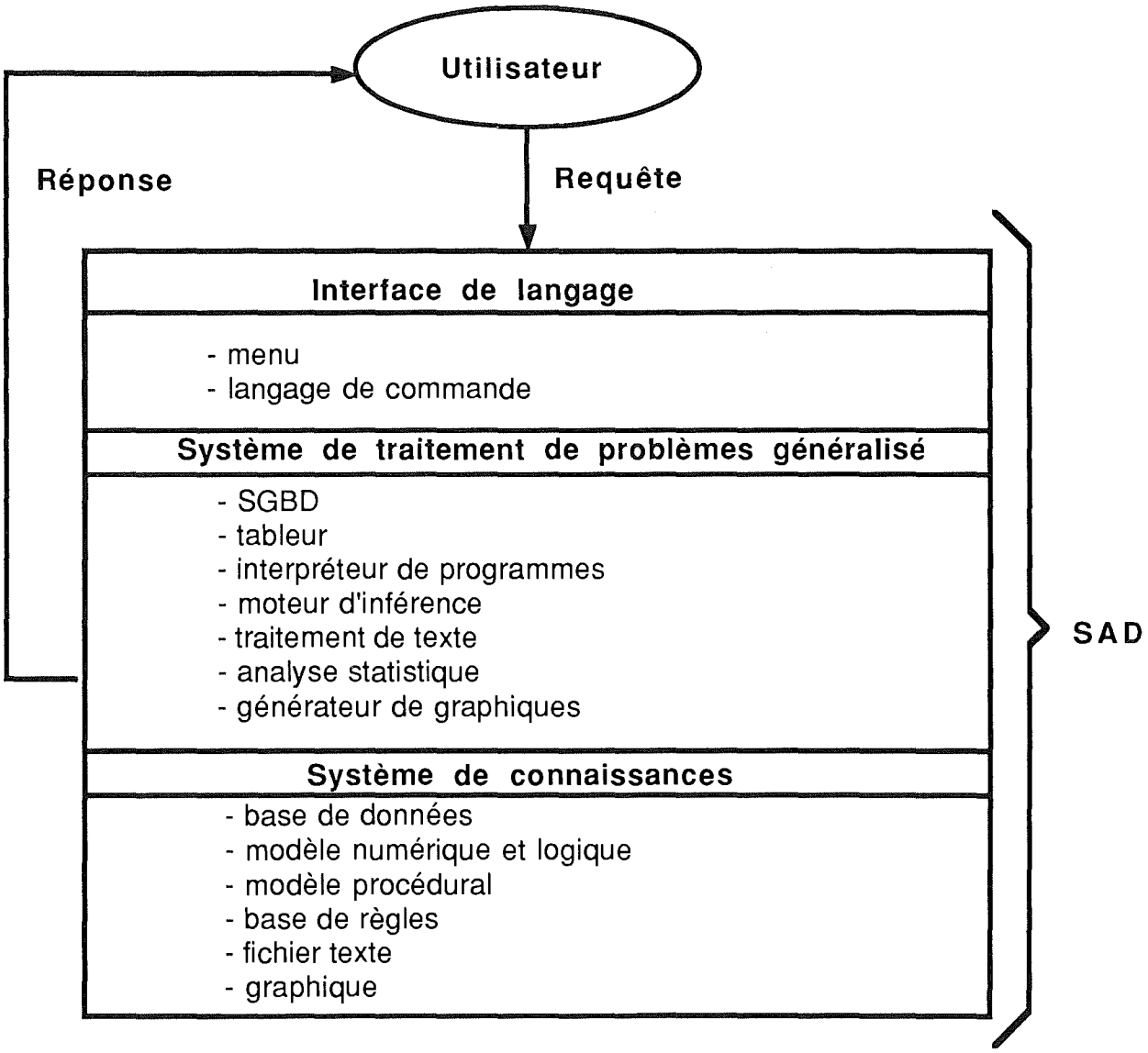


Figure 1.2. : Architecture d'un SAD intégré

I.2. Utilisation d'un système d'aide à la décision

Dans une perspective d"end-user computing", c'est le décideur en personne qui va réaliser des applications d'aide à la décision grâce au SAD. Ces applications se détachent totalement de l'informatique opérationnelle.

Ces applications, supportant très souvent des processus de décision rapides, sont non planifiables et exigent une réalisation accélérée. De plus, on observe fréquemment que l'utilisation d'une application d'aide à la décision induit une modification et une extension de celle-ci. En effet, son déroulement est plus guidé par l'expérience et l'intuition de l'utilisateur que par une série d'étapes figées.

Donc l'utilisateur intervient dans toutes les phases de développement de son application : les étapes de conception, de réalisation et d'utilisation/modification.

Au départ, les applications développées l'étaient principalement sur tableurs. Les raisons du succès de ces derniers [RONEN,89] sont bien connues : manipulation directe, programmation visuelle, langage non procédural, similitude entre modèle de l'outil et mode d'expression de problèmes de l'utilisateur. Mais ce premier type d'outils a été couplé avec d'autres logiciels (SGBD, moteur d'inférence, générateur graphique,...). Ce couplage s'est ensuite transformé en synergie pour donner naissance à la structure que nous avons détaillée au point précédent. L'utilisateur-développeur se trouve dès lors dans une situation proche de celle du programmeur du début des années 60, peu à peu conscient que la maîtrise de Fortran ou Cobol ne suffit plus à la construction systématique de programmes fiables et maintenables.

En résumé, l'utilisateur final est confronté à des logiciels de plus en plus puissants, de plus en plus difficiles à maîtriser et surtout aborde des problèmes de plus en plus complexes. A la différence du professionnel en systèmes d'informations ou en génie logiciel, il ne possède ni modèles, ni méthodes, ni, a fortiori, d'outils d'aide au développement.

Si l'état de l'art en matière d'outils de développement d'application d'aide à la décision évolue très rapidement, on ne peut en dire autant des **aspects méthodologiques**.

Peu d'auteurs se sont attelés à cette tâche ; cependant, elle devient primordiale. En effet, Ronen [RONEN,89] pose le problème de la fiabilité, de la qualité et de la maintenabilité de ces applications pour y relever une situation préoccupante.

De manière analogue aux techniques de génie logiciel, il serait urgent d'aider l'utilisateur dans l'ensemble des étapes de construction de son application et lui fournir des modèles de spécification de connaissances, des outils de réalisation d'applications, des aides pour la conception de tests, etc.

Nous posons le doigt sur des perspectives de recherche importantes et pourtant encore très peu exploitées. Si les travaux sont nombreux dans le domaine de l'aide à la décision, peu ont été adaptés aux problèmes méthodologiques des développeurs. Citons [HOLSAPPLE,87] qui développe une méthodologie de développement de petits systèmes experts et [RONEN,89] qui donne des voies de recherche en ce qui concerne les tableurs.

Dans le cadre de ce mémoire, nous voudrions apporter une contribution méthodologique au problème de développement, par l'utilisateur final, d'applications d'aide à la décision, exécutables sur des logiciels intégrés (intégration soit inclusive soit synergétique). Nous nous intéresserons exclusivement aux **modèles numériques et logiques.** En effet, ce sont les

modèles les plus utilisés par les décideurs, la conception d'une base de règles de production restant encore un problème trop délicat pour les non-initiés. Nous allons étudier les phases de conception, spécification et réalisation d'une application d'aide à la décision dans le champ défini.

Dans une première partie, nous aborderons les problèmes de spécification et de validation d'une base de connaissances. Après une approche globale du modèle de spécification (section II.1.), nous définirons le plus précisément possible les concepts du modèle (section II.2.), et nous proposerons un certain nombre de règles de cohérence pour sa validation (section II.3.).

Dans la deuxième partie, nous proposerons une démarche de conception de modèles.

Enfin, dans une troisième et dernière partie, nous évoquerons brièvement les possibilités de mise en oeuvre du modèle en solution exécutable.

Ces trois parties définissent une **méthodologie de développement** d'applications d'aide à la décision. En effet, "toute méthode doit proposer une **démarche** fondée sur des **modèles** et mise en oeuvre à l'aide d'**outils logiciels**" [BODART,89].

Précisons toutefois que notre ambition n'est pas de définir une méthodologie complète et rigoureuse, mais plutôt, surtout pour les deux dernières parties, de donner des propositions partielles et des voies de recherche.

Notre méthode se base sur un modèle. La définition de celui-ci constitue une étape charnière dans l'énoncé de la méthodologie. Nous terminerons donc cette introduction en précisant à quel type de modèle de spécification de bases de connaissances nous allons nous rattacher.

Le Modèle de spécification de base de connaissances

Le modèle se base très largement sur des concepts et une notation d'expression proposée par Hainaut [HAINAUT,89], [HAINAUT,90].

"D'un point de vue macroscopique, la solution à un problème s'exprime sous la forme d'une base de connaissances qui contient les structures pertinentes d'un domaine d'application. Eu égard, non seulement à l'état de l'art en matière de modélisation, mais aussi à l'architecture des outils actuels qui induit encore chez l'utilisateur la dichotomie données/traitements, nous décomposerons la base de connaissances en **base de données** et **base de règles**" [HAINAUT,90].

Produire une description conceptuelle d'une base de données [BODART,89] et l'exprimer selon la structure d'un SGBD [HAINAUT,86] sont des problèmes désormais bien maîtrisés.

Nous nous tournerons donc vers le problème de la spécification d'une base de règles, appelée modèle, et de son couplage avec le schéma d'une base de données.

Le sujet aurait pu être abordé sous l'angle d'une extension du schéma de la base de données. Dans ce cas, il n'y a pas séparation base de données/modèle, mais plutôt une "intégration" du modèle aux données. Lazimy ([LAZIMY,87], [LAZIMY,89]) propose une spécification de bases de connaissances orientée-objet sous forme d'un schéma Entité/Association étendu (ER^2). On trouvera une autre approche de ce type dans [CHEN,88].

Cette approche a été écartée en raison de l'objectif de clarté et de simplicité que nous nous sommes assigné. Notre méthodologie doit être utilisée par des non-informaticiens et notre travail, pour cette raison, a des **objectifs pédagogiques** importants.

PARTIE II

LE MODELE DE SPECIFICATION DE BASES DE CONNAISSANCES

Partie II : Le modèle de spécification de bases de connaissances

Dans l'introduction, nous avons présenté le type de bases de connaissances auquel nous allons nous intéresser. La spécification d'une base de connaissances doit être fondée sur un modèle. Ce dernier fait l'objet de cette deuxième partie.

Présenter les divers concepts du modèle de spécification les uns à la suite des autres est une tâche impossible car ceux-ci sont interdépendants. C'est pourquoi, nous avons estimé préférable de proposer, dans un premier temps, une **vue d'ensemble des concepts principaux du modèle**. Ensuite, nous analyserons en détail les divers concepts introduits par la vue d'ensemble. Nous terminerons cette partie en examinant le problème complexe de la validation d'une base de connaissances.

II.1. Vue d'ensemble du modèle de spécification de bases de connaissances

II.1.1. Introduction

L'objectif de ce chapitre est d'introduire brièvement les concepts les plus importants du modèle de spécification de bases de connaissances.

Premièrement, une présentation de certaines notions sera donnée : les modèles, les grandeurs et les règles.

Deuxièmement, nous allons approcher l'intégration modèle/base de données. L'origine des grandeurs d'un modèle peut être plus générale que la simple assignation de valeurs. Le concept de variable entité va supporter cette intégration et permettre l'extraction des valeurs de la base de données. Pour bien montrer les réalisations possibles, un exemple de base de données et de base de règles sera développé.

Cette section reprend des concepts examinés dans [HAINAUT,89] et [HAINAUT,90]. Il s'inspire d'ailleurs très largement de ces deux articles.

A la suite de cette vue d'ensemble, nous serons en mesure de préciser le plan de la section suivante qui a pour but d'analyser en détail les divers concepts du modèle de spécification.

II.1.2. Modèles numériques et logiques

II.1.2.1. Notion de modèle

On appelle ***modèle numérique et logique*** une collection de déclarations définissant un ensemble de données, un ensemble de résultats attendus par l'utilisateur et un ensemble de relations de calcul exprimant les résultats en fonction des données.

Dans ce travail, nous nous limiterons à ce genre de modèle. Connaissant la valeur de chacune des données, il est possible de déterminer les valeurs des autres grandeurs jouant le rôle de résultats. Les modèles dans lesquels on a choisi a priori les données et les résultats sont appelés ***modèles directionnels*** .

Supposons que **R** désigne la liste des grandeurs à calculer et **D** la liste des données du modèle, on peut symboliser le modèle de la manière suivante avec une notation vectorielle :

D	: données
R	: résultats
R=f(D)	: règles

Dans la suite du mémoire, nous n'envisagerons plus que ce type de modèles. Notons toutefois qu'il existe des systèmes pour lesquels la direction d'une relation (ou règle) est indifférente.

Etant donné que nous sommes dans le domaine de la spécification de bases de connaissances pour un utilisateur final ("end-user computing"), ce dernier doit avoir à sa disposition un modèle permettant de définir des grandeurs représentatives du domaine d'application qui ne sont pas des résultats attendus et dont le but premier est de simplifier l'élaboration des règles. Nous les appellerons ***grandeurs internes***. Le modèle symbolique ci-dessus peut être complété de la manière suivante :

D	: données
R	: résultats
I	: grandeurs internes
RI=f(G)	: règles

Il y a donc trois grandes classes de grandeurs dans un modèle : les données (**D**), les résultats (**R**) et les grandeurs internes (**I**).

Prenons un exemple très simple pour illustrer ces notions :

Données :
A : réel ; coefficient de X^2
B : réel ; coefficient de X^1
C : réel ; coefficient de X^0
Résultats :
X_1 : réel ; première racine de l'équation
X_2 : réel ; deuxième racine de l'équation
Grandeur Interne :
ρ : réel ; variable de calcul qui représente une constante dans le calcul des racines
Règles :

$$X_1 = \frac{-B+\sqrt{\rho}}{2A}$$

$$X_2 = \frac{-B-\sqrt{\rho}}{2A}$$

$$\rho = B^2 - 4AC$$

ρ est une grandeur interne facilitant la compréhension et l'élaboration du modèle, mais elle n'est pas indispensable à sa conception.

Figure 2.1. : Calcul des racines d'une équation du second degré : (AX^2+BX+C)

II.1.2.2. Notion de grandeur

Nous avons vu qu'un modèle est constitué d'un ensemble de données, de résultats et de grandeurs internes. Il s'agit de concepts mesurables que l'on va classer en deux types : les grandeurs simples et les grandeurs dimensionnées.

Une ***grandeur simple*** est une grandeur mesurable à laquelle on peut associer une valeur à tout instant. Cette valeur sera le plus souvent numérique (quantité, distance, coefficient, ...), parfois logique (vrai ou faux, grand ou petit, ...), qualitative (nom de personne, libellé de produit, ...) ou temporelle (date, mois, année, ...). Une grandeur prend ses valeurs dans un ensemble nommé ***domaine de valeurs*** .

Exemple : Dans la figure 2.1., **A** est une grandeur simple. Remarquons d'ailleurs que ce modèle ne contient que des grandeurs simples.

Par contre, on associe à une ***grandeur dimensionnée*** non pas une valeur mais une suite de valeurs. On envisage donc la description simultanée d'états ou d'éléments successifs du domaine de valeurs.

Supposons que l'on désire décrire le domaine d'application d'une grandeur dans son évolution temporelle. On prendra en compte la dimension du temps représentée par la grandeur **T**. Une telle grandeur peut être considérée comme une ***dimension*** du domaine. On utilisera la notation indicée pour ce nouveau concept.

Exemple : T : t1...tn ; T désigne les mois t1 à tn.

X_T et Y_T sont des grandeurs dimensionnées par le temps T. On associe donc à X_T et Y_T une suite de **n** valeurs.

$X_T = f(Y_T)$ est mis pour $X_{ti} = f(Y_{ti})$ (i=1... n) et représente donc les n règles similaires : $X_{t1} = f(Y_{t1})$

$X_{t2} = f(Y_{t2})$

...

$X_{tn} = f(Y_{tn})$.

Vu la distinction opérée sur les grandeurs, nous pouvons également parler de modèles simples et modèles dimensionnés. Un modèle simple est constitué de grandeurs simples et de règles. Un modèle dimensionné utilise des grandeurs dimensionnées. Il décrit soit un domaine d'application contenant plusieurs objets similaires obéissant aux mêmes règles de comportement soit une suite d'états du domaine.

II.1.2.3. Notion de règle

Dans un modèle, les résultats et les grandeurs internes sont définies par des relations ou règles. Une règle est une relation directionnelle et elle définit la grandeur indiquée en partie gauche. Elle aura la forme générale suivante :

$$G = f(H)$$

où **G** est la grandeur définie par la règle, **f** une fonction monovaluée dont on a soit le nom (si on ne désire pas la décrire explicitement : on parlera de sous-modèle) soit l'expression d'évaluation et **H** une liste de grandeurs.

Il existe différents types de règles. Voici une brève description de quelques relations particulières. Le point II.2.3. les examinera plus en détails.

L'expression d'évaluation d'une règle pourra faire appel à tout opérateur mathématique jugé utile. Il sera donc possible à l'utilisateur d'employer les fonctions d'agrégation : produit (Π) et somme (Σ).

Exemple : $X = \sum_{T} Y_T$ est mis pour $X = \sum_{t=t1, \ldots, tn} Y_t$

On n'admettra pas dans un modèle qu'une grandeur soit définie par plusieurs règles. Si une grandeur est définie par plusieurs expressions, on utilisera une règle à définitions multiples.

Exemple : X, Y, Z sont des grandeurs du modèle

$X = Y+1$ si $Z < 0$
$= Y$ si $Z = 0$
$= Y-1$ si $Z > 0$

Le modèle adopté permet également de mettre en oeuvre la notion de règle récurrente.

Exemple : $X_{T(1)} = 0$
$X_{T(i)} = X_{T(i-1)} + 2$, $(i = 2 \ldots n)$

II.1.3. Modèles et bases de données

II.1.3.1. Origine des grandeurs d'un modèle

Jusqu'à présent nous avons admis que, lors de l'évaluation d'un modèle, une valeur pour chacune des données devait être fournie par l'utilisateur. Dans certains cas, les données peuvent avoir une portée plus générale que celle du modèle élaboré. De même, des résultats d'un modèle pourraient être utilisés comme données dans un autre. Il existe donc une interdépendance entre des modèles relatifs à un même domaine d'application.

Les grandeurs et les règles d'un modèle ne couvrent que certains aspects du domaine modélisé. En particulier, la structuration du domaine de valeurs selon des classes ainsi que les relations entre celles-ci ne peuvent y être modélisées. C'est pourquoi, nous allons intégrer spécification de modèles et structure de bases de données. Nous utiliserons le modèle Entité/Association car il s'impose à l'heure actuelle comme standard de spécification du schéma conceptuel d'une base de données.

II.1.3.2. Présentation de l'exemple

Dans la suite de cette section, nous développerons un exemple de base de connaissances. L'objectif du modèle est de calculer le montant des commandes d'un client. Il nous semble intéressant de présenter dès à présent le schéma de la base de données pour permettre aux lecteurs de bien visualiser la situation. Dans le point II.1.3.4., le modèle sera développé entièrement afin d'illustrer les concepts décrits.

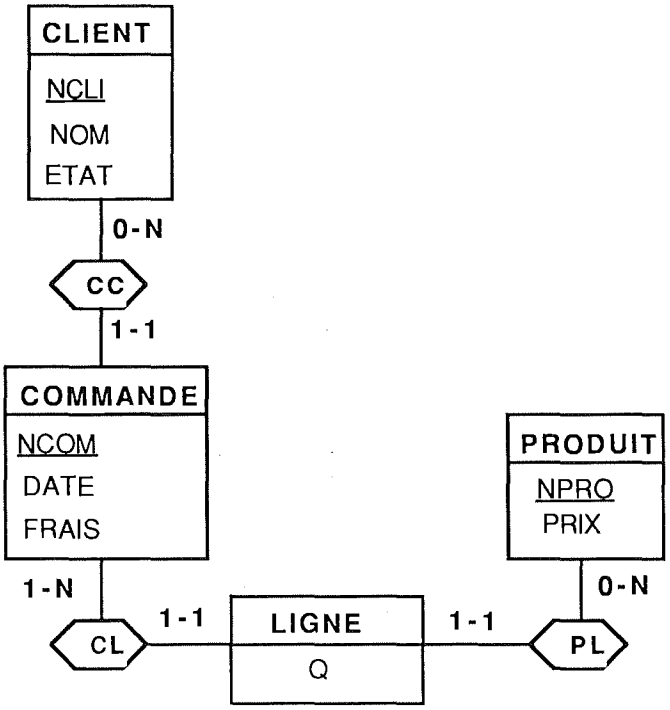


Figure 2.2. : Schéma de la base de données

II.1.3.3. Les variables entité

Une ***variable entité*** est une grandeur du modèle (donnée, résultat ou grandeur interne) jouant le rôle de dimension et définie sur un ensemble d'entités. Elle prendra ses valeurs dans un ensemble d'entités que l'on définira à l'aide d'une expression de requête. Le concept de variable entité va permettre de réaliser l'intégration modèle/base de données.

Une grandeur dimensionnée par une variable entité peut être assimilée aux valeurs d'un attribut attaché à l'ensemble d'entités représenté par la variable entité. Cette grandeur est soit un attribut réel (de la base de données) soit un attribut dérivé.

Avant toutes choses, il faut choisir un langage de requête permettant de désigner de tels ensembles. Pour une question de simplification, on utilisera un langage dédié à une restriction binaire du formalisme Entité/Association dans lequel les types d'associations sont de degré deux, sans attribut, et dans lequel les attributs des types d'entités sont obligatoires, monovalués, non répétitifs et indécomposables. Un schéma de structure plus complexe peut être transformé en un schéma respectant les restrictions ci-dessus et qui lui est sémantiquement équivalent [HAINAUT,86].

La désignation d'un ensemble d'entités couvert par une variable entité se fera dans une règle de domaine.

Toutefois, voici quelques exemples significatifs et usuels :

- CLI = CLIENT(:NCLI = 123)
  La variable entité CLI désigne une entité CLIENT dont l'attribut NCLI est égale à 123.
- COM = COMMANDE(CC:CLI)
  La variable entité COM désigne une entité COMMANDE associée via CC à l'entité CLI (entité dont le numéro de client NCLI est égale à 123).

- LC = LIGNE((CL:COM)et(:Q > 100))
  La variable entité LC désigne une entité LIGNE associée à une commande COM dont la quantité Q est supérieure à 100.
- COM = COMMANDE(CC:CLIENT(:ETAT>0))
  La variable entité COM désigne une entité COMMANDE associée via CC à une entité CLIENT dont l'état du compte est supérieur à 0.

II.1.3.4. Exemple

Après avoir introduit toutes les notions importantes du modèle de spécification, nous pouvons écrire la base de règles exprimant le calcul du montant dû par le client de numéro N :

Donnée :
N : entier ; numéro du client
Résultat :
MONTANT-DU $_{CLI}$: réel ; montant de la facture due par le client CLI
Grandeurs internes :
CLI : CLIENT ; client dont le numéro de client est N
COM : COMMANDE ; commande du client CLI
LC : LIGNE ; ligne de commande de COM
PRO : PRODUIT ; produit concerné par la ligne de commande LC
MONTANT-COM $_{COM}$: réel ; montant de la commande COM
MONTANT-LIGNE $_{LC}$: réel ; montant de la ligne LC
Règles :
Règles de domaine :
CLI = CLIENT(:NCLI = N)
COM = COMMANDE(CC:CLI)
LC = LIGNE(CL:COM)
PRO = PRODUIT(PL:LC)
Règles du problème :

$$\text{MONTANT-DU}_{CLI} = \sum_{COM} \text{MONTANT-COM}_{COM} - \text{ETAT}_{CLI}$$

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LIGNE}_{LC} + \text{FRAIS}_{COM}$$

$$\text{MONTANT-LIGNE}_{LC} = Q_{LC} * \text{PRIX}_{PRO}$$

Figure 2.3. : Modèle de calcul du montant dû par le client dont on donne le numéro.

Illustrons à partir de la figure 2.3. quelques notions vues précédemment :
- N est une grandeur simple.
- CLI, COM, LC, PRO sont des variables entité.
- MONTANT-COM $_{COM}$ est une grandeur dimensionnée par une variable entité. Elle représente un attribut dérivé du type d'entités COMMANDE.
- PRIX $_{PRO}$ est un attribut réel du schéma de la base de données et une grandeur dimensionnée par une variable entité.

- $\sum_{LC}$ MONTANT-LIGNE $_{LC}$ + FRAIS $_{COM}$ est une règle contenant une fonction d'agrégation. Le montant d'une commande est égale à la somme des montants de chaque ligne de la commande plus les frais la concernant.

II.1.4. Plan d'analyse des concepts du modèle de spécification

Maintenant que nous situons un peu mieux les principaux concepts du modèle de spécification, nous sommes en mesure d'établir un plan d'analyse de ces derniers. Les deux concepts qui prévalent sont la notion de grandeur et la notion de règle. Vu le caractère indissociable de ces deux notions, nous avons privilégié le concept de grandeur pour effectuer notre analyse.

Dans un premier temps, nous aborderons donc la notion de grandeur. Nous distinguerons les grandeurs jouant le rôle de dimension (II.2.1.) et les autres types de grandeurs (II.2.2.). Cette division se justifie par le rôle différent joué par ces deux types de grandeurs dans la détermination du résultat d'un modèle. Nous reviendrons plus tard sur cette dichotomie. Pour chacun de ces points, nous détaillerons la syntaxe et la sémantique des règles de définition des grandeurs analysées.

Ensuite, nous étudierons trois types de règles plus particulières : les règles de récurrence (II.2.3.1.), les règles récursives (II.2.3.2.) et les contraintes d'intégrité (II.2.4.). Cette analyse clôturera l'étude des deux notions primordiales que sont les règles et les grandeurs.

Nous détaillerons deux concepts non encore abordés mais tout aussi importants pour la construction de modèles : la notion de sous-modèle (II.2.5.) et le traitement de la valeur nulle (II.2.6.).

Enfin, pour reprendre tous les concepts analysés et mettre en évidence les liens qui les réunissent, nous proposons de les structurer dans un schéma Entité/Association (II.2.7.). Cette étape constituera un bon résumé de tout ce qui a été mis en évidence auparavant.

Pour clôturer l'analyse des concepts, nous nous proposons de prendre du recul et d'étudier les relations qui existent entre modèle et base de données (II.2.8.). Dans ce dernier point, nous constaterons l'analogie remarquable entre une base de données déductives et un système qui intégrerait base de données et gestion de modèles.

Tout au long de cette partie, nous allons définir le plus clairement possible des notions et des principes. Leur compréhension n'est pas toujours des plus aisée, car ce domaine assez neuf évolue et n'a pas encore atteint une stabilité théorique. Dans ce contexte, l'utilisation d'exemples clairs devient une nécessité. C'est la raison pour laquelle nous proposons **trois études de cas dans l'annexe I**. Elles seront continuellement utilisées. Nous conseillons donc aux lecteurs de les lire attentivement avant d'entamer la section suivante.

II.2. Les concepts du modèle de spécification

II.2.1. Les dimensions

Une grandeur dimension permet d'associer à une grandeur non pas une valeur mais une suite de valeurs. On associe à cette grandeur autant de valeurs qu'il y a d'éléments dans le domaine de valeurs de sa dimension. Prenons un exemple pour illustrer le concept de dimension.

Exemple : Soit la grandeur simple BUDGET.

Si l'on veut décrire l'évolution temporelle de la grandeur BUDGET, on la dimensionne avec une grandeur T qui représentera le temps.

Soit T = jan, fév, mar, avr, mai.

Dans ce cas, $BUDGET_T$ est une grandeur dimensionnée par le temps T.

On a ainsi :

$$BUDGET_T = \left| \begin{array}{l} BUDGET_{JAN} \\ BUDGET_{FEV} \\ BUDGET_{MAR} \\ BUDGET_{AVR} \\ BUDGET_{MAI} \end{array} \right|$$

Dans la section II.1., nous avons distingué, de manière implicite, deux types de dimensions : les variables entité et les indices non liés à la base de données. Pour faciliter la désignation de ces derniers, nous emploierons le terme générique de **variables simples**.

Dans un premier point, nous nous intéresserons aux **variables simples** et à leur règle de définition. Nous tenterons de définir clairement leur statut dans un modèle en utilisant la classification données, résultats et grandeurs internes. Nous détaillerons ensuite la syntaxe et la sémantique de leur règle de définition : les règles de domaine.

Dans un second point, nous ferons de même pour les **variables entité.**

Ensuite, nous examinerons **la relation de dépendance qui existe entre les variables entité** d'un modèle. En effet, ces dernières sont généralement définies à partir d'autres variables entité. De plus, ces variables désignent des entités particulières reliées dans le schéma de la base de données. C'est ce lien sémantique dont hérite les variables entité que nous allons décrire.

Enfin, nous terminerons cette analyse par l'adjonction d'un **paramètre externe** à une grandeur dimension. Ce paramètre va nous permettre de désigner une valeur particulière dans la suite de valeurs d'une dimension et d'introduire un ordre sur le domaine de celle-ci. Par exemple, ce concept permettra de désigner la valeur particulière $BUDGET_{FEV}$ de la grandeur dimensionnée $BUDGET_T$, ce qui était impossible avec l'ensemble des notions introduites précédemment.

II.2.1.1. Les variables simples

Au départ, le couplage modèle/base de données n'avait pas été envisagé. Dans ce contexte, les valeurs des données provenaient uniquement de l'assignation de valeurs par l'utilisateur. Toutefois, la notion de dimension avait déjà été introduite afin de **contracter l'expression** de certaines règles et permettre une **forme de généralité**.

Exemple : soit un modèle de calcul du salaire des employés d'une firme.

Supposons que cette firme possède deux employés (Emp1 et Emp2) et désire calculer leur salaire pour les mois de janvier et février. Dans ce cas, le modèle doit calculer quatre grandeurs différentes : SalaireE1Jan, SalaireE2Jan, SalaireE1Fev et SalaireE2Fev. Cependant, les règles de calcul de ces grandeurs sont identiques. C'est pourquoi on introduit la notion de dimension qui permet de "rassembler" ces quatres grandeurs en un seule grandeur dimensionnée.

Pour ce faire, on définit deux dimensions : E = Emp1, Emp2 et M = Jan, Fev.

Le modèle calcule alors une seule grandeur dimensionnée Salaire $_{E,M}$.

Remarquons que si on veut calculer le salaire d'un employé supplémentaire, le modèle n'est pas modifié. Seule la dimension E prendra une valeur supplémentaire, par exemple Emp3.

Après avoir introduit l'origine des variables simples, nous pouvons passer à une description plus formelle du concept et de ses règles de définition.

II.2.1.1.1. Définition

> Une variable simple (Vs) est une grandeur du modèle
> - jouant le rôle de dimension,
> - dont le domaine des valeurs n'est pas défini sur un ensemble d'entités

Le domaine de valeurs d'une variable simple est exprimé dans une règle de domaine. La syntaxe utilisée est beaucoup plus simple que dans le cas de variables entité (en l'absence de lien avec la base de données). Nous détaillerons la syntaxe des règles au point suivant.

Exemple : nous pouvons reprendre l'exemple du BUDGET qui nous a servi à introduire la notion de dimension.

T est la dimension.

Son domaine est défini par la règle de domaine :

T = jan, fév, mar, avr, mai.

Elle dimensionne la variable $BUDGET_T$ qui représente la suite de valeurs ($BUDGET_{JAN}$, $BUDGET_{FEV}$, $BUDGET_{MAR}$, $BUDGET_{AVR}$, $BUDGET_{MAI}$).

Replaçons la notion de variable simple dans les trois grandes classes de grandeurs : les données, les résultats et les grandeurs internes.

Données

Une variable simple est une donnée du modèle si l'utilisateur doit initialiser sa valeur pour connaître les résultats. Dans ce cas, l'utilisateur doit introduire la suite de valeurs de la dimension.

On aura :

Dans la rubrique "Données" :

Vs ':' Type ; définition en français

Dans la rubrique "Règles de domaine" :

Vs '=' Règle de domaine

Exemple : Soit un modèle qui calcule les prévisions de budget pour un certain nombre d'années. L'utilisateur doit introduire les différentes années pour lesquelles le budget devra être estimé.
On aura donc un modèle du type :
Données :
AN : entier ; années sur lesquelles vont porter l'analyse
Résultats :
$BUDGET_{AN}$: entier ; estimation du budget pour l'année AN
Règles de domaine :
AN = a1, a2,... ...,aN

Résultats

Une variable simple peut être un résultat si les valeurs désignées dans sa règle de domaine sont une solution du modèle.

On aura ainsi :

Dans la rubrique "Résultats" :

Vs ':' Type ; définition en français

Dans le rubrique "Règles de domaine" :

Vs '=' Règle de domaine

Exemple : Retrouver les années pour lesquelles le budget a été supérieur à la moyenne des budgets rassemblés.

La variable simple résultat dépendra de la suite de valeurs d'une autre. Dans les règles de domaine, nous devons donc prévoir une syntaxe qui permet de lier deux variables simples.

Notons également que, dans ce cas, la variable simple peut ne pas jouer son rôle de dimension.

Grandeurs internes

Pour faire partie de cette classe, la variable simple n'est ni un résultat attendu par l'utilisateur, ni une grandeur dont la suite de valeurs doit être initialisée.

Si ces valeurs ne doivent pas être introduites, alors la variable simple dépend des valeurs d'une autre, ou encore ses valeurs restent fixées pour toutes exécutions du modèle.

Nous distinguerons donc deux cas :

- la variable simple a sa suite de valeurs fixée pour toute exécution du modèle.

  Exemple : Soit un modèle qui calcule des prévisions de budget pour les années 1992, 1993 et 1994. On aura ainsi :

  Grandeurs internes :

  AN : entier ; années pour lesquelles on calcule le budget

  Règles de domaine :

  AN = 1992, 1993, 1994.

- la variable simple dépend de la suite de valeurs d'une autre, mais elle n'est pas une solution du modèle.

  C'est une situation identique aux variables simples résultats. On utilisera donc la même syntaxe pour lier les variables simples. Cette syntaxe, nous la présentons au point suivant.

II.2.1.1.2. Règles de domaine des variables simples

Avant de détailler la syntaxe, rappelons qu'une dimension prend ses valeurs dans le domaine défini par la règle de domaine. Un domaine est donc un ensemble de **valeurs distinctes**.

Suite à la classification des variables simples dans les trois grandes catégories de grandeurs, nous pouvons distinguer trois grands types de règles de domaine pour les variables simples.

1. La variable simple est une donnée et sa suite de valeurs doit être introduite par l'utilisateur du modèle.

   On aura dans ce cas la syntaxe suivante : $Vs = Vs_1, Vs_2, Vs_3, ..., Vs_N$

   où N représente le cardinal de la suite de valeurs de Vs

   Exemple : AN = a1, a2, ..., aN

2. La variable simple est une grandeur interne et sa suite de valeurs est fixée pour toutes les exécutions du modèle :

   On aura dans ce cas : $Vs = valeur_1, valeur_2, ..., valeur_N$.

   Exemple : AN = 1992, 1993, 1994.

3. La variable simple est une grandeur interne ou un résultat dont la suite de valeurs dépend d'une autre variable simple.

   Le formalisme proposé permet de désigner un sous-ensemble dans la suite de valeurs d'une variable simple. Il ne permet donc pas d'exprimer qu'une variable simple dépend de plusieurs autres. En fait, ce mécanisme n'est pas nécessaire, car quand on veut spécifier qu'une grandeur est

dimensionnée conjointement par plusieurs variables simples, ces dernières sont exprimées explicitement[1] .
Exemple : Soit les variables simples E et T qui désignent respectivement les
employés d'un firme et les mois de l'année.
Nous avons les règles de domaine suivante :
E = E1, E2, ..., EM
T = Jan, Fév, ..., Déc.
Soit SALAIRE une grandeur qui exprime le salaire de l'employé E au mois T. On utilisera la notation $SALAIRE_{E,T}$ pour exprimer cette situation.

Soit **Rd** la règle de domaine de Vs, une variable simple
Rd a la forme générale suivante :

Rd = (Vs1 / Csel)

où Vs1 est une variable simple qui désigne tous les éléments de sa suite de valeurs,
Csel est une **condition de sélection**.

Une condition de sélection est
soit une expression de deux conditions d'appartenance liées par un opérateur logique (et/ou) et entourée de parenthèses.
soit une **condition d'appartenance** unique.

Une condition d'appartenance (Cap) a la forme générale suivante :

Cap = G Rel Val

où G est soit Vs1 et, dans ce cas, désigne une valeur particulière de Vs1
soit une grandeur indicée par Vs1
Rel est un des opérateurs suivant (<,>,=,≥,≤,in,not-in,==,not==)
Val est soit une valeur unique
(avec Rel dans (<,>,=,≥,≤))
soit un ensemble de valeurs
(avec Rel dans (in,not-in,==,not==))

Exemples: Reprenons notre exemple du budget et définissons deux nouvelles variables simples à partir de AN
- ANB = (AN / $BUDGET_{AN}$ > 1.000.000)
Cette règle désigne les années pour lesquelles le BUDGET estimé sera supérieur à un million.
Dans ce cas, Vs1 = AN,
G est une grandeur indicée par AN,
Rel = ">" et
Val = 1.000.000

[1] Ce n'est pas le cas des variables entité : nous détaillerons cette relation de dépendance entre variables entité au point II.2.1.3.

- ANC = (AN / AN > 1989)
  ANC a comme suite de valeurs celle de AN amputée des valeurs inférieures ou égales à 1989. G est la variable simple AN.
  On a : Vs1 = AN,
  G = AN = Vs1 (la variable simple),
  Rel = ">" et
  Val = 1989

II.2.1.2. Les variables entité

II.2.1.2.1. Définition

> Une variable entité (Ve) est une grandeur du modèle
> - jouant le rôle de dimension
> - définie sur un ensemble d'entités d'un type

Un ensemble d'entités est désigné grâce à une expression de requête. Celle-ci permet de déterminer dans quel domaine la variable entité prend sa valeur. Une règle de ce type sera nommée **règle de domaine**. Nous détaillerons la syntaxe du langage de requête au point suivant.

Exemple : Reprenons le schéma des clients et des commandes de l'exemple introductif (figure 2.2.).
Soit CLI une variable entité tel que CLI = CLIENT(:NCLI>100)
La règle de domaine "CLIENT(:NCLI>100)" désigne tous les clients dont le numéro (NCLI) est supérieur à 100.`
Si dans la base de données, on a

CLIENT		
NCLI	NOM	ETAT
70	DUPOND	-1000
120	DURAND	0
150	DUVERT	0

La règle de domaine désigne les entités CLIENT qui ont leur numéro respectivement égal à 120 et à 150.
Si on veut calculer le montant dû par ces deux clients, on dimensionne la grandeur MONTANT-DU par la variable entité CLI.
On obtient de cette manière :

$$\text{MONTANT-DU}_{CLI} = \left| \begin{array}{l} \text{MONTANT-DU}_{\text{CLIENT(:NCLI=120)}} \\ \text{MONTANT-DU}_{\text{CLIENT(:NCLI=150)}} \end{array} \right|$$

En plus de son rôle de dimension, le concept de variable entité va permettre d'**extraire des valeurs de la base de données**. Prenons un exemple pour illustrer ce rôle des variables entité. La dimension CLI de l'exemple précédent va nous permettre d'extraire des valeurs d'attributs de l'entité CLIENT. Si on veut obtenir le nom des clients de numéro respectivement égal à 120 et à 150, il suffit de dimensionner l'attribut NOM de CLIENT par la variable entité CLI. On a ainsi :

$$NOM_{CLI} = \left| \begin{array}{l} NOM_{CLIENT(:NCLI=120)} \\ NOM_{CLIENT(:NCLI=150)} \end{array} \right| = \left| \begin{array}{l} DURAND \\ DUVERT \end{array} \right|$$

Soulignons le caractère très simple de la syntaxe proposée dans la règle de domaine : le langage de requête n'utilise aucun mot-clé superflu (seuls les opérateurs "et" ":" sont utilisés) et la notation NOM_{CLI} désigne de manière univoque et proche de l'intuition le NOM du client CLI.

Intéressons-nous à présent au rôle joué par les variables entité dans les trois grandes classes de grandeurs d'un modèle : les données, les résultats et les grandeurs internes.

Données

Les variables entité ne sont pas considérées comme des données du modèle. En fait, la base de données dans son ensemble fait partie des données. Ainsi, une variable entité, désignant par définition un ensemble d'entités, ne sera jamais incluse aux données.

Si on veut exécuter le modèle pour un ensemble d'entités d'un type, il faut spécifier cet ensemble dans les règles de domaine (même dans le cas particulier où on l'exécute pour toutes les entités d'un type).

Exemple : Dans le modèle introductif de la figure 2.3., aucune variable entité n'apparaît dans la rubrique "Données".
Si on modifie quelque peu l'exemple en calculant le montant dû par chaque client (et non plus uniquement pour celui de numéro N), la rubrique "Données" sera vide.

Résultats

Une variable entité peut être un résultat si l'ensemble d'entités désigné dans sa règle de domaine est solution du modèle.

On aura ainsi :

Dans la rubrique "Résultats" :

Ve ':' Type d'entité ; définition en français

Dans la rubrique "Règles de domaine" :

Ve '=' Règle de domaine

Dans un modèle un tant soit peu intéressant, apparaîtra dans la règle de domaine une grandeur interne faisant l'objet de nombreux calculs.

Exemple : Retrouver les clients qui ont payé plus de 80 % de leurs factures dans un délai de 5 jours.

De manière synthétique, ce type de modèles équivaudra à une condition de sélection complexe[1] .

Notons que, dans ce cas particulier, la variable entité peut ne pas jouer le rôle de dimension.

Grandeurs internes

Il s'agit des variables entité qui sont définies par une règle de domaine, mais qui ne sont pas un résultat attendu par l'utilisateur.

On a ainsi :

Dans la rubrique "Grandeurs internes" :

Ve ':' Type d'entité ; définition en français

Dans la rubrique "Règles de domaine" :

Ve '=' Règle de domaine

C'est dans cette classe que la plupart des variables entité apparaîtront. Dans ce contexte, elles jouent leur rôle de dimension.

Exemple : Dans l'exemple introductif de la figure 2.3., les quatre variables entité définies CLI, COM, LC et PRO sont des grandeurs internes.

II.2.1.2.2. Règles de domaine des variables entité

Comme son nom l'indique, ce type de règles définit le domaine de la variable entité. En d'autres termes, c'est la désignation du sous-ensemble d'entités du type de la dimension.

Procédons à la **définition syntaxique et sémantique** des règles de domaine. La syntaxe proposée s'inspire du langage de désignation de données proposé par Hainaut [HAINAUT,86].

Nous proposons d'abord une étude complète de la grammaire utilisée. Notre présentation de la syntaxe n'est pas des plus aisées à comprendre, c'est pourquoi nous conseillons au lecteur de consulter les exemples, placés à la suite, parallèlement à la théorie. Le cheminement sera facilité par des références aux exemples placés en marge droite dans la théorie.

Enfin, nous clôturerons cette analyse par deux restrictions au langage afin de satisfaire notre objectif pédagogique.

Avant de s'attaquer à la grammaire proprement dite, précisons ce que désigne une variable entité et un type d'entités dans une règle de domaine.

Le nom d'une variable entité désigne une entité particulière.

Exemple : si la variable entité CLI est du type CLIENT, alors l'expression CLI désigne une entité CLIENT particulière.

---

[1] Dans ce contexte, un modèle peut jouer le rôle de langage de requête (point II.2.8.4.).

Le nom du type d'entités désigne **toutes** les entités d'un type.
Exemple : COMMANDE désigne l'ensemble des entités du type COMMANDE.

Ainsi, une variable entité prend ses valeurs dans l'ensemble désigné par la règle de domaine et, quand elle apparaît en partie droite (d'une règle de domaine), elle désigne une entité particulière.

Une règle de domaine (Rd) se présente comme suit :

Rd '=' Ens Csel

où Ens est un type d'entités de la base de données
(on l'appelera le type d'entités **initial**)
Csel est une condition de sélection (éventuellement absente)

Condition de sélection (Csel)

Une condition de sélection est
soit une expression booléenne de deux conditions de sélection liées par un opérateur logique (et/ou) et entourée de parenthèses,
soit une condition d'association.

Condition d'association (Cas)

Une condition d'association (Cas) a la forme générale suivante :

Cas = (A : Card Ens)

où Ens est l'expression de désignation d'un ensemble d'éléments du type N2,
N1 est le nom du type des éléments évalués,
Card est la désignation d'un entier ou d'un intervalle d'entier,
A est le nom, éventuellement absent, d'un type d'associations entre les types N1 et N2.

Remarque : un intervalle s'exprime sous la forme i..j et désigne les entiers de i à j, où i et j sont les désignations de deux entiers. Le symbole * représente "le nombre d'éléments du type N2 associés par A à l'élément du type N1 en cours d'évaluation".
Exemples d'intervalles :
1..4 signifie "de 1 à 4",
0..1 signifie "de 0 à 1", c'est-à-dire "au plus 1",
1 mis pour 1..1, signifie "1 exactement",
1..* signifie "de 1 à tous",
* mis pour *..* signifie "tous",
4..* signifie "de 4 à tous" (au moins 4").
Cas particulier : l'absence de Card dans l'expression de la condition représente l'expression la plus fréquente : 1..*, c'est-à-dire "au moins 1".

Après avoir développé Ens et en supposant que la condition de sélection soit une condition d'association unique, une règle de domaine se présente comme suit:

Rd '=' Ens1 (A : Card Ens2 Cond)

Si Ens2 est un type d'entités de la base de données,
Alors - A est un type d'associations entre Ens1 et Ens2,
- Cond est soit absente (1)
soit une condition de sélection (Csel) (2)
soit une condition d'appartenance(Cap)
Si Ens2 est une variable entité, (3)
Alors - A est un type d'associations entre Ens1 et le type d'entités de Ens2,
- Cond est absente[1] .
Si Ens2 est un attribut de Ens1
Alors - A est absente,
- Cond est une condition d'appartenance

Condition d'appartenance (Cap)

Une condition d'appartenance (Cap) a la forme générale suivante :

Cap = Rel Ens

où Rel est une relation du type >,<,=,≥,≤,in,not-in,==, not==
Ens est l'expression de désignation d'un ensemble d'éléments
(la relation == se définit par l'égalité d'ensembles)

Dans le cas où Cond est une condition d'appartenance, on obtient pour la règle de domaine :

Rd '=' Ens1 (A : Card Ens2 Rel Ens3)

Si Ens2 est un type d'entités de la base de données,
Alors - Rel appartient à (in, not-in, ==, not==)
- Ens3 est un type d'entités sur lequel repose une condition de sélection (4)
Si Ens2 est un attribut de Ens1
Alors Ens3 est soit une valeur unique (avec Rel dans (>,<,=,≥,≤))
soit un ensemble de valeurs
(avec Rel dans (in, not-in, ==, not==))

Dès lors, si Ens3 est une valeur unique
alors Ens3 est soit une constante (5)
soit une grandeur du modèle (6)
si Ens3 est un ensemble de valeurs
alors Ens3 est soit un intervalle fixé (7)
soit un ensemble de valeurs de la base de données désigné par une règle (8)

---

[1] Cette contrainte (Cond absente)pourra être levée avec l'introduction du concept d'ensemble dérivé (point II.2.8.5.)

Remarque : pour désigner un ensemble de valeurs de la base de données, on emploiera toujours une expression de la forme suivante :

Attr (:Ens Csel)

où Ens est un type d'entités de la base de données,
Attr est un attribut de Ens et
Csel est une condition de sélection.

Illustrons la syntaxe des règles de domaine par quelques **exemples**. Nous utiliserons les références précédemment placées en marge droite pour bien préciser à quel cas on se rapporte. Les exemples se basent sur le schéma de base de données des clients de la figure 2.2.

(1) COM = COMMANDE (CL : LIGNE)
La variable entité COM désigne une entité COMMANDE qui comporte au moins une ligne.
Dans cet exemple, on a le schéma suivant :
Ens1 = COMMANDE
Ens2 = LIGNE (type d'entités)
A = CL (type d'associations entre LIGNE et COMMANDE)
Cond est absente

(2) CLI = CLIENT (CC : COMMANDE (CL : 10..* LIGNE))
La règle de domaine désigne l'ensemble des clients qui ont passé au moins une commande qui comporte au moins 10 lignes.
On a : Ens1 = CLIENT
Ens2 = COMMANDE
A = CC (type d'associations entre CLIENT et COMMANDE)
Cond = (CL : 10..* LIGNE) (condition de sélection)

(3) COM=COMMANDE (CC : CLI)
CLI est une variable entité de type CLIENT.
La variable COM désigne une entité COMMANDE associée via CC à l'entité CLI.
On a : Ens1 = COMMANDE
Ens2 = CLI (variable entité)
A = CC (type d'associations entre COMMANDE et le type de CLI (CLIENT)) et Cond est absente

(4) CLI = CLIENT (CC : COMMANDE not-in COMMANDE(CL: 10..* LIGNE)
La règle de domaine désigne l'ensemble des clients qui ont passé au moins une commande qui n'appartient pas à l'ensemble des commandes qui ont au moins 10 lignes.
On a : Ens1 = CLIENT
Ens2 = COMMANDE
Rel = not-in
Ens3 = COMMANDE (CL : 10..* LIGNE)
(type d'entités + condition de sélection)

(5) CLI = CLIENT (: NCLI = "123")
La variable CLI désigne l'entité CLIENT de numéro "123".
On a : Ens1 = CLIENT
Ens2 = NCLI
A absente
Rel = '='
Ens3 = "123" (constante)

(6) CLI = CLIENT (: NCLI = N)
N est une donnée du modèle.
La variable CLI désigne l'entité de CLIENT de numéro N.
Les caractéristiques de la règle sont identiques à l'exemple précédent à l'exception d'Ens3 égal à N (grandeur du modèle).

(7) CLI = CLIENT (: NCLI IN 1..N)
N est une donnée du modèle.
La règle de domaine désigne l'ensemble des clients qui ont un numéro compris entre 1 et N.
On a : Ens1 = CLIENT
Ens2 = NCLI
A est absente
Rel = '='
Ens3 = 1..N (intervalle à bornes fixées)

(8) CLI = CLIENT (: NOM in NOM (:PRODUIT))
La règle de domaine désigne l'ensemble des clients dont le nom est le même que celui d'un produit.
On a : Ens1 = CLIENT
Ens2 = NOM
A est absente
Rel = in
Ens3 = NOM (: PRODUIT) (ensemble de valeurs)
avec Ens = PRODUIT
Attr = NOM
Csel est absente

Si la description formelle n'est pas toujours des plus aisée à cerner, les exemples soulignent une forte similitude entre notre langage d'expression de requêtes et le schéma de la base de données. Cette similitude permet une navigation aisée à travers un grand nombre de relations.

Exemple : P = PRET((concerne:EXEMPLAIRE(ne:NUMERO (constitution:REV))) et (:DATEDEB ≥ A-D))

Dans cette règle de domaine, on relie l'entité PRET à l'entité REV en suivant le "chemin" des associations "concerne", "ne" et "constitution".

Insistons sur l'absence de tout mot-clé superflu (par exemple, référence au sens des accès) qui accentuerait l'écart entre syntaxe et schéma de la base de données.

Le langage de requête proposé possède un pouvoir d'expression étendu. Il permet de spécifier des requêtes très complexes. Cependant, nous ne devons pas oublier notre objectif pédagogique. Dans cette optique, nous devons trouver un **équilibre** entre le pouvoir d'expression du langage et la complexité des constructions qu'il permet.

Lors de la construction de tout modèle, le concepteur devra toujours veiller à définir le domaine de ses variables entité le plus **"simplement"** possible.

De plus, nous allons adjoindre **deux restrictions** au langage proposé afin d'augmenter la lisibilité des requêtes. Nous montrerons également que ces deux restrictions ne sont pas trop limitatives en expliquant comment elles peuvent généralement être remplacées par des constructions plus simples.

Une condition d'appartenance sera uniquement utilisée pour qualifier des valeurs

Les règles de domaine où une condition d'appartenance porte sur un ensemble d'entités sont rares ((4) en marge droite). De plus, la règle est difficilement compréhensible. C'est pourquoi, nous préférons interdire ce genre de construction. Grâce à l'exemple, nous verrons aussi comment éviter ce type de règles.

Exemple : CLI = CLIENT (CC : COMMANDE not-in COMMANDE(CL: 10..*LIGNE)

La règle de domaine désigne l'ensemble des clients qui ont passé au moins une commande qui n'appartient pas à l'ensemble des commandes qui ont au moins 10 lignes.

On peut réécrire cette règle de manière plus simple :

CLI = CLIENT (CC:COMMANDE(CL:0..9 LIGNE))

Une condition d'association ne comportera qu'une seule condition d'appartenance si Rel appartient à (<,>,≥,≤,=)

Si Rel appartient à (<,>,≥,≤,=), l'ensemble désigné dans la condition d'appartenance (Ens3) est une valeur unique. Dans ce cas, elle est associée à une et une seule entité. Cette dernière peut être spécifiée sous la forme d'une nouvelle variable entité. C'est pourquoi on ne tolérera dans une condition d'association qu'une seule condition d'appartenance quand Rel est du type précité.

Exemple : (inspiré par l'étude de cas n°1)

ABO = ABONNEMENT(aa:ANNEE(:AN ≥ DATEDEB(:REVUE(:TITRE= NOMREV))))

ABO désigne l'ensemble des abonnements qui ont été pris après la date de parution de la revue de nom NOMREV.

Nous avons deux Cap avec les relations "=" et "≥". On ne tolère pas ce cas.

Ens3 = DATEDEB(:REVUE(:TITRE=NOMREV))

REVUE est le type d'entités "associé à" Ens3. La condition d'association (:TITRE=NOMREV) désigne une entité unique de REVUE qui peut être spécifiée comme suit :

REVUEBIS=REVUE(:TITRE=NOMREV).

On peut réécrire la règle de domaine grâce à cette nouvelle variable entité comme suit :

ABO = ABONNEMENT(aa:ANNEE(:AN=DATEDEB $_{\text{REVUEBIS}}$))

Si Rel appartient à (in,not-in,==), Ens3 est un ensemble de valeurs. Dans ce cas, on n'imposera aucune condition sur le contenu de Cond (hormis, qu'elle ne peut contenir qu'une seule condition d'appartenance avec une relation du type (<,>,=,≥,≤)).

Exemple : (inspiré étude cas n°2)

PRO=PRODUIT(:NOM in NOM(:REGION(cr:CLIENT)))

Cette règle de domaine désigne tous les produits dont le nom est celui d'une région où il y a au moins un client.

Terminons l'analyse des règles de domaine en précisant que les différents opérateurs décrits (:,et,ou,<,>,=,≥,≤,in,not-in,==) sont les seuls qui sont tolérés dans l'expression de la condition de sélection. Les fonctions d'agrégation et les fonctions de domaine (Taille, Maximum...) en sont donc exclues.

On tolérera cependant un certain nombre de fonctions élémentaires sur les grandeurs (Fannée(date)...).

Exemple : (étude de cas n°1)

A = ANNEE(:AN=Fannée(DATEDEB $_P$))

II.2.1.3. Une relation de dépendance entre variables entité

Le langage de requête proposé au point précédent permet de définir une variable entité à partir d'autres.

Exemple : Reprenons les variables COM et CLI de l'exemple introductif de la figure 2.3. Nous avions COM=COMMANDE(cc:CLI). Cette règle de domaine désigne les commandes passées par le client CLI. On peut dire que la dimension COM dépend de l'entité CLI.

Il existe donc une relation de dépendance entre la variable entité définie et celle(s) qui apparaît(apparaissent) dans sa règle de domaine. Cette dépendance possède des propriétés particulières. En effet, les variables entité sont définies sur des ensembles d'entités de la base de données. Par le schéma qui représente cette dernière, les entités sont reliées par diverses associations. Ce sont ces liens sémantiques supplémentaires dont héritent les variables entité qui permettent de caractériser de manière particulière les dépendances entre celles-ci.

Exemple : si on reprend l'exemple précédent, CLI représente un client et COM une COMMANDE. COM dépend d'une entité CLIENT par la syntaxe du langage. A CLI correspond N COMMANDE par le schéma de la base de données.

Dans cette section, nous allons caractériser la relation de dépendance et montrer en quoi cette étude est un apport important au modèle de spécification.

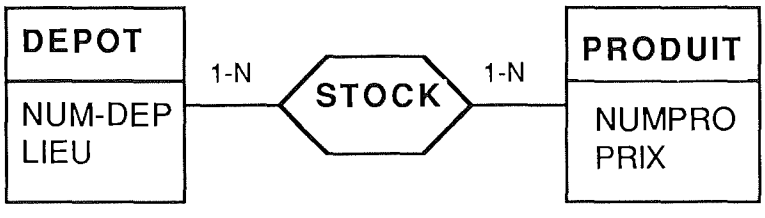
Dans un premier point, nous définirons avec précision la relation de dépendance entre variables entité.

Dans un deuxième point, nous verrons comment, à partir de l'analyse de toutes les dépendances entre variables entité d'un modèle, établir un graphe de dépendances entre variables entité.

Dans un troisième et dernier point, nous introduirons l'utilisation du graphe dans la vérification de cohérence. En effet, le but premier de la construction de ce graphe est de permettre un contrôle de l'emploi des indices dans les règles du problème.

II.2.1.3.1. Définition de la relation de dépendance

Tentons de définir clairement cette dépendance en partant d'un exemple. Soit le schéma Entité/Association suivant :

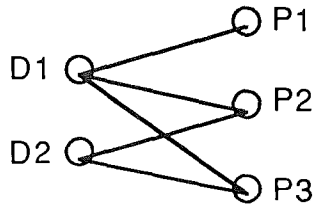


Définissons les variables entité DEP et PRO de la manière suivante:

- DEP = DEPOT
  La règle de domaine désigne l'ensemble des entités de DEPOT
- PRO = PRODUIT (stock : DEP)
  Cette règle désigne les produits stockés dans le dépôt DEP.

On perçoit qu'il existe un lien étroit entre PRO et DEP. Mais quelle est la nature exacte d'une telle relation ?

Prenons un exemple de base de données pour tenter de déceler la relation entre DEP et PRO. Soit (D1, D2), les entités de DEPOT dans la base de données et (P1, P2, P3) les trois entités de PRODUIT.



DEP peut prendre les valeurs D1 et D2.

Si Grd est une grandeur dimensionnée par DEP, on peut écrire :

$$Grd_{DEP} = \begin{vmatrix} Grd_{D1} \\ Grd_{D2} \end{vmatrix}$$

La règle de domaine de PRO est plus complexe à décrire. Du fait que DEP désigne une entité particulière, la règle désigne **un domaine différent** pour les diverses valeurs possibles de DEP. On dira que le domaine de PRO **est paramètré selon** la variable entité DEP (qui en constituera donc le paramètre).

Essayons d'illustrer cette situation en "dépliant" une grandeur indicée par PRO (Grd). Le domaine de PRO est différent selon la valeur de DEP.

Dès lors, $Grd_{PRO} = Grd_{PRODUIT(stock:DEP)}$.

Ce que l'on peut noter en dépliant :

$$Grd_{PRO} = \begin{vmatrix} Grd_{PRODUIT(stock:D1)} \\ Grd_{PRODUIT(stock:D2)} \end{vmatrix}$$

où

$$Grd_{PRODUIT(stock:D1)} = \begin{vmatrix} Grd_{P1,D1} \\ Grd_{P2,D1} \\ Grd_{P3,D1} \end{vmatrix}$$

Remarque importante : il ne faut surtout pas considérer une grandeur indicée par PRO comme une suite de valeurs se rapportant chacune à une valeur de PRODUIT vérifiant la règle de domaine.

$$Grd_{PRO} <> \left| \begin{array}{l} Grd_{P1} \\ Grd_{P2} \\ Grd_{P3} \end{array} \right|$$

A la vue de cet exemple, on peut dire qu'une grandeur indicée par PRO est **implicitement dimensionnée** par DEP. On pourrait dénoter Grd sous la forme $Grd_{PRO,DEP}$. Cependant, la variable PRO est liée à DEP par sa règle de domaine : la notation Grd_{PRO} est donc parfaitement définie et sera toujours utilisée aux dépens de la précédente. Mais l'absence de DEP dans la notation ne doit jamais faire oublier son rôle.

De plus, Grd_{PRO} désigne un attribut (réel ou virtuel) de PRODUIT, car PRO, même si elle est liée à DEP, est définie sur le type d'entités PRODUIT. C'est la deuxième raison pour laquelle on n'exprimera pas explicitement DEP.

En résumé, à une valeur de DEP correspond au plus un domaine de valeurs pour PRO. La relation entre DEP et PRO est celle d'une dépendance fonctionnelle de PRO envers DEP.

Nous pouvons à présent définir la relation de paramètrage :

> Une variable entité Ve est paramètrée selon une variable entité V.
>
> V ⟶ Ve
>
> ssi
>
> Le domaine de Ve dépend fonctionnellement de la valeur de V

Nous pouvons aisément étendre la définition donnée en considérant qu'un déterminant est constitué d'un groupe d'une ou plusieurs variables entité.

Classes fonctionnelles

On distinguera deux classes fonctionnelles de relation :

- relation 1 : la relation ne contient qu'une seule cible.
  (le domaine de Ve contient au plus une valeur pour chaque valeur de V)
- relation N : la relation peut contenir un nombre quelconque de cibles.
  (le domaine de Ve contient un nombre quelconque de valeurs possibles).

Exemples : (étude de cas n°1)

- P = PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV))) et (:DATEDEB ≥ A-D))

  Cette règle de domaine désigne tous les prêts concernant la revue REV qui ont pris cours avant une date donnée A-D.
  A une entité particulière REV correspond un sous-ensemble de valeurs pour P. On a ainsi la relation suivante :

$$REV \xrightarrow{N} P$$

- PA = ANNEEXEM ((ea:EXEMPLAIRE(concerne:P)) et (aae:ANNEE(:AN=Fannee(DATEDEB $_P$))))

  Cette règle désigne toutes les "années-exemplaires" qui concerne le prêt P et dont l'année correspond à l'année du prêt.
  A l'entité P correspond une et une seule valeur d'ANNEEXEM. On peut donc écrire :

$$P \xrightarrow{1} PA$$

II.2.1.3.2. Graphe de dépendances entre variables entité d'un modèle (GDV)

La relation détaillée au point précédent provient de l'analyse d'une seule règle de domaine. Si on décide d'étudier l'ensemble des règles de domaine d'un modèle, on peut mettre en évidence plusieurs liens de dépendances. Ces derniers peuvent être rassemblés dans un graphe unique qui constituera le GDV du modèle.

Nous ne disposons pas encore, à ce stade, d'une méthode d'analyse des règles de domaine. Celle-ci sera présentée dans la suite de ce mémoire. On peut cependant tenter de créer, de manière intuitive, le GDV du modèle calculant le montant dû par un client (figure 2.3.).

Analysons les quatre règles de domaine du modèle les unes à la suite des autres :

1. CLI = CLIENT(:NCLI = N)

   La règle de domaine définissant CLI ne comporte aucune variable entité. Nous pouvons affirmer qu'elle est indépendante.

2. COM = COMMANDE(CC:CLI)

   Le domaine de COM dépend fonctionnellement de l'entité CLI. COM est paramètrée selon CLI. Dans le schéma de la base de données, à une entité CLIENT correspond plusieurs entités du type COMMANDE. Donc, le domaine de COM peut contenir un nombre quelconque de valeurs. La classe fonctionnelle de la relation est N. On peut donc écrire :

$$CLI \xrightarrow{N} COM$$

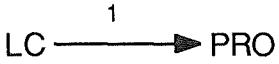
3. LC = LIGNE(CL:COM)

   A une valeur de COM correspond, au plus, un domaine de valeurs pour LC. La variable entité LC est donc paramètrée selon COM. Si on consulte le schéma Entité/Association, à une entité COMMANDE correspond un nombre quelconque de lignes. La classe fonctionnelle de la relation de dépendance de LC vis-à-vis de COM est donc N. On a ainsi :

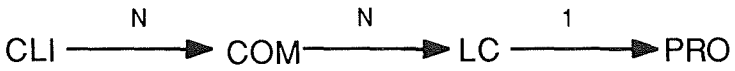
$$COM \xrightarrow{N} LC$$

4. PRO = PRODUIT(PL:LC)

Le domaine de PRO est paramètré selon LC. A une ligne de commande LC correspond un seul PRODUIT. Pour une valeur de LC, le domaine de PRO contiendra au plus une valeur. La classe fonctionnelle de la dépendance est donc 1. On peut écrire :



Après l'analyse des règles de domaine, nous avons mis en évidence trois relations de paramètrage entre les variables entité du modèle. Nous pouvons les rassembler dans un graphe qui constituera le GDV du modèle. On obtient :



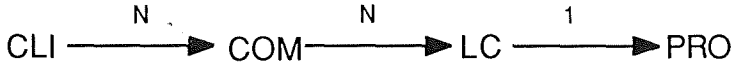
Remarque terminologique : Pour plus de clarté dans la désignation des variables entité dans le GDV, nous utiliserons un certain nombre de concepts de la théorie des graphes :

1. Une relation entre deux Ve correspond a un ARC du graphe orienté.
   Exemple : l'arc (CLI,COM)
2. Soit l'arc (V,Ve)
   Ve est appelé le SUIVANT de V.
   V est appelé le PRECEDENT de Ve.
3. Le POIDS de l'arc (V,Ve) correspond à la classe fonctionnelle de la relation.
   Exemple : Poids(CLI,COM)=N
4. Un CHEMIN (Ve_1, Ve_N) est une suite finies d'arcs {Ve_1, Ve_2,...,Ve_{i-1}, Ve_i, Ve_{i+1},...,Ve_N} telle que pour tout Ve_i, Ve_{i-1} précède Ve_i (avec i dans {2..N}).
5. Ve_1 est un ANCETRE de Ve_2, s'il existe un chemin (Ve_1,Ve_2).
6. Ve_1 est un DESCENDANT de Ve_2, s'il existe un chemin (Ve_2,Ve_1).
7. Le POIDS d'un chemin équivaut au produit (arithmétique) des poids des arcs qui le composent.
   Exemple : Poids(CLI,LC)=N^2

II.2.1.3.3. Utilisation du GDV

Le but du GDV est de permettre un contrôle de cohérence au niveau des indices. Ce contrôle fait l'objet d'une étude précise dans le point II.3.2.1. Nous proposons dès à présent un bref exemple afin de pouvoir mesurer l'impact du GDV.

Reprenons le GDV que nous venons de construire pour l'exemple de la figure 2.3. :



Voici deux erreurs qui seront facilement détectables grâce au graphe :

1. MONTANT-COM $_{COM}$ = TAXE $_{CLI}$ + MONTANT-LIGNE $_{LC}$

Soit TAXE un nouvel attribut de CLIENT.
A une commande correspond bien un client, mais N lignes de commande. Il y a manifestement une erreur qui correspond soit à l'oubli d'une fonction d'agrégation, soit à une mauvaise définition de LC.

2. MONTANT-LIGNE $_{LC} = \sum_{PRO}$ PRIX $_{PRO}$

A une ligne de commande LC correspond un et un seul PRODUIT et donc un seul prix. On détecte ainsi un abus de l'utilisation de la fonction d'agrégation.

II.2.1.4. Les domaines paramètrés

Nous avons vu dans la définition de la relation de dépendance entre variables entité qu'un domaine peut être lié à une dimension qui en constitue donc un paramètre (point II.2.1.3.). Ce paramètre n'apparaît explicitement que dans la règle de domaine.
Mais il est également possible d'adjoindre un **paramètre externe** à la définition d'un domaine. Dans ce cas, il est spécifié explicitement dans le nom de la variable associée au domaine.
La notation que nous allons introduire a pour but de permettre la désignation d'éléments particuliers dans le domaine de valeurs d'une dimension. En effet, pour satisfaire l'objectif de contraction des règles du problème, on a dimensionné les grandeurs. Cette opération, si elle a permis de rassembler les grandeurs en un vecteur unique, supprime la possibilité de désigner un valeur particulière de ce vecteur. Pour permettre "à nouveau" cette opération, il faut introduire un nouveau concept : celui de paramètre externe lié à une dimension.
Exemple : Reprenons le schéma des clients et des commandes (figure 2.2.)
Imaginons une base de données avec les entités CLIENT suivantes:

CLIENT		
NCLI	NOM	ETAT
1	DUPOND	-1000
2	DURAND	0
3	DUVERT	0
4	HIVER	+1000
5	DAMS	0
6	BOND	0

Soit CLI une variable entité tel que CLI = CLIENT.
(La règle de domaine désigne ainsi l'ensemble des entités de CLIENT)
Soit I un paramètre externe associé à CLI par la règle de domaine : CLI(I) = CLI(:NCLI=I).
CLI(I) désigne l'entité CLIENT de numéro I.
On peut écrire grâce à l'introduction du paramètre I :
CLI(2) qui désigne la deuxième entité de CLIENT,
$ETAT_{CLI(4)} = 1000$.

II.2.1.4.1. Définition

Un paramètre externe attaché à une dimension est une grandeur du modèle qui permet de désigner un élément particulier de son domaine selon un ordre prédéfini.

Un paramètre externe peut être attaché aux deux types de dimensions. Le paramètre est associé à sa dimension par une règle de domaine. Nous aborderons dans la suite les paramètres des variables entité et ceux des variables simples. Mais, avant d'entamer ces deux points, examinons quelques **propriétés** importantes des paramètres externes.

Tout d'abord, un paramètre externe n'a pas d'existence propre et est toujours subordonné à une dimension. Dans ce contexte, il ne pourra jamais appartenir aux classes "Données" et "Résultats". Un paramètre externe sera toujours défini dans la classe des **grandeurs internes**.

Ensuite, avec cette notion, on désire désigner les éléments d'un domaine dans un certain **ordre**. Ainsi, lorsque l'on écrira Dimension(1), on veut désigner la première valeur du domaine associé à Dimension. C'est pourquoi, un paramètre sera toujours de type entier et ses valeurs seront comprises entre 1 et le cardinal du domaine sur lequel il est défini.

Enfin, un paramètre peut définir la **portée** d'une règle du problème. Si on veut calculer une grandeur dimensionnée seulement pour certaines valeurs de la dimension, on peut utiliser le concept de paramètre externe afin d'indiquer le sous-ensemble de valeurs pour lesquelles la grandeur doit être calculée.
Exemple : soit la règle suivante du problème introductif de la figure 2.3.

$$MONTANT\text{-}DU_{CLI} = \sum_{COM} MONTANT\text{-}COM_{COM} - ETAT_{CLI}$$

Supposons à présent que pour les dix premières entités de CLIENT, le calcul de MONTANT-DU soit différent. En effet, pour ces derniers, les plus fidèles de la firme, on accorde une remise (REMISE : nouvelle donnée du problème). La notion de paramètre va permettre de différencier les règles de calcul.
Définissons CLI et I comme dans l'exemple précédent : CLI = CLIENT et CLI(I) = CLI(:NCLI=I).
Pour différencier les deux types de CLIENT, on écrira

$$MONTANT\text{-}DU_{CLI(I)} = \sum_{COM} MONTANT\text{-}COM_{COM} - ETAT_{CLI(I)} - REMISE \qquad \mathbf{(I = 1..10)}$$

$$\text{MONTANT-DU}_{CLI(I)} = \sum_{COM} \text{MONTANT-COM}_{COM} - \text{ETAT}_{CLI(I)}$$

(I = 11..card(CLI))

La première règle se rapporte aux clients dont le numéro est compris entre 1 et 10. La seconde se rapporte aux restes des entités CLIENT. L'expression "card(CLI)" désigne le nombre d'entités appartenant au domaine de CLI qui est ici l'ensemble des clients.

En résumé de ces trois grandes propriétés, on peut écrire :

Dans la rubrique "Données, Résultats ou Grandeurs internes" :

Dimension ':' type ';' définition

Dans la rubrique "Grandeurs internes" :

Paramètre ':' entier ';' définition

Dans la rubrique "Règles de domaine" :

Dimension (Paramètre) = Règle de domaine

Dans la rubrique "Règles du problème" :

Grandeur $_{DIMENSION(Paramètre)}$ = règle ,
(Paramètre = a....b)

avec a,b bornes (inférieure et supérieure) appartenant à l'ensemble des entiers et $a \leq b$,
$1 \leq a$,
$b \leq$ card (Dimension) .

II.2.1.4.2. Variable entité paramètrée

Définition

Si on transpose la définition pour les variables entité, les deux objectifs principaux de la notion de paramètre deviennent :
- la désignation d'une entité particulière du domaine,
- "introduire" un ordre sur le domaine.

Afin de satisfaire le premier objectif, le paramètre externe associé à la variable entité doit permettre d'**identifier** celle-ci. A une valeur du paramètre doit correspondre, au plus, une seule entité du domaine.

Exemple : Si on reprend les définitions de CLI et de son paramètre I, on a que pour toute valeur de I comprise entre 1 et card(CLI), CLI(I) désigne une entité unique de CLIENT.

Dans la règle de domaine qui le définit, le paramètre externe doit donc être lié à un identifiant (strict ou non strict) du type de la variable entité. Rappelons que, dans notre restriction binaire du schéma Entité/Association, une entité est identifiée par une combinaison d'attribut(s) et/ou de type(s) d'entités associé(s) à l'entité identifiée.
Exemple : I est lié à CLI par l'attribut identifiant NCLI.

La syntaxe utilisée pour exprimer la règle de domaine d'un paramètre externe est celle que l'on a déjà utilisée pour les variables entité. On aura le schéma suivant :

Dim (Param) = Dim Csel

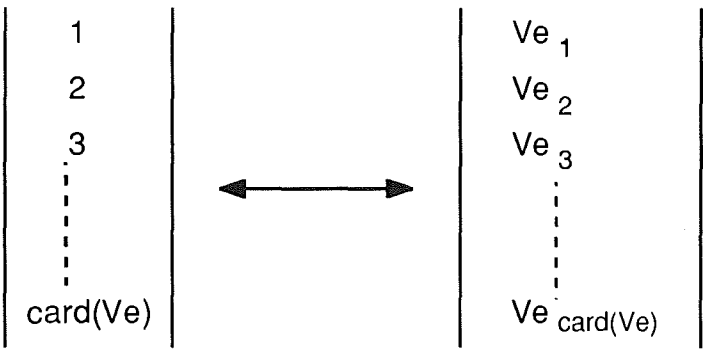
où Dim est la dimension à laquelle on adjoint le paramètre,
Param est le paramètre externe,
Csel est la condition de sélection qui lie le paramètre à un identifiant. Pour lier le paramètre à l'identifiant, on utilisera le mécanisme de la condition d'appartenance.

Notons que, dans ce cas particulier, le nom de la variable entité apparaissant en partie droite désigne toutes les entités du domaine.

Pour introduire **un ordre sur le domaine**, il faut "classer" les entités du domaine. Ce classement doit être strict et est réalisé à l'aide de l'identifiant concerné par la définition du paramètre externe.

Tentons à présent de mettre en oeuvre les deux objectifs d'identification et d'ordre sur le domaine de valeurs d'une variable entité. Dans un premier point, nous analyserons le problème de manière "théorique". Dans un second point, nous illustrerons les concepts par une série d'exemples.

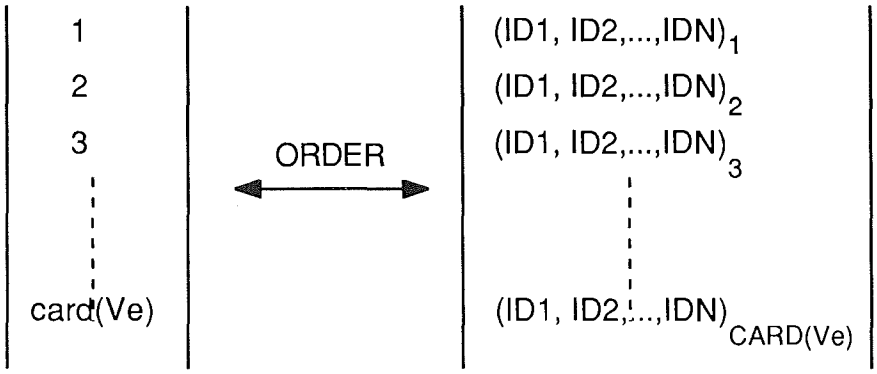
Soit Ve une variable entité de type T.
Soit I un paramètre externe lié à Ve par l'identifiant (ID1, ID2, ... , IDN) où IDj est soit un attribut de T, soit un type d'entités associé à T. Par définition d'un parmètre externe, I est compris entre 1 et card(Ve).
A chaque valeur de I, il faut faire correspondre une entité unique du domaine de Ve. Pour cela, le système de gestion de modèles devra automatiquement établir une **table de correspondance** entre les valeurs de I et les valeurs de Ve. Cette table de correspondance établit ainsi une **bijection** entre paramètre et valeurs du domaine.
On procède comme suit :



où Ve_i désigne une entité particulière du domaine de Ve.

Pour classer les valeurs de Ve, on peut spécifier un tri sur les identifiants. Ceux-ci permettront de ranger les valeurs possibles de Ve de manière stricte. Si on ne spécifie pas de clé de tri, c'est l'ordre chronologique d'entrée des entités dans la base de données qui est pris en compte.

Deux cas sont dès lors à envisager : soit le système crée la table de correspondance en ayant préalablement trié les entités du domaine de Ve sur base de l'identifiant, soit aucun tri n'est spécifié et la table est établie selon l'ordre d'insertion des entités dans la base de données. Appelons **ORDER** et **NORDER**, les deux fonctions prédéfinies de construction de tables de correspondance respectivement avec et sans tri préalable. Si on veut un tri préalable, on obtient :



où $(ID1, ID2,..., IDN)_i$ "précède" $(ID1, ID2,... ,IDN)_k$ avec $i < k$

On a ainsi que $ORDER(1) = (ID1, ID2,...IDN)_1$.

Si on veut désigner le deuxième élément d'un vecteur d'identifiants, on écrit $ORDER(1,2) = (ID2)_1$.

Remarque : nous n'insisterons pas sur le type de tris (croissant, décroissant...) que pourrait spécifier l'utilisateur. Nous supposerons seulement que la fonction ORDER trie le domaine de la variable entité et que l'utilisateur peut spécifier exactement le tri désiré. Une syntaxe plus précise sera donc nécessaire dans des développements ultérieurs.

Nous allons, à présent, envisager un certain nombre d'**exemples** où la notion de table de correspondance est mise en oeuvre :

Cas où l'identifiant est un attribut de type entier et où toutes ses valeurs pour les entités du domaine se suivent par pas de un

Cette première hypothèse, peu fréquente, est la plus simple. Il suffit d'égaler l'attribut identifiant au paramètre dans la règle de domaine. Dans ce cas particulier, identifiant et paramètre sont confondus ; il n'est donc pas nécessaire d'établir une table de correspondance.

Exemple : Nous pouvons reprendre notre exemple introductif des domaines paramètrés où l'attribut NCLI est identifiant de CLI, est de type entier et a toutes ses valeurs qui se suivent par pas de un.

CLI = CLIENT

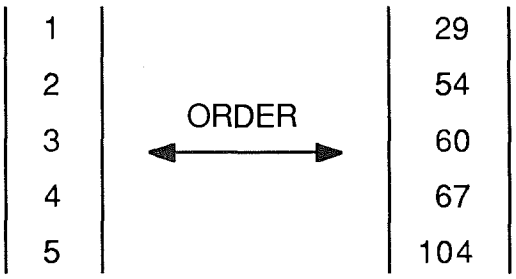
CLI(I)=CLI(:NCLI = I)

Dans la règle de domaine, on "égalise" le paramètre à la valeur de l'identifiant.

Cas où l'identifiant est un attribut de type numérique et où ses valeurs pour les entités du domaine ne se suivent pas par pas de un

Dans ce cas, le système devra construire une table de correspondance qui fera la bijection entre l'ensemble des valeurs de l'identifiant et l'ensemble des entiers compris entre 1 et le cardinal du domaine de la variable entité. Toutefois, avant de construire cette table, on peut convenir de trier les valeurs de l'identifiant.

Exemple : Supposons que, dans l'entité CLIENT, l'ensemble des valeurs de NCLI est (67,29,104,60,54). L'utilisateur veut lier un paramètre externe à CLI (I) afin de pouvoir désigner ses valeurs particulières par ordre croissant de numéro. Le système doit donc établir la table suivante :



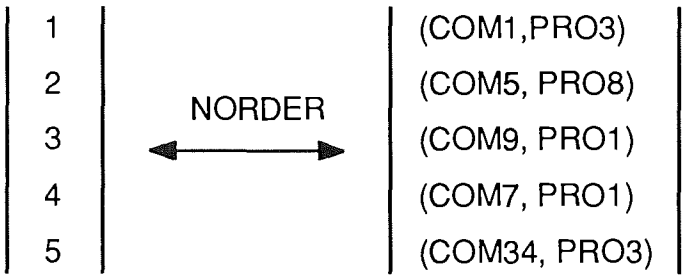
Dans la règle de domaine qui définit le paramètre externe, l'utilisateur devra employer la fonction ORDER qui indiquera au modèle qu'il faut établir une table de correspondance avec tri préalable.

Exemple : L'utilisateur devra donc définir la règle de domaine suivante : CLI(I)=CLI(:NCLI=ORDER(I)).
Si on utilise CLI(1), cela existe et c'est le client dont le numéro (NCLI) est égal à 29.

Cas où l'identifiant est une combinaison de types d'entités

Repartons du schéma entité/association des clients et des commandes de la figure 2.2. Supposons que l'on a défini la variable entité LIG de la manière suivante : LIG = LIGNE(CL:COMMANDE(CC:CLI)). Cette règle de domaine désigne l'ensemble des lignes de commandes du client CLI. Supposons à présent que l'utilisateur veut désigner des valeurs particulières de LIG. Il ne désire pas que celles-ci soient triées.

Pour désigner une entité particulière de LIG, le paramètre externe utilisé devra être lié à un identifiant. LIGNE est identifiée par les types d'entités COMMANDE et PRODUIT. Ecrivons la table de correspondance que le système devra établir :



où COMj désigne une entité de COMMANDE et PROk une de PRODUIT

Dans la règle de domaine, l'utilisateur doit lier le paramètre externe à l'identifiant. Un problème se pose pour désigner les entités particulière COMj et PROk dans la requête. Il faut recourir de nouveau au mécanisme d'identification pour les types d'entités COMMANDE et PRODUIT . On écrira la règle de domaine suivante :

LIG(I)=LIG((CL:COMMANDE(:NCOM=NORDER(I,1)) et
(PL:PRODUIT(:NPRO=NORDER(I,2))))

Remarque : *Propagation du paramètre entre variables entité*

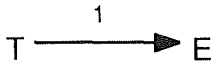
On accepte de **propager** le paramètre **à travers le graphe de dépendances entre variables entité**. C'est-à-dire que l'on peut paramètrer une autre variable entité avec ce paramètre uniquement dans le cas où les deux variables entité sont reliées par un chemin de poids 1.

Exemple : (étude de cas n°3)

T(I) : Ième tronçon

E = ESCALE-VOL(de:T)

On a dans le GDV la relation suivante :



Alors on peut employer E(I) dans les règles sans l'avoir défini.

II.2.1.4.3. Variable simple paramètrée

Au niveau des variables simples, la définition d'un paramètre externe est moins complexe. Dans ce cas, il ne faut pas introduire un ordre sur le domaine de la variable simple, car la règle de domaine de celle-ci le fournit déjà.

Pour les variables simples, nous avons distingué trois types de règles de domaine (point II.2.1.1.2.). Pour les deux premiers, on donne explicitement la suite de valeurs de la variable simple. Cette suite de valeurs distinctes définit un ordre sur le domaine. Il n'est donc pas nécessaire d'établir une table de correspondance.

Exemple : AN = 1987, 1988, 1989, 1990, 1991

I : entier ; paramètre externe de AN

AN(I) : Ième année de la suite de valeurs de AN

AN(3) = 1989

Pour le dernier cas où la variable simple dépend d'une autre variable simple, l'ordre considéré est celui de la suite de valeurs sur laquelle elle est définie. On itère le processus jusqu'à ce que la variable simple ait sa suite de valeurs explicitement donnée dans une règle de domaine.

Exemple : AN = 1987, 1988, 1989, 1990, 1991

ANB = (AN / AN > 1988)

ANB(2) = 1990

II.2.2. Les grandeurs

Nous appelerons grandeurs l'ensemble des grandeurs du modèle qui ne sont pas des dimensions. En fait, les grandeurs que nous allons détailler à présent interviennent "directement" dans la détermination du résultat. **Ces grandeurs expriment ainsi la règle de calcul du (des) résultat(s)**. Il n'en est pas de même des dimensions qui servent plutôt de notation afin de rassembler une suite de grandeurs dans un vecteur (la grandeur dimensionnée).

Exemple : reprenons à nouveau l'exemple introductif de la figure 2.3. et illustrons le rôle différencié joué par les grandeurs et les dimensions. Le modèle possède trois grandeurs MONTANT-DU $_{CLI}$, MONTANT-COM $_{COM}$ et MONTANT-LIGNE $_{LC}$. Ces grandeurs interviennent directement dans le calcul du résultat. Ainsi, pour connaître le montant dû par le client, on somme le montant de chacune de ses commandes. Et, pour calculer ce dernier, on somme le montant de chacune de ses lignes. Les deux grandeurs MONTANT-COM $_{COM}$ et MONTANT-LIGNE $_{LC}$ sont révélatrices de la démarche de calcul que le concepteur a suivie pour déterminer le résultat. Les dimensions CLI,COM,LC et PRO ne sont en rien révélatrices de la solution. On se rend compte à ce niveau de leur rôle artificiel par rapport au réel perçu. Elles constituent donc essentiellement une notation pour faciliter l'expression de la solution et intégrer modèle/base de données.

Cette dichotomie sera également mise en évidence lors de l'élaboration du graphe de dépendances entre grandeurs et de sa confrontation avec celui des variables entité (II.2.2.6.).

Les grandeurs du modèle sont définies par les **règles du problème**. Quelle que soit la catégorie de grandeurs, ces règles respectent une syntaxe unique. C'est pourquoi, nous débuterons ce point en décrivant les règles de définition des grandeurs (II.2.2.1.)

Après avoir détaillé les règles du problème, nous classerons les grandeurs en quatre grandes catégories. Cette classification est basée sur le type d'indices éventuels qui dimensionne la grandeur. Dans la section précédente, nous avons distingué deux types de dimensions : les variables entité et les variables simples. A la lumière de cette distinction, nous diviserons les grandeurs en grandeurs dimensionnées par une variable entité (II.2.2.2.), grandeurs dimensionnées par une ou plusieurs variables simples (II.2.2.3.), grandeurs dimensionnées par les deux types de dimensions (les **grandeurs mixtes** : II.2.2.4.) et grandeurs non dimensionnées (les **grandeurs simples** : II.2.2.5.).

Nous analyserons en détail chacune des catégories en essayant de mettre en évidence leurs caractéristiques propres. Nous examinerons également leur statut en utilisant la classification habituelle : données, résultats et grandeurs internes.

Pour terminer l'analyse, nous présenterons un graphe de dépendances entre grandeurs. Nous montrerons comment le créer et examinerons ses rapports avec le GDV.

II.2.2.1. Les règles du problème

Les règles du problème et les grandeurs sont des notions indissociables dans notre modèle de spécification. Une grandeur apparaît toujours dans une règle soit en étant définie[1] par celle-ci, soit en intervenant dans la règle de définition d'une autre grandeur. En ce qui concerne les grandeurs, la syntaxe des règles est unique et nous la présentons avant toute analyse préalable de celles-ci.

Notre modèle de spécification accepte deux types de règles[2] : celles à définition unique et celles à définitions multiples. Détaillons les caractéristiques de chacune d'elles.

Règle à définition unique[3]

Elle a la forme générale suivante :

GRANDEUR '=' EXPRESSION

Les différentes formes d'une expression sont :
- soit une grandeur,
- soit un entier, un réel, un booléen, ...
- soit une combinaison des deux formes précédentes liées par des opérateurs.

Quels sont les opérateurs utilisés ? Pour la plupart, il s'agit des opérateurs arithmétiques habituels (+,-,/,Π,Σ). Il existe des priorités entre ceux-ci. La **table des priorités** nous les rappelle :

i \ j	+	-	*	/	Σ	Π
+	=	=	<	<	<	<
-	=	=	<	<	<	<
*	>	>	=	=	<	<
/	>	>	=	=	<	<
Σ	>	>	>	>	=	=
Π	>	>	>	>	=	=

opi = opj s'ils ont la même priorité,
opi < opj si opj est prioritaire,
opi > opj si opi est prioritaire.

---

[1] On utilisera indifféremment les expressions "grandeur expliquée" ou "grandeur définie" par une règle.

[2] En réalité, il y a trois types de règles. Les règles de contrainte seront analysées au point II.2.4.

[3] Par la suite, on utilisera tout simplement le vocable de REGLE.

Pour expliquer la notion de **priorité**, utilisons un exemple :

$$\sum_{COM} \text{MONTANT-COM}_{COM} + \text{FRAIS}_{COM} * \text{REDUCTION}$$

Comment évaluer cette expression ? L'opérateur Σ "est le plus prioritaire" ; on évaluera $\sum_{COM} \text{MONTANT-COM}_{COM}$; ensuite on calculera FRAIS_{COM} * REDUCTION (la multiplication étant prioritaire par rapport à l'addition). Pour terminer, on additionnera les deux membres.

L'utilisation des parenthèses va permettre de changer l'ordre d'évaluation des éléments d'une règle en modifiant les priorités. Par exemple,

$$\sum_{COM} (\text{MONTANT-COM}_{COM} + \text{FRAIS}_{COM}) * \text{REDUCTION}$$

La fonction d'agrégation porte sur $(\text{MONTANT-COM}_{COM} + \text{FRAIS}_{COM})$. On effectue d'abord l'addition, on agrège et, finalement, on multiplie.

Dans la table des priorités, certains opérateurs sont sur le même pied (+ et -, * et /, Σ et Π). Pour éviter toute ambiguïté, on utilisera **l'associativité des opérateurs à gauche** lorsqu'on ne sait quel opérateur prendre en compte. Ainsi, l'expression $\text{MONTANT-COM}_{COM} + \text{FRAIS}_{COM}$ - REDUCTION est équivalente à $(\text{MONTANT-COM}_{COM} + \text{FRAIS}_{COM})$ - REDUCTION.

Maintenant, attardons-nous quelque peu sur les fonctions d'agrégation (la somme Σ et le produit Π). Ces fonctions, comme leur nom l'indique, ont pour but d'agréger, de rassembler plusieurs objets en un seul. Une agrégation s'effectue toujours sur un indice, appelé **indice d'agrégation**, qui est soit une dimension, soit un paramètre externe lié à une dimension.

Nous allons déplier un règle pour bien montrer cette notion.

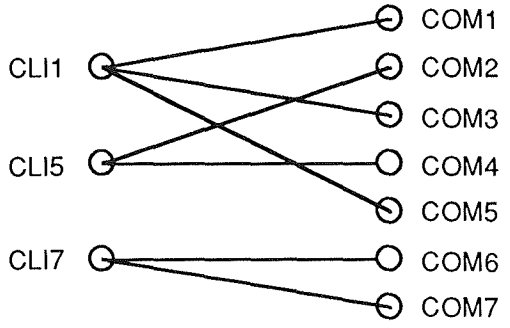
Prenons la règle de la figure 2.3. :

$$\text{MONTANT-DU}_{CLI} = \sum_{COM} \text{MONTANT-COM}_{COM}$$

Le montant de la facture du client est égal à la somme des commandes qu'il a passées. On prend comme domaine de valeurs pour les dimensions :

CLI = CLIENT(:NCLI < 10) = (CLI1,CLI5,CLI7)
COM = COMMANDE(CC:CLI)
= (COM1,COM2,COM3,COM4,COM5,COM6,COM7)

Soit la base de données :



Si on déplie la règle, on obtient :

$$\begin{vmatrix} \text{MONTANT-DU}_{\text{CLI1}} \\ \text{MONTANT-DU}_{\text{CLI5}} \\ \text{MONTANT-DU}_{\text{CLI7}} \end{vmatrix} = \begin{vmatrix} \sum_{\text{COMMANDE(cc:CLI1)}} \text{MONTANT-COM}_{\text{COM,CLI1}} \\ \sum_{\text{COMMANDE(cc:CLI5)}} \text{MONTANT-COM}_{\text{COM,CLI5}} \\ \sum_{\text{COMMANDE(cc:CLI7)}} \text{MONTANT-COM}_{\text{COM,CLI7}} \end{vmatrix}$$

Plus explicitement,

$$\begin{vmatrix} \text{MONTANT-DU}_{\text{CLI1}} \\ \text{MONTANT-DU}_{\text{CLI5}} \\ \text{MONTANT-DU}_{\text{CLI7}} \end{vmatrix} = \begin{vmatrix} \text{MONTANT-COM}_{\text{COM1}} + \text{MONTANT-COM}_{\text{COM3}} + \text{MONTANT-COM}_{\text{COM5}} \\ \text{MONTANT-COM}_{\text{COM2}} + \text{MONTANT-COM}_{\text{COM4}} \\ \text{MONTANT-COM}_{\text{COM6}} + \text{MONTANT-COM}_{\text{COM7}} \end{vmatrix}$$

Si on veut effectuer la fonction d'agrégation sur certaines valeurs d'une dimension, l'indice d'agrégation sera le paramètre externe lié à cette dimension. Ainsi, dans notre exemple, si on désire calculer, pour chaque client, la somme des deux premières commandes qu'il a passées, on écrira la règle suivante :

$$\text{MONTANT-DU}_{\text{CLI}} = \sum_{i=1}^{2} \text{MONTANT-COM}_{\text{COM(i)}}$$

L'utilisation du paramètre externe comme indice d'agrégation permet de limiter le domaine d'application de la fonction d'agrégation en déterminant les valeurs de la dimension qui seront prises en compte dans la règle.

Règle à définitions multiples

Elle a la forme générale :

GRANDEUR '=' EXPRESSION1 si CONDITION1
EXPRESSION2 si CONDITION2
...
EXPRESSIONp si CONDITIONp

GRANDEUR reçoit plusieurs expressions de définition selon le contexte dans lequel elle est définie. Les expressions (EXPRESSIONi) ont la même forme que celles des règles à définition unique.Les conditions (CONDITIONi) définissent les contraintes à respecter pour expliquer GRANDEUR par EXPRESSIONi. Les différentes formes d'une condition sont :

- une comparaison d'expressions avec des opérateurs (<,>,=,≥,≤).
- une condition par défaut (sinon).
- une combinaison de comparaisons d'expressions avec des opérateurs logiques (et/ou).

Prenons une règle à définitions multiples de la deuxième étude de cas :

ECONOMIE $_{REG}$ = bonne si CROISSANCE $_{REG} \geq 0.04$ ou
CHOMAGE $_{REG} < 0.076$
moyenne si $0.02 \leq$ CROISSANCE $_{REG} < 0.04$ et
$0.076 \leq$ CHOMAGE $_{REG} < 0.082$
mauvaise si CROISSANCE $_{REG} < 0.02$ ou
CHOMAGE $_{REG} \geq 0.082$

($0.02 \leq$ CROISSANCE $_{REG} < 0.04$) est une comparaison de trois expressions, (CROISSANCE $_{REG} < 0.02$ ou CHOMAGE $_{REG} \geq 0.082$) est une combinaison employant l'opérateur logique "ou".

Pour terminer cette présentation des règles à définitions multiples, précisons que, dans le but de satisfaire notre objectif pédagogique, on décide de ne pas utiliser de fonctions d'agrégation dans l'expression des conditions. Cette limite peut être aisément contournée en utilisant une grandeur interne définie par une règle contenant la fonction d'agrégation.

II.2.2.2. Les grandeurs dimensionnées par une variable entité

Définition

Une grandeur dimensionnée par une variable entité est une grandeur mesurable à laquelle on peut associer une suite de valeurs d'attribut attachée à l'ensemble d'entités représenté par la variable entité.

Le lecteur pourrait se demander pourquoi une grandeur ne peut être dimensionnée que par une seule variable entité. La définition répond directement à cette question. On se trouve en face de grandeurs soit attributs **réels**, soit attributs **virtuels** associés à un seul type d'entités. Il est inconcevable d'associer un attribut à plusieurs types d'entités même si, implicitement, il peut être dimensionné par d'autres variables entité (cfr relation de dépendances entre variables entité : point II.2.1.3.).

Suivant la catégorie (données, résultats ou grandeurs internes) dans laquelle elles ont été définies, les grandeurs dimensionnées par une variable entité peuvent jouer des rôles différents. Examinons leur statut dans chacune des catégories.

Données

Une grandeur dimensionnée par une variable entité est une donnée du modèle, si l'utilisateur doit l'initialiser pour connaître les résultats. Une telle grandeur apparaît uniquement en partie droite d'au moins une règle. Distinguons deux cas suivant que la grandeur est un attribut réel ou virtuel d'une base de données.

Si c'est une donnée d'une base de données, il s'agit d'un attribut réel du type d'entités de la variable entité. Dans ce cas, la grandeur n'apparaît pas dans la rubrique "Données :".

Exemple : (Etude de cas n°2)

PRIX $_{PRO}$ est un attribut de l'entité VENTEREG dans la base de données. Il ne figure pas dans la définition des données.

Seules les données qui ne sont pas des attributs réels d'une base de données figurent dans la rubrique "Données :". Cela pose toutefois certains problèmes.
Exemple : (Etude de cas n°2)
QUOTANOUVPRO $_{PRO}$ est une donnée du problème. C'est le quota de vente du nouveau produit PRO.

Dans ce cas, l'acquisition de la donnée dépend de la détermination de certaines grandeurs du modèle. Une partie des règles doit être évaluée pour permettre l'acquisition de la donnée. Nous supposerons donc l'existence d'une fonction capable de calculer les éléments strictement nécessaires à l'acquisition et de guider l'utilisateur dans l'introduction des valeurs de la donnée dimensionnée par la variable entité. Cette fonction s'appelera ACQUISITION.

Si on reprend l'exemple, pour introduire les valeurs de QUOTANOUVPRO$_{PRO}$, la fonction doit calculer le domaine de PRO. Pour déterminer celui-ci, on a également besoin d'évaluer le domaine de REG. Ensuite, la fonction guide l'utilisateur en proposant une valeur de PRO et de REG pour qu'il introduise une valeur de QUOTANOUVPRO.

Il est important de signaler l'obligation d'exécuter cette fonction pour introduire les données. C'est pourquoi, dans la rubrique "Données :", toute grandeur dimensionnée par une variable entité n'étant pas un attribut réel de la base de données sera accompagnée de la déclaration de la fonction d'acquisition ayant comme paramètre la variable entité. Nous adopterons la notation suivante :

Grandeur $_{Ve}$ ':' Type ';' définition en français

avec ACQUISITION(Ve)

Résultats

Une grandeur dimensionnée par une variable entité est un résultat, si elle est solution du modèle. Dans la rubrique "Résultats" :

Grandeur $_{Ve}$ ':' Type ';' définition en français

Dans la rubrique "Règles du problème" :

Grandeur $_{Ve}$ '=' Expression (règle à définition unique)
ou
Grandeur $_{Ve}$ '=' Ensexpressions (règle à définitions multiples)

Ces grandeurs sont obligatoirement définies par une règle. Elles peuvent également faire partie de la règle de définition d'une autre grandeur dans les cas de récurrence, récursivité[1] ou de détermination d'un autre résultat.
Exemples : - (Calcul du montant des commandes d'un client : figure 2.3.)
MONTANT-DU $_{CLI}$ est le montant de la facture du client CLI, le résultat calculé par le modèle.

[1] Les notions de récurrence et récursivité seront traitées plus en détail dans le point Règles particulières (II.2.3.).

- (Etude de cas n°2)
  PUB $_{REG}$ et NOUVQUOTA $_{REP}$ sont les deux résultats calculés par le modèle. Il s'agit d'un modèle récursif car PUB $_{REG}$ est défini à partir de NOUVQUOTA $_{REP}$ et NOUVQUOTA $_{REP}$ est défini à partir de FACTEURPUB $_{REG}$ dont la définition dépend de PUB $_{REG}$.

Grandeurs internes

Une grandeur dimensionnée par une variable entité est une grandeur interne si elle n'est ni un résultat attendu, ni une donnée du modèle. La définition des grandeurs internes est identique à celle des résultats.

Une telle grandeur est définie par une règle. Elle intervient dans au moins une règle de définition d'une autre grandeur.

Exemple : MONTANT-COM $_{COM}$ est une grandeur interne représentant le montant d'une commande permettant de calculer le résultat MONTANT-DU $_{CLI}$. Elle a sa propre règle de définition et elle participe à la règle de définition du résultat.

II.2.2.3. Les grandeurs dimensionnées par une ou plusieurs Variables simples

Afin de mieux saisir le rôle de ce type de grandeurs, nous proposons de reprendre l'exemple du calcul du montant de chaque commande. Mais, cette fois-ci, on ne dispose pas de base de données. Le modèle se présente de la sorte :

Données :
CO : entier ; commandes passées à la firme
PR : entier ; produits qui peuvent être commandés
Q $_{CO,PR}$: réel ; quantité commandée du produit PR dans la commande CO
FRAIS $_{CO}$: réel ; frais concernant la commande CO
PRIX $_{PR}$: réel ; prix du produit PR
Résultat :
MONTANT-COM $_{CO}$: réel ; montant total de chacune des commandes
Grandeur Interne :
MONTANT-LC $_{CO,PR}$: réel ; montant d'une ligne de commande concernant la commande CO et le produit PR
Règles de domaine :
CO = co1, co2, ..., coN
PR = pr1, pr2, ..., prM
Règles du problème :

$$\text{MONTANT-COM}_{CO} = \sum_{PR} (\text{MONTANT-LC}_{CO,PR}) + \text{FRAIS}_{CO}$$

$$\text{MONTANT-LC}_{CO,PR} = Q_{CO,PR} * \text{PRIX}_{PR}$$

Figure 2.4. : Calcul du montant des commandes

Nous sommes en présence d'un modèle dimensionné sans variable entité. Puisque ce modèle ne repose sur aucune base de données, tous les éléments nécessaires aux calculs sont définis dans la rubrique "Données". Nous avons besoin de renseignements sur les commandes et les produits pouvant être

commandés. Pour ce faire, deux variables simples (PR et CO) ont été définies pour spécifier les domaines de valeurs des grandeurs du modèle.

Définition

Une grandeur dimensionnée par une ou plusieurs variables simples est une grandeur mesurable à laquelle on peut associer une suite de valeurs. Ces dimensions décrivent le domaine d'application de la grandeur.

Les grandeurs dimensionnées par des variables simples peuvent appartenir à une des trois catégories.

Données

Ces grandeurs apparaissent uniquement dans la règle de définition d'autres grandeurs et, en aucun cas, ne sont des attributs d'une base de données.
Exemple : $Q_{CO,PR}$ est la quantité commandée du produit PR dans la commande CO.

Si la grandeur dimensionnée par une ou plusieurs variables simples dépend de variables qui sont des données, on n'a pas besoin de fonction d'acquisition évaluant ces grandeurs avant l'exécution du modèle. Dans la rubrique "Données", on aura :

Grandeur $_{Vs1,Vs2,\ ...,\ VsN}$ ':' type ';' définition

avec Vs1, Vs2, ..., VsN : des variables simples différentes.

Si une ou plusieurs variables simples dimensionnant la grandeur sont des grandeurs internes, elles doivent être calculées par la fonction d'acquisition. La syntaxe est la même que pour les grandeurs dimensionnées par une variable entité.

Dans l'exemple introductif (figure 2.4.), en supposant que le modèle est toujours calculé pour les mêmes produits (PR devient une grandeur interne), on aura dans la rubrique "Données :" :

CO : entier ; commandes passées par tous les clients (identifiées par un numéro)
$Q_{CO,PR}$: réel ; quantité commandée du produit PR dans la commande CO
avec ACQUISITION(PR)
$FRAIS_{CO}$: réel ; frais concernant la commande CO
$PRIX_{PR}$: réel ; prix du produit PR
avec ACQUISITION(PR)

Résultats et grandeurs internes

Dans les rubriques "Résultats" ou "Grandeurs internes" :

Grandeur $_{Vs1,Vs2, ..., VsN}$ ':' Type ';' définition

Dans la rubrique "Règles du problème" :

Grandeur $_{Vs1,Vs2, ..., VsN}$ = Expression
ou
Grandeur $_{Vs1,Vs2, ..., VsN}$ = Ensexpressions

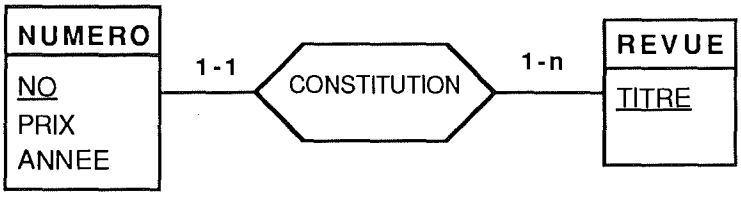
Exemples : - MONTANT-COM $_{CO}$ est le résultat dimensionné par CO. Le domaine d'application est une suite de valeurs.
- MONTANT-LC $_{CO,PR}$ est une grandeur interne utile au calcul de MONTANT-COM $_{CO}$.

II.2.2.4. Les grandeurs mixtes

Présentons tout d'abord un exemple qui calcule le coût d'abonnement à une revue depuis une année donnée. Ce modèle est inspiré de la première étude de cas. Malgré son caractère artificiel, celui-ci permettra de mettre en évidence le concept de grandeur mixte.

Le modèle calcule également l'écart par rapport au coût moyen de la revue sur les années concernées. Pour chaque année, on va calculer la différence entre le coût de la revue pour cette année-là et le coût moyen de la revue (sur la période d'analyse). Dans la base de données, certaines informations sont absentes, comme les réductions accordées et les amendes pour retard de paiement. Nous avons aussi introduit une variable simple AN désignant les années d'analyse. Avec la variable entité REV, elle va dimensionner certaines grandeurs. On parlera de grandeurs mixtes.

Voici le schéma de la base de données et le modèle :



Données :
AN : entier ; années sur lesquelles va porter l'analyse
NOMR : string ; nom de la revue objet de l'analyse
RED $_{AN}$: réel ; réduction accordée chaque année sur les abonnements aux revues par la ministère de la culture
AMENDE $_{AN,REV}$: réel ; amende pour retard de payement à l'année AN pour la revue REV
avec ACQUISITION(REV)

Résultats :
COUT-PAIEMENT $_{REV}$: réel ; coût de l'abonnement à la revue REV pour les années AN
ECART $_{AN,REV}$: réel ; écart entre le coût moyen d'un abonnement à la revue REV et le montant de REV à l'année AN

Grandeurs internes :
REV : REVUE ; revue de nom NOMR
N : NUMERO ; numéro de la revue REV
MONTANT-REV $_{AN,REV}$: réel ; montant de l'abonnement à la revue REV pour l'année AN
MONTANT-NUM $_{AN,N}$: réel ; prix du numéro N pour l'année AN
Règles de domaine :
AN = a1,a2,a3,...,a**M**
REV = REVUE(:TITRE=NOMR)
N = NUMERO((constitution:REV) et (:ANNEE=AN))
Règles du problème :

$$\text{COUT-PAIEMENT}_{REV} = \sum_{AN} \text{MONTANT-REV}_{AN,REV}$$

$$\text{ECART}_{AN,REV} = \text{MONTANT-REV}_{AN,REV} - (\text{COUT-PAIEMENT}_{REV} / M)$$

$$\text{MONTANT-REV}_{AN,REV} = \sum_{N} \text{MONTANT-NUM}_{AN,N} + \text{AMENDE}_{AN,REV}$$

$$\text{MONTANT-NUM}_{AN,N} = \text{PRIX}_{N} * \text{RED}_{AN}$$

Figure 2.5. : Calcul du coût d'abonnement à une revue

Définition

Une grandeur mixte est une grandeur mesurable dont les dimensions décrivent le domaine d'application. Celui-ci est rattaché à un type d'entités représenté par la variable entité.

Intuitivement, de par la variable entité qui la dimensionne, une grandeur mixte peut être vue comme **un attribut d'une association virtuelle** entre un type d'entités et une ou plusieurs variables simples. Il est répétitif à cause du domaine d'application décrit par les variables simples. Pour la même raison que les grandeurs dimensionnées par une variable entité, une seule dimension attachée à la base de données est permise.

Voyons d'abord comment représenter ces grandeurs dans les trois catégories:

Données

Une grandeur mixte, étant toujours un attribut virtuel de la base de données (à cause des variables simples), doit figurer dans la rubrique "Données :" et être introduite par l'utilisateur. Puisqu'elle a comme dimension une variable entité, sa déclaration est toujours accompagnée de la fonction d'acquisition ayant comme paramètre la variable entité et toutes les variables simples qui ne sont pas des données.

Dans la rubrique "Données :" :

Grandeur $_{Vs1,Vs2, ..., VsN,Ve}$ ':' Type ';' définition en français

avec ACQUISITION(Ve,...)

où Vs1, Vs2, ..., VsN sont des variables simples différentes et Ve une variable entité.

Exemple : AMENDE $_{AN,REV}$ est une grandeur mixte (donnée) qui serait un attribut répétitif virtuel du type d'entités REVUE. A une revue correspondrait M valeurs de l'attribut AMENDE (une par année).

Résultats et grandeurs internes

Dans les rubriques "Résultats" ou "Grandeurs internes" :

Grandeur $_{Vs1,Vs2, ..., VsN,Ve}$ ':' Type ';' définition en français

Dans la rubrique "Règles du problème" :

Grandeur $_{Vs1,Vs2, ..., VsN,Ve}$ = Expression
ou
Grandeur $_{Vs1,Vs2, ..., VsN,Ve}$ = Ensexpressions

Exemples : - ECART $_{AN,REV}$ est un résultat. Pour chaque année (an1, ..., anM), cette grandeur calcule ce qu'on a payé en plus pour l'abonnement à la revue REV par rapport au coût d'abonnement moyen.
- MONTANT-NUM $_{AN,N}$ est une grandeur interne mixte. On peut la voir comme un attribut virtuel répétitif du type d'entités NUMERO. A un numéro correspond une suite de valeurs représentant le prix annuel du numéro.

II.2.2.5. Les grandeurs simples

Un exemple typique de modèle simple est le calcul des racines d'une équation du second degré vue au point II.1.2.1. (figure 2.1.).

Définition

Une grandeur simple est une grandeur à laquelle on peut associer une valeur à tout instant. Elle prend ses valeurs dans un ensemble appelé domaine de valeurs.

Ces grandeurs peuvent être définies dans les trois catégories habituelles :

Données

Dans les rubriques "Données" :

Grandeur ':' Type ';' définition en français

Exemple : A, B, C sont des données ayant une seule valeur à tout instant.

Résultats et grandeurs internes

Dans la rubrique "Résultats" ou "Grandeurs internes :" :

Grandeur ':' Type ';' définition en français

Dans la rubrique "Règles du problème" :

Grandeur '=' Expression
ou
Grandeur '=' Ensexpressions

Exemples : - X_1 et X_2 sont des grandeurs simples résultats
- ρ est la grandeur interne du modèle.

II.2.2.6. Graphe de dépendances des grandeurs

Au point II.2.1.3., nous avons montré l'existence d'un graphe de dépendances entre variables entité (GDV) d'un modèle. Il est également possible de créer un graphe pour les grandeurs que nous venons d'analyser.

D'abord, nous allons montrer comment construire un tel graphe. Nous citerons également quelques utilisations possibles de ce dernier. Enfin, nous tenterons de lier les deux graphes de dépendances : celui des variables entité et celui des grandeurs.

En début d'analyse du concept de grandeur, nous avons insisté sur leur rôle dans l'expression de la règle de calcul du (des) résultat(s). Un graphe reprenant les dépendances entre grandeurs permettra de mettre en évidence les **structures d'un modèle** et de saisir la solution proposée par les concepteurs.

Construction et utilisation du graphe de dépendances

L'**établissement du graphe** de dépendances à partir des règles du problème est relativement aisé. On dira qu'une grandeur A dépend **directement** d'une grandeur B si elle intervient dans la règle de définition de A. Nous représenterons cette dépendance par un arc orienté en trait plein qui part de B et aboutit à A. On dira également qu'une grandeur A dépend **indirectement** d'une grandeur B si A est expliqué par une règle à définitions multiples et si B intervient dans une ou plusieurs conditions. On représentera cette dépendance par un arc orienté en pointillé partant de B et aboutissant à A. Il nous a paru intéressant de distinguer règles de définition et conditions vu leur rôle différencié par rapport au calcul de la solution.

Remarquons que la notion de dépendance est transitive. En effet, si A dépend de B, et si B dépend de C, alors A dépend aussi de C.

Le graphe de la figure 2.6. représente les dépendances directes et indirectes du modèle de l'étude de cas n°1.

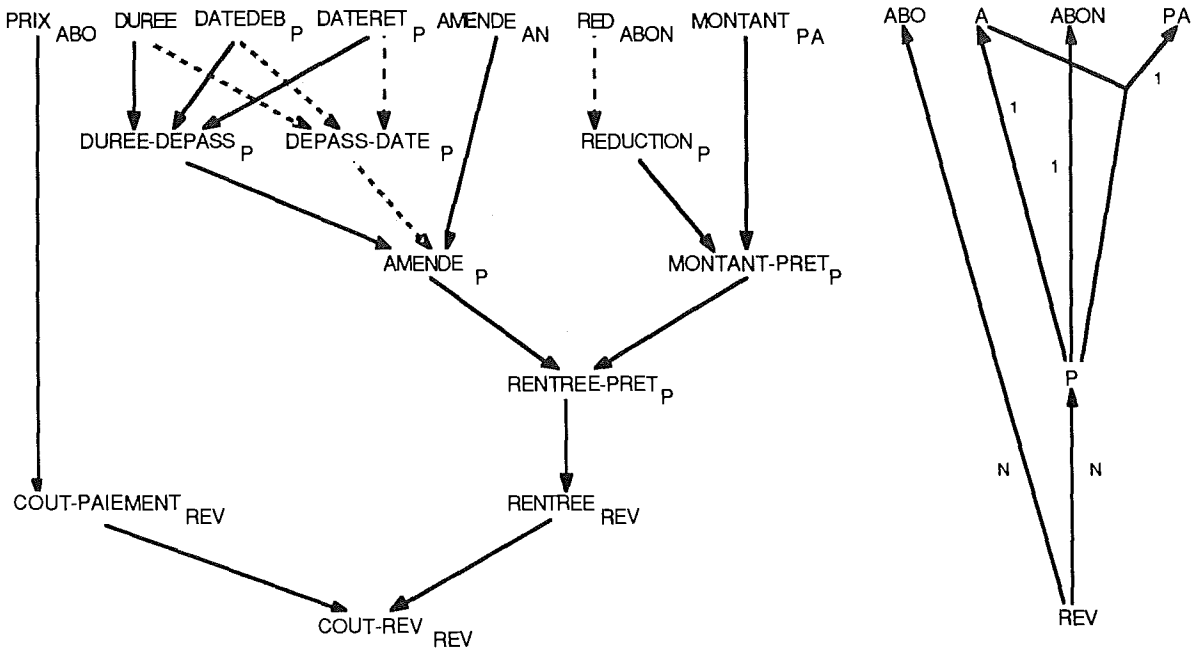


Figure 2.6. : Graphe de dépendances des grandeurs (étude de cas n°1)

Ce graphe de dépendances constitue un outil primordial pour la vérification de cohérence (II.3.). La plupart des mécanismes de contrôle de la **cohérence globale** (au niveau de l'ensemble des règles (II.3.3.)) sont basés sur des recherches dans le graphe.

Exemple : lors de la vérification de cohérence globale, nous tentons derepérer les règles qui n'apportent aucune information au modèle (règles subsumantes (II.3.3.3.)) Ce sont des règles qui définissent des grandeurs internes qui n'interviennent pas dans la détermination du résultat. On repère ces grandeurs internes à l'aide du graphe : il s'agit de celles qui n'ont aucun résultat parmi leurs descendants.

Nous ferons également appel au graphe dans la partie consacrée à la démarche de conception (partie III). En effet, lors de la conception, nous conseillerons à l'utilisateur de construire la dépendance avant de définir la relation mathématique. Le graphe, par son aspect ergonomique, permettra à l'utilisateur de mieux visualiser la solution qu'il conçoit.

Comparaison entre les deux graphes de dépendances

Les variables entité, en plus de leur rôle de "notation", permettent d'extraire les données de la base de données. Toutefois, elles n'expriment qu'auxiliairement le calcul du résultat. Si on met en parallèle les deux graphes (figure 2.6.), on perçoit une même "découpe en niveau". En effet, dans le graphe des grandeurs, en commençant par le résultat et en remontant vers les données (niveau le plus haut), nous rencontrons d'abord les grandeurs dimensionnées par REV, puis les

grandeurs dimensionnées par P et enfin celles dimensionnées par ABO, A, ABON et PA. Ces trois niveaux se retrouvent dans le GDV en commençant cette fois par le sommet du graphe (REV).

En supposant que l'on a conçu le modèle d'une manière top-down (des résultats vers les données), les deux graphes ont donc suivi une construction parallèle. Cette remarque est à la base d'une partie de la démarche de conception où l'on définit dimensions et grandeurs de manière imbriquée.

II.2.3. Règles particulières

Les concepts de règle et de grandeur sont indissociables dans le modèle de spécification. Pour étudier l'un, il faut analyser l'autre et inversement. Dans cette optique, nous avons privilégié la notion de grandeur pour établir un plan d'analyse des concepts. Nous avons proposé un définition des divers types de grandeurs auquel nous avons adjoint une description de leur règle de définition. Cependant, dans cette démarche, nous avons laissé de côté deux types de règles particulières de définition de grandeurs : les règles de récurrence et les règles récursives.

Ces deux types de règles ajoutent chacune une dimension supplémentaire au modèle de spécification. Détaillons ces deux formes d'expression, en débutant par les règles de récurrence.

II.2.3.1. Règles de récurrence

Une règle de récurrence concerne une grandeur dimensionnée qui se retrouve à la fois en partie gauche et en partie droite de cette règle. Elle exprime que la valeur de la grandeur dépend d'une ou de plusieurs de ses valeurs **précédentes** ou **suivantes** .

Pour faire dépendre une valeur particulière d'une grandeur dimensionnée d'une autre valeur de celle-ci (précédente ou suivante), il faut être capable de désigner un élément particulier dans la suite de valeurs de la grandeur. Nous avons déjà mis en place un mécanisme pour effectuer cette opération : la notion de **paramètre externe** lié à une dimension (II.2.1.4.). La spécification de règles de récurrence fera donc un usage intensif du concept de domaine paramètré.

A présent que nous connaissons l'objectif des règles de récurrence et un mécanisme pour le réaliser, nous pouvons détailler les différentes formes possibles de règles de récurrence. Cependant, avant toute définition formelle, nous préférons illustrer ce concept par un exemple simple.

Considérons le modèle de la figure 2.7. Il s'agit de calculer le stock d'un entrepôt sur un certain nombre de périodes. De manière classique, la valeur d'un stock à la fin d'une période est calculée à partir des achats, des ventes de la période et, également, **à partir du stock de la période précédente**. La règle de définition du stock fait appel à une valeur précédente de stock : on a bien le schéma d'une règle récurrente.

Données :
T : entier ; périodes sur lesquelles on va calculer le stock
CONSOMMATION $_T$: réel ; quantité du stock consommée durant la période T
ACHATS $_T$: réel ; achats de matières premières au début de la période T
Résultat :
STOCK $_T$: réel ; stock disponible de matières premières au début de la période T
Grandeurs internes :
T(i) : entier ; ième période
i : entier ; paramètre externe de T

Règles :

$T = T_1, \ldots, T_n$

$STOCK_{T(1)} = ACHATS_{T(1)} - CONSOMMATION_{T(1)}$

$STOCK_{T(i)} = STOCK_{T(i-1)} + ACHATS_{T(i)} - CONSOMMATION_{T(i)}$, $(i = 2...n)$

Figure 2.7. : Modèle du calcul des stocks

D'après la définition, trois formes générales de règles de récurrence sont envisageables :

- celles où interviennent une ou plusieurs valeurs précédentes de la grandeur expliquée. On peut formaliser cette situation de la manière suivante :
  $Q_{T(1)} = Expr1, \ldots, Q_{T(k)} = Exprk$ (initialisation)
  $Q_{T(i)} = h(Q_{T(i-k)})$, $(i = k+1..n)$ (règle de récurrence)
  avec Q une grandeur dimensionnée par T, (Expr1,...,Exprk) des expressions ne contenant pas Q_T et h une expression où apparaît obligatoirement $Q_{T(i-k)}$. Le pas[1] utilisé est k. Par conséquent, les k premières valeurs de Q_T sont expliquées par des règles non récurrentes.
- celles où interviennent une ou plusieurs valeurs suivantes de la grandeur expliquée. Elles peuvent s'écrire comme suit :
  $Q_{T(n-k)} = Exprn-k, \ldots, Q_{T(n)} = Exprn$ (initialisation)
  $Q_{T(i)} = h(Q_{T(i+k)})$, $(i = 1..n-k)$ (règle de récurrence)
  A partir des k dernières valeurs de Q_T, on définit les (n-k) premières.
- dans des cas particuliers, on pourrait même envisager des règles de récurrence où interviennent des valeurs précédentes et suivantes de la grandeur expliquée :
  $Q_{T(1)} = Expr1,\ldots,Q_{T(k)} = Exprk, Q_{T(n-p)} = Exprn-p,\ldots,Q_{T(n)} = Exprn$
  $Q_{T(i)} = h(Q_{T(i-k)}, Q_{T(i+p)})$, $i = (k+1..n-p)$
  Le pas pour les valeurs précédentes est k, celui pour les suivantes est p.

Trois remarques se dégagent de ces définitions.

Premièrement, pour être le plus général possible, nous avons utilisé des variables (k et p) désignant le pas, bien que, dans la plupart des cas, le pas sera égal à un.

Deuxièmement, nous avons dit qu'une grandeur pouvait être expliquée par une ou plusieurs de ses valeurs précédentes ou suivantes. Dans le cas où il s'agit de **plusieurs** valeurs, l'initialisation se fait pour le plus grand pas utilisé (ainsi il englobera automatiquement les autres). L'exemple du calcul de la suite de Fibonacci en est un bonne illustration.

Exemple : $F_{T(1)} = 0$

$F_{T(2)} = 1$

$F_{T(i)} = F_{T(i-1)} + F_{T(i-2)}$, $(i = 3..n)$

La grandeur F_T est expliquée à partir de plusieurs de ses valeurs précédentes. Les pas sont respectivement égal à 1 et à 2. L'initialisation se fait pour le plus grand ; elle comporte donc deux règles.

Finalement, il ne faut pas oublier d'accompagner la règle de récurrence du domaine de valeurs dans lequel voyage le paramètre externe. La portée est fixée de telle sorte que la grandeur expliquée par la règle est définie pour toutes les valeurs possibles du paramètre (exceptées celles définies par l'initialisation).

---

[1] Le pas représente la différence entre le rang de la dimension de la grandeur expliquée et le rang de la dimension de la grandeur causant la récurrence.

Examinons quelques exemples :

- (Figure 2.7. : modèle de calcul des stocks)
  $STOCK_{T(i)} = STOCK_{T(i-1)} + ACHAT_{T(i)} - CONSOMMATION_{T(i)}$, i = (2,...,n)
  La grandeur $STOCK_T$ est expliquée à partir de sa valeur précédente.
- si, dans l'exemple du modèle de calcul des stocks, on décidait de trouver les stocks à partir de sa valeur suivante, alors on commencerait par le stock de la période n. On obtiendrait donc les règles:
  $STOCK_{T(n)} = ACHAT_{T(n)} - CONSOMMATION_{T(n)}$
  $STOCK_{T(i)} = STOCK_{T(i+1)} + ACHAT_{T(i)} - CONSOMMATION_{T(i)}$, (i = 1..n-1)

Concept plus avancé : récurrence emboîtée

Qu'entend-t-on par "récurrence emboîtée" ? Il s'agit d'une règle de récurrence dans laquelle apparaît une grandeur dimensionnée par une dimension autre que celle de récurrence mais dont le paramètre externe est identique.

Exemple : (étude de cas n°3)

Transformons la règle de récurrence pour calculer le prix initial à l'escale et non plus la quantité initiale.

$PRIX\text{-}INIT_{T(1)} = QRES\text{-}INIT * TARIF\text{-}CARB_{E(1)}$

$PRIX\text{-}INIT_{T(i)} = (QINIT_{T(i-1)} + QEMP_{T(i-1)} - QCONS_{T(i-1)}) * TARIF\text{-}CARB_{E(i-1)}$, (i = 2...n)

Le montant initial de la quantité restant dans le réservoir au tronçon T(i) est égale à la quantité résiduelle fois le prix du carburant à l'escale E(i-1).

Pour mettre en oeuvre ce genre de règle, une condition primordiale doit être vérifiée. Le cardinal du domaine de valeurs de la dimension (non de récurrence) doit être **égal** au cardinal de celui de la dimension de récurrence. Plus clairement, dans l'exemple, T et E ont le même cardinal puisque E est paramètré selon T avec un chemin de poids 1. A une occurrence de TRONCON correspond une occurrence d'ESCALE-VOL.

Nous voyons donc que, en ce qui concerne les dimensions variables entité, l'intuition voudrait que celles-ci, en plus d'avoir une cardinalité identique, soient reliées par un chemin de poids 1. Nous sommes dans un cas de propagation du paramètre entre variables entité.

Pour ce qui est des variables simples, aucune relation concrète n'ayant été clairement définie entre elles, la contrainte de **cardinalité** est une condition nécessaire et suffisante pour la cohérence.

Pour conclure, précisons qu'il s'agit d'un concept difficile à saisir et qu'il présente finalement peu d'intérêt pour l'utilisateur.

II.2.3.2. Règles récursives

Une forme d'expression couramment utilisée est celle des règles récursives. Une règle récursive exprime qu'une grandeur dépend d'elle-même directement ou non.

Exemples : - Une entreprise vend un type d'appareils et dépense un budget non négligeable pour en faire la publicité. Il est clair que les ventes dépendent de la publicité (plus elle sera importante plus les ventes augmenteront), de même que la publicité est influencée par les ventes. Le modèle pourrait se schématiser de la sorte :

VENTES = f(PUB, ...)
PUB = g(VENTES, ...)

- (étude de cas n°2)
PUB $_{REG}$ et NOUVQUOTA $_{REP}$ sont les deux résultats calculés par le modèle. Il s'agit d'un modèle récursif car PUB $_{REG}$ est défini à partir de NOUVQUOTA $_{REP}$ et NOUVQUOTA $_{REP}$ est expliqué à partir de FACTEURPUB $_{REG}$ dont la règle de définition dépend de PUB $_{REG}$.

Notre modèle de spécification accepte les modèles récursifs qui consistent à rechercher (si elles existent) les valeurs de PUB et de VENTES qui valident les règles du modèle et qu'on appelle **solutions du modèle**[1].

Les règles récursives peuvent être développées sous la forme de règles récurrentes. Soit le modèle,

$$A = f(B)$$
$$B = g(A)$$

il est développé comme suit :

$$A_1 = f(B_1)$$
$$B_2 = g(A_1)$$
$$A_2 = f(B_2)$$
$$B_3 = g(A_2)$$
$$\ldots$$
$$A_{n-1} = f(B_{n-1})$$
$$B_n = g(A_{n-1})$$
$$A_n = f(B_n)$$

Soit plus généralement,

$$B_i = g(A_{i-1})$$
$$A_i = f(B_i)$$

L'indice n peut être soit fixé à l'avance, soit régi par une condition de convergence représentant un critère d'arrêt. Ce critère s'exprime sous la forme : $|A_i - A_{i-1}| < E$ avec E relativement petit (on aurait pu prendre indifféremment B). Cela signifie qu'on s'arrête lorsque le A calculé (A_i) est fort proche du A trouvé précédemment (A_{i-1}).

La formulation semble récursive. Toutefois, elle est traitée de manière itérative. Chaque itération consiste à calculer la valeur de A puis celle de B. Deux exemples simples nous montrent que de tels modèles peuvent converger vers des valeurs ou tout simplement diverger.

---

[1] Il existe une seconde catégorie de modèles récursifs qui décrivent des systèmes d'évolution faisant appel à des règles plus complexes dans lesquelles on fait intervenir les valeurs des grandeurs mais également leur vitesse d'évolution. Ces modèles ne seront pas envisagés dans le cadre de ce mémoire.

Exemples : - $A = B - 100$
$A = 3 * A$
Si on développe : $B_1 = 0$, $A_1 = -100$
$B_2 = -300$, $A_2 = -400$
$B_3 = -1200$, $A_3 = -1300$
...
$|A_2 - A_1| = 200$, $|A_3 - A_2| = 900$, ...
Plus on calcule le critère, plus il augmente. Il n'y a donc pas de convergence vers des valeurs de A et B.

- $A = B - 100$
$B = -A / 2$
Si on développe : $B_1 = 0$, $A_1 = -100$
$B_2 = 50$, $A_2 = -50$
$B_3 = 25$, $A_3 = -75$
$B_4 = 37,5$, $A_4 = -62,5$
$B_5 = 31,25$, $A_5 = -68,75$
$B_6 = 34,375$, $A_6 = -65,1875$
$B_7 = 32,59375$, $A_7 = -67,40625$
$B_8 = 33,703125$, $A_8 = -66,296875$
$B_9 = 33,1484375$, $A_9 = -66,8515625$
$B_{10} = 33,42578125$, $A_{10} = -66,5742187$
On voit clairement que l'on converge vers la valeur 33,33... pour B et -66,66... pour A.

Il sera possible de repérer ce genre de règles grâce aux graphes de dépendances des grandeurs du modèle. Lorsqu'il y a un cycle dans ce graphe, on a repéré un modèle récursif.

II.2.4. Les contraintes d'intégrité

Dans ce point, nous allons également aborder des règles particulières. Ces règles sont différentes de celles déjà rencontrées dans le sens où elles ne définissent pas une grandeur du modèle. Les contraintes d'intégrité sont des règles qui ont pour objectif de poser des conditions que devront respecter les divers objets du modèle. Elles ne se rattachent donc plus à une seule grandeur mais à plusieurs.

II.2.4.1. Définition

Pour introduire la notion de contraintes d'intégrité d'un modèle, nous nous proposons de repartir de notre exemple générique des clients et des commandes de la figure 2.3. Supposons que pour toute commande effectuée par le client, les frais adjoints à celle-ci doivent être inférieurs de moitié à son montant (frais déduits).
On exprimera cette contrainte de la manière suivante :

$$FRAIS_{COM} < MONTANT\text{-}COM\text{-}SF_{COM} / 2$$

$$MONTANT\text{-}COM\text{-}SF_{COM} = \sum_{LC} MONTANT\text{-}LIGNE_{LC}$$

où $MONTANT\text{-}COM\text{-}SF_{COM}$ représente le montant de la commande COM frais déduits.

Par ce court exemple, on se rend compte que les concepts proposés jusqu'à présent ne permettent pas de spécifier toutes les propriétés sémantiques de la situation réelle que l'on veut modéliser. La propriété, introduite par l'exemple, a pour effet de limiter les valeurs possibles de diverses grandeurs du modèle. Ce sont des **contraintes d'intégrité**.

> Une contrainte d'intégrité est une propriété formelle que les composants du modèle doivent vérifier à tout instant

Dans la suite de cette section, nous allons exposer deux grands types de contraintes d'intégrité : les domaines de valeurs et les relations entre grandeurs.
Notre approche des contraintes est loin d'être complète. Cette notion devrait faire l'objet d'études plus poussées. Ne sont-elles pas, au même titre que les règles de domaine et les règles du problème, un concept central du modèle ?

Notre concept de contrainte d'intégrité d'un modèle est évidemment très proche de celui de **contrainte d'intégrité de la base de données**. Comme on extrait un certain nombre de valeurs d'une base de données, les contraintes d'intégrité définies sur ces dernières sont toujours valables quand elles sont utilisées dans un modèle. On peut donc dire que les contraintes d'intégrité de la base de données font partie des contraintes d'intégrité du modèle.

II.2.4.2. Domaines de valeurs

Ces contraintes ont pour but de restreindre l'ensemble des valeurs que peut prendre une grandeur du modèle. La propriété consistera à ajouter des éléments contraignants au domaine de valeurs de la grandeur.
Exemple : (figure 2.3.)

MONTANT-LIGNE $_{LC} > 0$

Dans le modèle du calcul du montant dû par un client, la grandeur MONTANT-LIGNE $_{LC}$ doit être positive à tout moment.

II.2.4.2.1. Domaines de valeurs des grandeurs dimensionnées et non dimensionnées

On n'exprimera un domaine de valeurs pour ce type de grandeurs que si elles sont numériques. L'expression d'un tel domaine peut concerner les données, les résultats et les grandeurs internes.
Exemple : Dans le modèle de la figure 2.3., on pourrait imposer que la donnée N soit toujours supérieure à zéro. On aurait alors une règle de contrainte : $N > 0$

Remarque : que se passe-t-il si on exprime un domaine de valeurs sur un attribut de la base de données ? Dans ce cas, on exprime une contrainte sur la base de données. Le concepteur devra veiller à ce que cette règle n'entre pas en conflit avec les contraintes d'intégrité de la base. S'il n'y a pas de contradiction, cette contrainte définit une propriété que devront vérifier les attributs uniquement lors de leur utilisation dans le modèle.

II.2.4.2.2. Domaines de valeurs des dimensions

Variables simples

On pourra donner une restriction aux diverses valeurs de la variable simple si elle est du type numérique.
Exemple : AN = a1, a2, ..., aN

Soit AN les années pour lesquelles on va estimer un budget. Cependant cette estimation ne peut être effectuée que pour les années avant l'an 2000. On peut exprimer cette propriété dans une règle de contrainte : Pour tout i dans (1..N), $a_i < 2000$

De même, on pourra poser des contraintes sur tout l'ensemble de valeurs de la variable.
Exemple : taille (AN) < 20

Cette contrainte exprime le fait que le budget ne peut être évalué que pour vingt années différentes.

Variables entité

Une entité n'étant pas de type numérique, on exprimera uniquement des conditions sur l'ensemble d'entités désigné par la règle de domaine de la variable.
Exemple : On peut définir des contraintes sur les variables entité de la figure 2.3.

- taille (LC) < 10
  Le concepteur exige que pour toutes les commandes du client dont on calcule le montant, il n'y a pas plus de dix lignes.
- taille (CLI) =1. CLI désigne le client de numéro N.
  Le modèle ne s'exécutera que si CLI a une valeur.

Remarquons à nouveau, comme pour le cas des attributs, la possibilité de conflits avec les contraintes d'intégrité de la base de données.

II.2.4.3. Relations entre grandeurs

Une contrainte de ce type exprime une relation entre composants du modèle qui doit être vérifiée à tout instant. On distinguera les relations entre grandeurs dimensionnées et non dimensionnées et les relations entre dimensions.

II.2.4.3.1. Relations entre grandeurs dimensionnées et non dimensionnées

La syntaxe qui permet d'exprimer une relation entre grandeurs est analogue à celle des règles du problème. Seul l'opérateur "=" diffère et peut être remplacé par les opérateurs (<, >, ≥, ≤). Pour des raisons de simplicité, nous ne tolèrerons pas de fonction d'agrégation dans aucun des deux membres de la règle. On peut facilement pallier à cette limitation en introduisant de nouvelles grandeurs internes.
Exemple : Dans l'exemple introductif des contraintes, nous avons créé une grandeur interne supplémentaire (MONTANT-COM-SF $_{COM}$) pour éviter d'introduire une fonction d'agrégation dans la contrainte d'intégrité.

Remarquons que les relations entre grandeurs seront également soumises au contrôle de cohérence au niveau d'une règle (point II.3.2.).
Exemple : (étude de cas n°3)

$QMINT_T = QCONS_T - QINIT_T$
$QMAX_T = CAP\text{-}RESERVOIR_{APP} - QINIT_T$
$QMIN_T \leq QEMP_T$ (1)
$QMAX_T \geq QEMP_T$ (2)

Pour satisfaire notre restriction et être le plus clair possible dans l'expression des contraintes, deux nouvelles grandeurs internes ont été introduites :

- $QMIN_T$: quantité minimum de carburant à acheter à l'escale de départ du tronçon T.
- $QMAX_T$: quantité maximum de carburant à acheter à l'escale de départ du tronçon T.

La première contrainte (1) exprime que pour effectuer le tronçon T, il faut au moins emporter $QEMP_T$ .La seconde contrainte (2) exprime qu'on ne peut emporter plus de $QMAX_T$ pour effectuer le tronçon T.

II.2.4.3.2. Relations entre dimensions

Ce type de règles de contrainte met en rapport des ensembles de valeurs. Une telle contrainte possédera donc un opérateur ensembliste du type (in, not-in, ==, not==). Le type de contraintes le plus fréquent sera une inclusion entre ensembles.

Le concepteur devra à nouveau veiller à l'absence de conflits avec les contraintes de la base de données.

Exemple : reprenons l'exemple de la figure 2.3.

CLI = CLIENT(:NCLI=N)
CLI in CLIENT(CC:COMMANDE)
Le client dont on calcule le montant dû doit avoir passé au moins une commande.

II.2.5. Notion de sous-modèle et modularisation

De la même manière qu'un programme complexe est constitué d'un assemblage de procédures, un modèle complexe pourra faire appel à des **sous-modèles**, qui sont des modèles déjà définis (ou à définir). Un sous-modèle n'étant utilisé que comme composant d'un modèle, seul son aspect externe nous intéresse. Ainsi, un modèle complexe pourra appeler une table, une procédure en Pascal, un calcul de régression linéaire, un système expert, etc.
Exemple : (étude de cas n°1)

A=ANNEE(:AN=Fannée(DATEDEB $_P$)) où Fannée(DATEDEB $_P$) est évalué par une procédure en Pascal retirant l'année d'une date, son argument étant DATEDEB $_P$.

Dans ce point, nous ne ferons qu'aborder certains aspects de la notion de sous-modèle. Notre approche sera tout à fait informelle et basée essentiellement sur des exemples.

Pour mieux saisir l'intérêt de la modularisation, reprenons l'étude de cas n°2 dans laquelle nous allons introduire un sous-modèle. En effet, les trois dernières équations ressemblent très fortement au modèle de la figure 2.3. (modèle calculant le montant dû par un client dont on connaît le numéro). On va reprendre ce modèle en le transformant quelque peu pour l'adapter à la base de données de la deuxième étude de cas. Nous appelerons ce modèle CALCULMONTANT. Il se présente comme suit :

Donnée :
N : entier ; numéro de client
Résultat :
MONTANT $_{CL}$: réel ; total des achats du client CL
Grandeurs internes :
CL : CLIENT ; client dont le numéro est N
REG : REGION ; région du client CL
COM : COMMANDE ; commande du client CL
LC : LIGNE ; ligne de commande de COM
VR : VENTEREG ; prix du produit de LC et vendu dans la région REG
MONTANT-COM $_{COM}$: réel ; montant de la commande COM
MONTANT-LC $_{LC}$: réel ; montant de la ligne de commande LC
Règles :
CL = CLIENT(:NOCLI = N)
REG = REGION(cr:CL)
COM = COMMANDE(cc:CL)
LC = LIGNECOM(lc:COM)
VR = VENTEREG((vp:PRODUIT(pl:LC)) et (rv:REG))

$$\text{MONTANT}_{CL} = \sum_{COM} \text{MONTANT-COM}_{COM}$$

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LC}_{LC}$$

$$\text{MONTANT-LC}_{LC} = \text{QUANTITE}_{LC} * \textbf{PRIX}_{VR}$$

Dans l'étude de cas n° 2, on pourrait remplacer les trois dernières équations par un appel au modèle CALCULMONTANT. Il devient ainsi un sous-modèle. La règle de définition du montant dû par le client CLI devient :

$$\text{MONTANT-CLI}_{CLI} = \text{CALCULMONTANT}(\text{NOCLI}_{CLI})$$

où NOCLI_{CLI} est le numéro du client CLI.

Si on déplie cette règle, on obtient pour CLI = (c1,c2,c3,c4) :

$\text{MONTANT-CLI}_{c1} = \text{CALCULMONTANT}(\text{NOCLI}_{c1})$
$\text{MONTANT-CLI}_{c2} = \text{CALCULMONTANT}(\text{NOCLI}_{c2})$
$\text{MONTANT-CLI}_{c3} = \text{CALCULMONTANT}(\text{NOCLI}_{c3})$
$\text{MONTANT-CLI}_{c4} = \text{CALCULMONTANT}(\text{NOCLI}_{c4})$

Le sous-modèle CALCULMONTANT sera appelé autant de fois qu'il y a d'éléments dans CLI. Remarquons que le sous-modèle n'a comme paramètres que les données définies dans le modèle appelé.

Comme dans la programmation traditionnelle, la modularisation d'un modèle conduit à une simplification de sa construction, ainsi qu'à une plus grande stabilité du résultat, lorsque des modifications doivent intervenir.

La forme générale de l'appel d'un sous-modèle est :

$$\text{GRANDEUR} = \text{S-MOD}\ (d_1\ ,\ \ldots\ ,\ d_n)$$

où S-MOD représente le nom du modèle appelé, $(d_1,\ldots,d_n)$ la liste des paramètres qui sont toutes les valeurs pour les données du modèle appelé.

Un sous-modèle appelé ne fournit pas toujours un seul résultat, mais parfois un vecteur de résultats. Comment distinguer, dans ce contexte, le résultat désiré lors de l'appel ? Nous utiliserons la notation suivante :

(Res1, ..., Resm) = S-MOD (d1, ...,dn)

Pour obtenir le ième résultat :

GRANDEUR = (i,NOM (d1, ..., dn))

Pour employer un sous-modèle dans une règle, il y a un certain nombre de contraintes à respecter.

D'abord, le modèle appelé et le modèle appelant doivent être définis sur le **même schéma de base de données**. Cette contrainte est merveilleusement illustrée par notre exemple des clients et des commandes. Nous avons dû modifier les règles de domaine de la figure 2.3. (en introduisant les deux nouvelles variables entité VR et REG pour extraire le prix des produits) afin de les rendre compatibles avec le schéma des données du modèle appelant.

Lors de l'appel, il y a également des contraintes à respecter surtout au niveau de l'emploi des dimensions. Dans le cadre de ce mémoire, nous ne rentrerons pas dans une analyse détaillée de ces **contraintes d'appel**. Toutefois, le sujet se doit d'être étudié en profondeur dans des développements futurs.

Prenons un exemple afin de mettre en évidence certaines difficultés d'utilisation des sous-modèles. Considérons le modèle de la deuxième étude de cas. Celui-ci calcule, entre autres, le budget publicitaire d'une région de nom donné. Supposons qu'un utilisateur veuille appeler ce modèle dans une de ses règles du problème. Dans le modèle appelant, l'utilisateur désire calculer la publicité (PUBLICITE) pour un ensemble d'entités de REGION désigné par la variable entité R. La règle de définition de $PUBLICITE_R$ se présente comme suit :

$$PUBLICITE_R = (1,ETUDECAS2(NOM_R, AN, NOUVQUOTA_{PR})$$

Pour obtenir le premier résultat fourni par le modèle ETUDECAS2, nous avons employé la notation définie précédemment. Les arguments de l'appel sont les données du modèle.

PUBLICITE est dimensionnée par R qui désigne un sous-ensemble de régions dont l'utilisateur veut estimer le budget publicitaire. Les variables entité R et REG (dimension de la grandeur résultat PUB dans l'étude de cas n°2) sont définies sur le **même type d'entités** REGION. Cette contrainte est obligatoire, car le sous-modèle ETUDCAS2 calcule un attribut virtuel de REGION. On remarque que le cardinal du domaine de valeurs de REG est égal à un, tandis que le domaine de R est un ensemble de plusieurs entités. Pour faire correspondre ces deux domaines, le modèle ETUDECAS2 sera appelé autant de fois qu'il y a de régions dans le domaine de R (pour que la grandeur PUBLICITE représente une suite de valeurs).

Pour calculer le modèle ETUDECAS2, il faut lui donner un nom de région (NOMR). Toutefois, comme il y aura plusieurs appels, il convient de passer le bon nom. C'est pourquoi on utilisera l'attribut réel du type d'entités REGION, NOM dimensionné par R.

L'année de début de l'analyse doit également être passée en paramètre. Quelque soit la région, cette donnée sera toujours la même. AN reste une grandeur simple.

Les nouveaux quota des produits sont aussi des données du modèle ETUDECAS2. Dans celui-ci, la grandeur QUOTANOUVPRO est dimensionnée par la variable entité PRO, une grandeur interne. Pour que l'appel soit correct, la variable entité PR, du modèle appelant, doit avoir le même domaine que la dimension PRO. Cette dernière dimension est, par sa règle de domaine, implicitement paramètrée selon REG. Lors de l'appel, il ne faut donc pas dimensionner $NOUVQUOTA_{PR}$ par R.

II.2.6. Valeur nulle et propagation de l'erreur

II.2.6.1. Valeur nulle

Il peut arriver dans une base de données qu'on ait enregistré, au sujet d'un fait du réel perçu, un ensemble incomplet d'informations (certaines sont connues et d'autres restent à déterminer). On admet donc que certains attributs puissent prendre une valeur particulière dite inconnue ou appelée, dans certains gestionnaires de bases de données, ***valeur nulle*** .

Mais, avant d'avoir une valeur connue ou inconnue, une information existe ou n'existe pas, dénotant l'existence ou l'absence de celle-ci dans le monde réel. On parlera d'***information facultative*** [HAINAUT,88].

A l'heure actuelle, il existe un certain nombre de politiques pour gérer le traitement des informations inconnues et facultatives. Les problèmes entraînés par ces deux types de valeurs sont assez comparables. Toutefois, la situation est différente : dans le cas de l'information inconnue, elle donne un statut temporaire à l'entité qui la contient ; dans le cas de l'information facultative, le statut de l'entité peut être permanent car la valeur peut correspondre non pas à une perception incomplète du réel mais à une situation perçue très clairement.

Exemples : - (information facultative)
Un type d'entités du schéma de la base de données PERSONNE contient les attributs (NOM, N°IMMATRICULATION, NOM-CONJOINT). L'attribut NOM-CONJOINT est facultatif.
- (information inconnue)
Dans une occurrence du type d'entités CLIENT, il se peut qu'un nom de client (NOM) soit absent (oubli d'encodage par exemple).

Finalement, notons que les langages relationnels ne distinguent pas ces deux types de valeurs. Pour plus de facilités, nous les assimilerons donc tous les deux aux valeurs nulles.

II.2.6.2. Propagation de l'erreur

Dans le cadre de ce mémoire, les problèmes engendrés par les informations incomplètes et facultatives, pour des systèmes de gestion de bases de données, ne nous intéressent pas. D'ailleurs, il existe une littérature abondante sur le sujet. Par contre, la gestion de telles informations dans une base de règles va attirer notre attention.

Pour mieux saisir le problème, replaçons-nous dans le modèle de calcul du montant dû par le client dont on donne le numéro (figure 2.3.). Supposons que le prix d'un produit ne soit pas connu. Il s'agit bien d'une information incomplète. Quelle stratégie va-t-on adopter pour gérer cette situation ?

Plusieurs stratégies sont envisageables :

a) une solution radicale serait d'empêcher l'exécution du modèle tant que la base de données n'est pas complète ou, du moins, tant qu'il existe encore des valeurs d'attribut inconnues ou inexistantes (c'est une contrainte supplémentaire pour l'utilisateur).

b) une solution très facile serait de garder la valeur inconnue et de la propager à travers tout le modèle, le modèle étant alors inhibé (c'est la solution choisie dans LOTUS 123).

c) une solution plus mitigée serait d'essayer de retrouver l'information manquante à partir des autres (nous expliquerons dans quelles conditions dans le point suivant).

La deuxième stratégie est la plus employée à l'heure actuelle et sera également utilisée dans notre modèle de spécification.

II.2.6.3. Prolongement possible

Un prolongement possible serait de développer une stratégie essayant de pallier le manque d'informations fournies par la base de données. Cette solution apparaît comme la moins contraignante. Ne serait-il pas possible dans certains cas particuliers de l'appliquer ? Nous avons essayé de distinguer plusieurs cas (définis par des hypothèses) où une solution constructive pourrait être envisagée :

Premier cas

Hypothèses : - un attribut d'une entité a une valeur nulle (information inconnue).
- en remontant de règle en règle, on aboutit à une fonction d'agrégation où apparaît une grandeur calculée directement ou indirectement à partir de cet attribut.
- la dimension de la fonction et celle de la grandeur sont identiques ou bien il existe un chemin de poids 1 entre ces deux dimensions (cfr graphe de dépendances entre variables entité, point II.2.1.3.).

Solution : on peut pallier le manque d'information en faisant une moyenne sur toutes les valeurs de l'attribut (en additionnant toutes les valeurs prises par cette attribut dans la base de données et en divisant par le nombre de valeurs). Toutefois, cette solution n'a du sens que si l'écart entre les valeurs (minimale et maximale) prises par l'attribut est petit. L'utilisateur a évidemment la possibilité de définir ce qu'il entend par "petit".

Exemple : dans la figure 2.3. (point II.1.3.4.), le prix d'un produit n'est pas présent dans la base de données. Si on remonte à la règle précédente, on voit que la grandeur MONTANT-LIGNE $_{LC}$ intervient dans la somme :

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LIGNE}_{LC} + \text{FRAIS}_{COM} .$$

Cette grandeur est calculée à partir du prix du produit (PRIX $_{PRO}$). La dimension de la fonction et celle de la grandeur sont identiques. Nous sommes bien dans le premier cas. Il reste à voir si l'écart entre les valeurs extrêmes de l'attribut PRIX de l'entité PRODUIT est assez petit. Si les produits se ressemblent beaucoup et que la variation de prix entre chaque gamme de produits est minime, on peut appliquer la stratégie. Prenons l'exemple de clous, la variation de prix entre catégories de clous est minime. On pourra donc employer la stratégie.

Deuxième cas

Hypothèses : - un attribut quelconque a une valeur nulle pour une entité.
- il n'existe pas de fonction d'agrégation où apparaît soit cet attribut soit une grandeur définie directement ou indirectement par cet attribut.
- si une telle fonction d'agrégation existe, alors la dimension de la fonction et celle de la grandeur sont différentes et elles ne sont pas reliées par un chemin de poids 1.

Solution : Nous avons des valeurs indéfinies pour certaines occurrences de la variable entité dimensionnant l'attribut considéré (c'est la solution de la propagation de l'erreur).

Exemple : (point II.1.3.4., figure 2.3.)
Supposons qu'on ne connaisse pas la valeur de ETAT $_{CLI}$. Il est clair que cet attribut n'intervient dans aucune fonction d'agrégation (directement ou indirectement). Le résultat MONTANT-DU $_{CLI}$ est dimensionné par CLI qui est indéfinie pour une occurrence. CLI ne représentant qu'une seule occurrence (celle du client de numéro N), le résultat est indéfini. Mais, si CLI avait représenté tous les clients (CLIENT), alors MONTANT-DU $_{CLI}$ serait indéfini pour une seule de ses occurrences (celle représentée par l'entité dont la valeur de l'attribut ETAT est nulle).

Troisième cas

Il ne reste plus qu'à traiter les possibilités écartées dans le premier cas parce que la différence entre les valeurs extrêmes trouvées pour un attribut était significative. Dans cette situation, on peut éventuellement envisager d'avertir l'utilisateur du danger de calculer des réponses erronées et de lui demander s'il désire continuer ou abandonner l'exécution du modèle.

Exemple : Reprenons l'exemple du premier cas, mais, cette fois-ci, avec des produits très différents et des prix incomparables. Il ne faut pas appliquer la stratégie car elle risque de fausser le problème. Supposons que le prix minimal est égal à 100 et le prix maximal à 2000. Dans ce contexte, le prix inconnu sera remplacé par 1050. Si, dans la réalité, il est égal à 1900, on a une sous-estimation du montant dû par le client qui serait d'autant plus grave si celui-ci avait commandé beaucoup de ce produit.

Dans ce point, nous avons essayé de montrer une voie de recherche qui nous semblait intéressante. Quant à savoir si elle l'est vraiment, seule une étude approfondie pourrait nous répondre.

II.2.7. Schéma Entité/Association des concepts du modèle

Une bonne façon de visualiser les différents concepts du modèle est de les représenter sous forme d'un schéma Entité/Association. C'est pourquoi nous proposons deux sous-schémas, l'un se rapportant aux règles et l'autre aux grandeurs, qui permettent de structurer les liens entre les concepts qui ont été proposés dans toute cette partie.

De plus, ce schéma pourra servir à la construction d'une base de données pour une future application de gestion de modèles (partie IV).

Tout au long de la définition des concepts du modèle, les deux notions prépondérantes ont été les concepts de grandeur et de règle. Cette position centrale apparaît clairement dans la construction du schéma Entité/Association. Pour des raisons de présentation, celui-ci a été divisé en deux sous-schémas : le premier reprend l'essentiel des concepts proposés en détaillant celui de règle, le second décrit la notion de grandeur.

II.2.7.1. Sous-schéma 1 : Les règles du modèle

Ce premier schéma structure les concepts de règle, de grandeur et de modèle. La notion de grandeur n'est pas entièrement développée dans ce premier schéma.

De manière informelle, nous pouvons repérer les principales notions représentées : - les règles de définition de grandeur,
- les règles à définitions multiples,
- les contraintes d'intégrité,
- la portée d'une règle (à l'aide d'un paramètre).

Donnons une description courte mais précise de l'ensemble des entités et associations présentes dans ce premier sous-schéma.

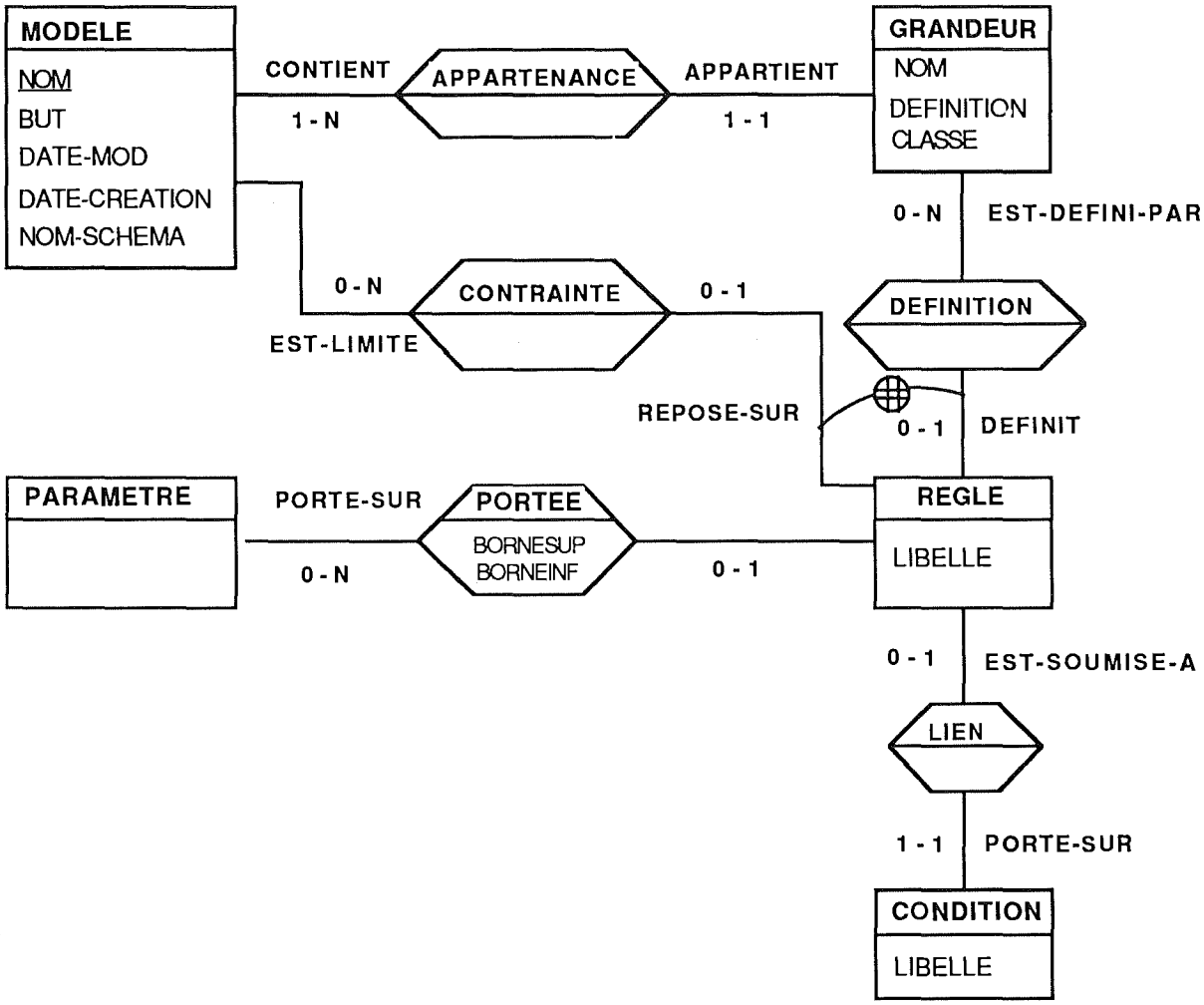
Une entité MODELE représente un modèle. Elle est caractérisée par son nom (NOM) qui l'identifie, un objectif (BUT), une date de création (DATE-CREATION), une date de dernière mise à jour (DATE-MOD) et, éventuellement, le nom du schéma de base de données auquel elle est reliée (NOM-SCHEMA). Un modèle est également caractérisé par les grandeurs qu'il contient (GRANDEUR via *appartenance*) et, éventuellement, des règles de contraintes (REGLE via *contrainte*).

Une entité GRANDEUR représente une grandeur du modèle. Elle est identifiée par un nom (NOM) et un modèle (MODELE) dans le contexte de l'association *appartenance*. Elle est caractérisée par une définition en français (DEFINITION) et une classe de grandeurs (CLASSE). L'attribut CLASSE peut prendre trois valeurs différentes: donnée, résultat ou grandeur interne. Une entité grandeur peut être définie par une ou plusieurs règles (REGLE via *definition*).

Une entité REGLE représente une règle simple du modèle. Elle est soit une règle de contrainte associée à un modèle (MODELE via *contrainte*), soit une règle de définition associée à une grandeur (GRANDEUR via *definition*). Elle est identifiée par son libellé (LIBELLE) et, soit par un modèle (MODELE via *contrainte*) si c'est une règle de contrainte, soit par une grandeur (GRANDEUR via *definition*) si c'est une règle définissant une grandeur. Elle peut être soumise à une condition (CONDITION via *lien*). La portée de l'entité REGLE peut également être définie par un paramètre externe (REGLE via *portee*).

Une entité CONDITION représente une condition liée à une règle de définitions multiples (REGLE via *lien*). Elle est caractérisée par un libellé (LIBELLE). Une entité CONDITION est identifiée par son attribut LIBELLE et par le rôle EST-SOUMISE-A joué par une entité REGLE.

Le type d'entités PARAMETRE est décrit dans le sous-schéma suivant.



ID(GRANDEUR) = (MODELE,NOM)

ID(REGLE) = (MODELE,LIBELLE) si c'est une contrainte d'intégrité du modèle
= (GRANDEUR,LIBELLE) si c'est une règle qui définit l'entité GRANDEUR

ID(CONDITION) = (LIBELLE,REGLE)

Nous pouvons exprimer les **contraintes d'intégrité** suivantes :

- Les rôles REPOSE-SUR et DEFINIT du type d'entités REGLE sont mutuellement exclusifs.
- Si une entité REGLE ne joue pas le rôle DEFINIT, alors elle joue obligatoirement le rôle REPOSE-SUR et inversement.
- Si une grandeur est une donnée, alors elle n'est définie par aucune règle. Donc, si une entité GRANDEUR a son attribut CLASSE égal à "donnée", elle ne peut jouer le rôle EST-DEFINI-PAR.

- Si une grandeur est définie par plusieurs règles, alors sur chacune des règles qui la définit, repose une condition. Donc, si une entité GRANDEUR joue le rôle EST-DEFINI-PAR plus d'une fois, chacune des entités REGLE qui lui sont associées via *definition* joue le rôle EST-SOUMISE-A.

II.2.7.2. Sous-schéma 2 : Les grandeurs du modèle

Ce second schéma décrit uniquement la notion de grandeur.

De manière informelle, nous pouvons repérer les différents types de grandeurs :
- les grandeurs non dimensionnées,
- les grandeurs dimensionnées,
- les paramètres externes liés à une dimension,
- les variables entité,
- les variables simples.

Nous avons déjà décrit le type d'entités GRANDEUR dans le sous-schéma précédent, mais uniquement dans ses rôles vis-à-vis d'un modèle et d'une règle. Ce type d'entités est en fait une **généralisation** des types GRANDEUR DIMENSIONNEE, GRANDEUR NON DIMENSIONNEE, PARAMETRE et DIMENSION.

Une entité GRANDEUR NON DIMENSIONNEE représente une grandeur simple du modèle. Son type est un sous-type de GRANDEUR. Elle est caractérisée par un type (TYPE = entier, booléen...)) et, éventuellement, une unité de mesure (UNITE).

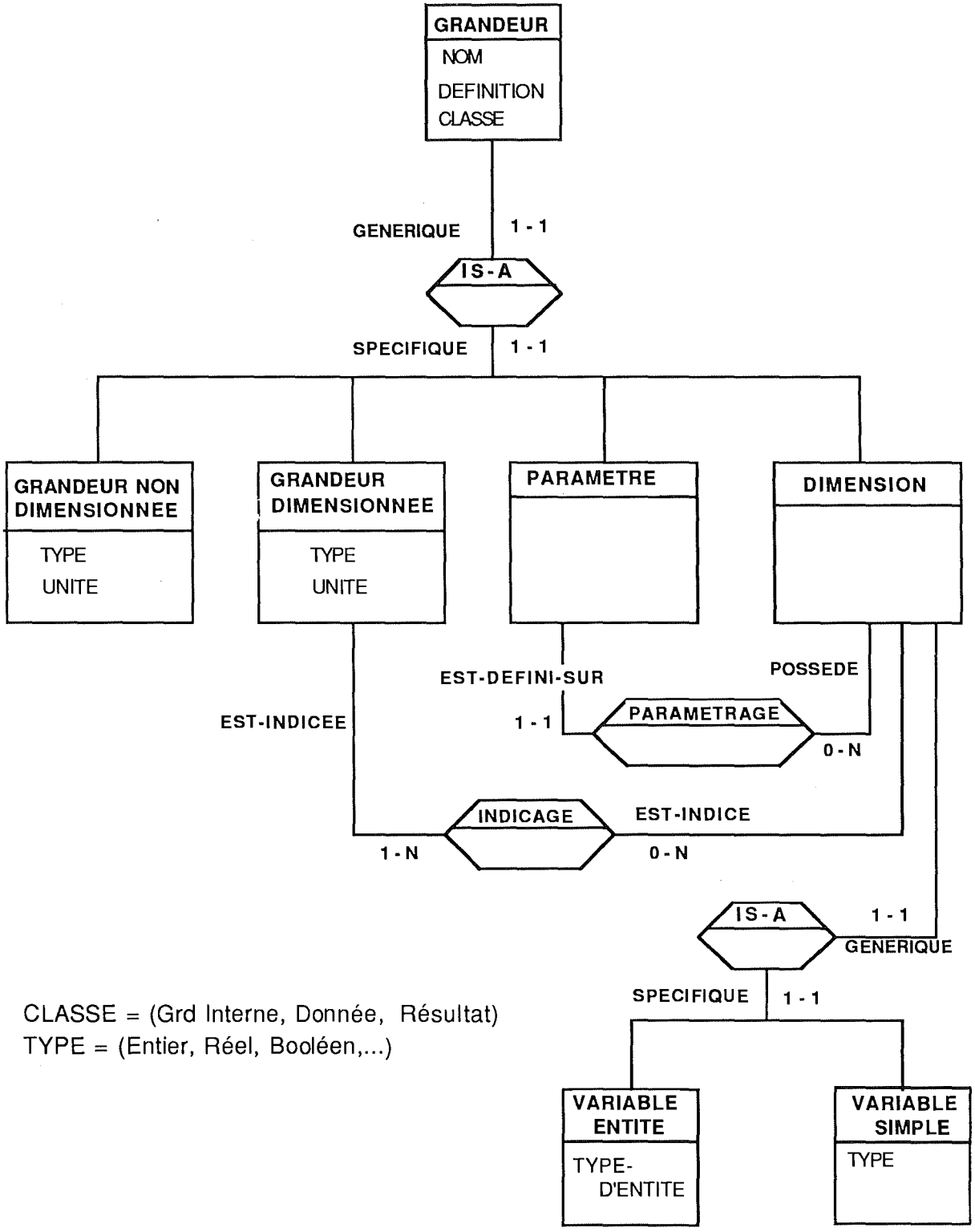
Une entité GRANDEUR DIMENSIONNEE représente une grandeur dimensionnée du modèle et son type est un sous-type de GRANDEUR. Elle est caractérisée par un type (TYPE = entier, booléen...)) et, éventuellement, une unité de mesure (UNITE). Elle possède un ou plusieurs indices (DIMENSION via *indiçage*).

Une entité DIMENSION est une variable dimension. Son type est un sous-type de GRANDEUR qui en définit les propriétés stables. Elle peut dimensionner plusieurs grandeurs (GRANDEUR DIMENSIONNEE via *indiçage*). On peut également adjoindre à une entité DIMENSION un paramètre externe (PARAMETRE via *parametrage*).

Une entité DIMENSION est une **généralisation** des types d'entités VARIABLE ENTITE et VARIABLE SIMPLE.

Une entité VARIABLE ENTITE est une variable entité du modèle. Elle est un sous-type de DIMENSION et est caractérisée par le type d'entités sur lequel elle est définie (TYPE-ENT).

Une entité VARIABLE SIMPLE est une variable simple du modèle. Son type est un sous-type de DIMENSION et elle est caractérisée par son type (TYPE).



CLASSE = (Grd Interne, Donnée, Résultat)
TYPE = (Entier, Réel, Booléen,...)

Une entité PARAMETRE représente le paramètre externe d'une seule dimension (DIMENSION via *parametrage*). Elle peut également définir la portée d'une ou plusieurs règles (REGLE via *portee*). Chaque association *portee* entre une entité REGLE et une entité PARAMETRE est caractérisée par une borne supérieure (BORNESUP) et une borne inférieure (BORNEINF).

Nous pouvons exprimer les **contraintes d'intégrité** suivantes :

- Une entité DIMENSION qui ne joue pas le rôle INDICE a son attribut CLASSE (qu'elle hérite de GRANDEUR) égal à "résultat".
- Une entité PARAMETRE a toujours son attribut CLASSE égal à "grandeur interne".
- L'attribut TYPE-ENT d'une entité VARIABLE ENTITE doit désigner un type d'entités qui existe dans le schéma de la base de données dont le nom est NOM-SCHEMA (attribut de MODELE).

II.2.8. Relations base de données/modèle

En guise de conclusion de l'analyse des concepts, nous nous proposons de prendre du recul et d'analyser les liens étroits qui existent entre la base de données et un modèle qui l'utilise pour puiser des informations.

Il ne s'agit plus d'aborder "syntaxiquement" les relations entre l'un et l'autre mais plutôt de s'interroger sur leur statut mutuel. En inspectant les rôles possibles d'un modèle par rapport à une base de données, nous aboutirons à des utilisations concrètes de notre modèle de spécification de base de connaissances. Nous constaterons également l'analogie entre **bases de données déductives** ([GALLAIRE,84], [DATE,90]) et un système qui intégrerait bases de données classiques et notre modèle de spécification.

Nous avons choisi, pour cette section, une présentation informelle qui s'adapte bien au thème. En fait, nous proposons une progression historique de nos idées et observations sur le sujet.

II.2.8.1. Attributs virtuels

Les grandeurs résultats ou grandeurs internes dimensionnées par une variable entité peuvent être considérées comme des attributs virtuels d'un sous-ensemble du type d'entités initial (référencé par la variable). Ce sous-ensemble d'entités est défini par une règle de domaine. Dans certains cas, il s'agira de tout l'ensemble d'entités du type de la variable entité[1].

Exemple : - *Attribut virtuel d'uns sous-ensemble d'entités d'un type*

Supposons la base de données représentée par le schéma Entité/Association suivant :

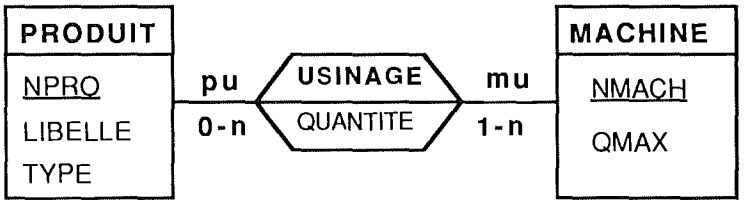


Figure 2.8. - Schéma Entité/Association.

TYPE = (fini, semi-fini, matière première)
C.I. : seuls les produits semi-finis et les matières premières peuvent jouer le rôle "pu".
Si, dans la base de règles, on a une grandeur dimensionnée:
TOTALTRANSF $_{PRO}$: entier ; total des quantités de PRO mises à la disposition des machines pour transformation
avec PRO : PRODUIT ; produit semi-fini ou matière première
Cette grandeur n'est un attribut virtuel, attaché à la base de données, que pour un sous-ensemble des éléments de PRODUIT. Ce sous-ensemble est défini par PRO = PRODUIT((:TYPE="semi-fini") ou (:TYPE="matière première")).

---

[1] Pour que chaque grandeur dimensionnée par une variable entité soit vue comme un attribut pouvant être rattaché à toutes les entités d'un type, on utilisera les valeurs indéfinies qui vont permettre de la définir comme un attribut de l'ensemble entier des entités du type de la variable.

- *Attribut virtuel d'un ensemble complet d'entités d'un type*
  On peut modifier légèrement l'exemple des commandes de la figure 2.3. pour qu'il s'adapte à notre proposition. L'ensemble des données devient vide et la règle de domaine de CLI s'écrit à présent CLI = CLIENT. On calcule, dès lors, la grandeur MONTANT-DU $_{CLI}$ pour toutes les entités du type CLIENT. On peut considérer que MONTANT-DU est un attribut virtuel de CLIENT.

Le modèle, dans ce contexte, définit les **règles d'inférence** qui étendent le schéma de la base de données. On rejoint là le domaine des bases de données déductives. Nous ne développerons pas plus avant cet aspect qui déborde du cadre limité de cette section.

II.2.8.2. Attributs dérivables

En prolongeant le concept d'attribut virtuel, nous débouchons spontanément sur la possibilité d'utiliser un modèle pour définir un attribut existant dans la base de données. Ainsi, la (les) grandeur(s) résultat(s) d'un tel modèle définit un objet de la base de données comme étant dérivable ou calculable.

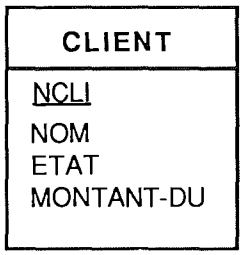
Nous pouvons partitionner l'ensemble de ces modèles en deux : les modèles dont toutes les données sont extraites de la base de données et les modèles dont certaines données (au moins une) ne sont pas issues de la base de données.

Examinons ces deux possibilités plus en détail.

Cas n°1 : *Toutes les données sont issues de la base de données*

Dans ce contexte, le modèle peut être considéré comme l'expression d'une **contrainte d'intégrité** de la base de données. Il s'agit du domaine de valeurs pour l'attribut calculable, grandeur résultat du modèle.

Exemple : On décide, pour des raisons de rapidité, d'ajouter au type d'entités CLIENT de la figure 2.3. l'attribut MONTANT-DU. On obtient ainsi :



MONTANT-DU est un attribut dérivable et, si on modifie le modèle comme dans l'exemple précédent (suppression de N dans les données et nouvelle définition de CLI (CLI = CLIENT), celui-ci permet d'exprimer la règle de calcul de MONTANT-DU.

Le nouveau modèle de calcul exprime dès lors une contrainte d'intégrité interne à la base de données.

Cas n°2 : *Certaines données ne sont pas extraites de la base de données*

Dans ce contexte,les règles du modèle peuvent être considérées comme des **règles de mise à jour** de la base de données.

Exemple : Illustrons ce cas à l'aide de notre exemple générique des clients et des commandes. Supposons que les FRAIS (attribut de COMMANDE initialement) varient selon les saisons et sont introduits par l'utilisateur. Nous gardons les modifications de l'exemple précédent afin de calculer le MONTANT-DU, attribut de l'entité CLIENT.

Nous obtenons le modèle suivant :

Données :

FRAIS $_{COM}$; frais attachés à une commande
avec ACQUISITION (COM)

Grandeurs internes :

CLI : CLIENT ; tout client
COM : COMMANDE ; commande du client CLI

Résultats :

MONTANT-DU $_{CLI}$: réel ; montant dû par le client CLI

Les règles restent identiques à la figure 2.3., à l'exception de la règle de domaine définissant CLI qui s'écrit CLI = CLIENT.

Ce modèle peut être considéré comme une règle de mise à jour car, pour chaque modification de MONTANT-DU, on introduira les frais et le modèle en déduira la nouvelle valeur.

II.2.8.3. Contraintes d'intégrité de la base de données

Dans le premier cas du point précédent, nous avons souligné la possibilité de décrire des contraintes d'intégrité de dérivation de valeur grâce au modèle. Prolongeons cette observation en nous intéressant à l'expression d'autres types de contraintes d'intégrité.

Exemple : Reprenons l'exemple des clients et des commandes. Supposons la contrainte d'intégrité suivante : l'attribut FRAIS de COMMANDE doit avoir une valeur nulle pour toute entité commande dont le montant des lignes est supérieur à X.

Cette contrainte pourrait s'exprimer sous la forme d'une égalité de deux ensembles : les commandes dont le montant est supérieur à X et les commandes dont les frais sont nuls. Dans ce modèle, il n'y aura ni donnée, ni résultat mais une **règle de contrainte**.

Grandeurs internes :

COM : COMMANDE ; toute commande
LC : LIGNE ; ligne de commande de COM
PRO : PRODUIT ; produit concerné par la ligne de commande LC
COM-O : COMMANDE ; commande dont les frais sont nuls
COM-MX : COMMANDE ; commande dont le total est supérieur à X
MONTANT $_{COM}$: entier ; montant de la commande COM
MONTANT-LIGNE $_{LC}$: entier ; montant de la ligne LC

Règles de domaine :

COM = COMMANDE
LC = LIGNE(CL:COM)
PRO = PRODUIT(CL:COM)
COM-O = COMMANDE(:FRAIS=0)
COM-MX = COMMANDE(:MONTANT > X)

Règles du problème :

$$MONTANT_{COM} = \sum_{LC} MONTANT\text{-}LIGNE_{LC}$$

$$MONTANT\text{-}LIGNE_{LC} = Q_{LC} * PRIX_{PRO}$$

Règle de contrainte :

COM-O == COM-MX

Nous pouvons également créer un modèle dont le but est de contrôler si la contrainte est vérifiée pour chaque commande:

Résultat :

FRAISNUL $_{COM}$: booléen ; est vrai si la contrainte d'intégrité que l'on vient d'exprimer est vérifiée pour la commande COM

Grandeurs internes :

COM : COMMANDE ; toute commande
LC : LIGNE ; ligne de commande de COM
PRO : PRODUIT ; produit concerné par la ligne de commande LC
$MONTANT_{COM}$: entier ; montant de la commande COM
$MONTANT\text{-}LIGNE_{LC}$: entier ; montant de la ligne LC

Règles de domaine :

COM = COMMANDE
LC = LIGNE(CL:COM)
PRO = PRODUIT(CL:COM)

Règles du problème :

$$MONTANT_{COM} = \sum_{LC} MONTANT\text{-}LIGNE_{LC}$$

$$MONTANT\text{-}LIGNE_{LC} = Q_{LC} * PRIX_{PRO}$$

$FRAISNUL_{COM}$ = faux si ($MONTANT_{COM}$ > X) et($FRAIS_{COM}$ > 0)
= vrai sinon

Notre modèle semble parfaitement s'adapter à l'expression des contraintes d'intégrité de toutes sortes. En fait, pour exprimer des contraintes d'intégrité dans un langage formel, on emploie généralement un langage de désignation de données. Dans les règles de domaine, nous employons un langage de requête de ce type. Cependant, exprimer des contraintes complexes dans de tels langages est un exercice périlleux pour un non-initié. Le langage de requête montre souvent trop peu de sophistication pour exprimer de telles règles. En ce sens, notre modèle pourrait s'envisager comme un prolongement du langage de désignation de données utilisé dans les règles de domaine.

Avant de s'interroger sur le lien requête/modèle, relevons quelques avantages de l'expression des contraintes d'intégrité sous forme de modèle.

Lors de l'élaboration d'un schéma de structuration des données (schéma Entité/Association par exemple), dès l'analyse conceptuelle, il est nécessaire d'exprimer des contraintes d'intégrité. A l'exception de quelques types de contraintes particuliers (identification, connectivité, inclusion de rôles ou d'associations,etc), il n'existe pas de représentation graphique des contraintes d'intégrité. On utilise alors le langage naturel pour exprimer ces dernières. Pour des contraintes relativement simples, le langage naturel se montre efficace ; mais, pour l'exemple du montant, il montre ses limites par rapport à tout langage formel (ambiguïté inhérente au langage naturel). Il faut donc utiliser un langage quelque peu formalisé pour la précision, mais la contrainte d'intégrité doit rester communicable aux utilisateurs du système d'information. Il semble qu'un modèle s'intègre parfaitement dans ce compromis "formalisation/communication" des contraintes d'intégrité.

Toutefois, l'utilisation de notre modèle peut se prolonger dans l'étape de mise en oeuvre de la solution fonctionnelle. Un certain nombre de systèmes de gestion de base de données (SGBD) offrent des facilités pour la gestion des contraintes d'intégrité. Cette gestion se fait grâce à un langage formel d'expression de contraintes d'intégrité. Ce langage de manipulation de données se montre souvent difficile à maîtriser pour exprimer des contraintes complexes. Dans cette optique, pourquoi ne pas imaginer un SGBD capable de gérer des contraintes d'intégrité mises sous forme d'un modèle (on rejoint dans ce cadre les bases de données déductives) ?

De plus, les SGBD offrent rarement ces facilités pour la gestion des contraintes d'intégrité (citons l'exemple du SGBD SYBASE). On est dès lors tenu de les intégrer dans les procédures de traitement de l'application. Dans ce contexte, on pourrait imaginer un générateur de code (Partie IV) capable de générer un programme exécutable à partir du modèle. Même si ce générateur n'existe pas, il sera bien plus aisé pour le programmeur de "traduire" en programme une contrainte exprimée sous forme d'un modèle qu'une autre exprimée en langage naturel.

II.2.8.4. Langage de requête

Pour désigner des sous-ensembles de données (valeurs d'attributs et d'entités), on utilise un langage de désignation de données (LDA). Tout système de gestion de bases de données possède un tel langage. Ce dernier permet d'exprimer des requêtes pour extraire des données de la base. Comme il a été mentionné précédemment, il est également utilisé pour exprimer des contraintes d'intégrité. Nous utilisons un langage de désignation de données pour la définition des règles de domaine.

Toutefois, le pouvoir d'expression de ces langages se limite souvent à la navigation à travers un petit nombre d'associations. L'algèbre relationnel se montre plus puissant, mais reste difficile à manipuler pour les non initiés. Pourquoi ne pas utiliser notre modèle pour exprimer des requêtes complexes sur la base de données ?

Exemple : Quels sont les clients qui ont commandé pour un montant supérieur à Y?
Nous reprenons le modèle de la figure 2.3. en modifiant quelque peu la définition de la variable entité CLI et en introduisant un nouveau résultat CLI-MY :

Résultat :

CLI-MY : CLIENT ; clients qui ont commandé pour un montant supérieur à Y

Grandeurs internes :

CLI : CLIENT ; tout client

MONTANT-DU $_{CLI}$: entier ; montant dû par le client CLI

Règles de domaine :

CLI = CLIENT

CLI-MY = CLIENT(:MONTANT-DU > Y)

Dans cet exemple, l'utilisation de la grandeur interne MONTANT-DU $_{CLI}$ (attribut virtuel de CLIENT) dans la règle de domaine définissant le résultat permet d'exprimer la requête. Dans cet emploi, notre modèle apparaît comme une **extension du langage de requête** défini par Hainaut [HAINAUT,86].

II.2.8.5. Utilisation et extension du concept de variable entité

Le concept de variable entité peut permettre de faciliter l'expression de requête d'extraction de données. Nous avons vu qu'une règle de domaine définit le domaine de la variable entité. **Nous allons à présent utiliser le nom de cette variable pour désigner son domaine.**

Exemple : Soit CLI une variable entité qui désigne tous les clients de la base de données (CLI = CLIENT).

Nous voudrions définir la variable entité COM-0 qui désigne l'ensemble des commandes de CLI dont les frais sont nuls.

On l'exprime par la règle de domaine suivante :

COM-0 = COMMANDE((CC:CLI) et (FRAIS=0)).

On pourrait également écrire :

COM = COMMANDE(CC:CLI) et

COM-0 = COM(:FRAIS=0).

Cette dernière règle définit COM-0 comme un sous ensemble de celui de COM.

Donc, nous permettons de définir des variables entité (Ves) comme des sous-ensembles d'une variable entité existante (Ve). On dira que les variables entité Ves désignent des **ensembles dérivés** de Ve.

Que va nous permettre la mise en place d'une telle extension ?

1. Un langage de requête incrémental

Ce concept d'ensemble dérivé va nous permettre d'exprimer les règles de domaine de manière incrémentale.

Exemple : Quels sont les clients qui ont passé au moins 10 commandes de plus de X francs comprenant le produit de numéro "BU89" et qui ont un état de compte positif ?

Décomposons la requête et adjoignons à chaque sous-requête une variable entité :

1. Les commandes qui ont au moins une ligne concernant le produit de numéro "BU89" :

COM-1 = COMMANDE(CL:LIGNE(PL:PRODUIT(:NPRO="BU89")))

2. Les commandes qui ont au moins une ligne concernant le produit de numéro "BU89" et de montant supérieur à X :
   COM-2 = COM-1 (:MONTANT-COM > X)
   (le calcul de MONTANT-COM $_{COM}$ est analogue à celui de la figure 2.3)
3. Les clients qui ont un état de compte positif :
   CLI-1 = CLIENT(:ETAT > 0)
4. Les clients de type (3) et qui ont passé des commandes de type (2) :
   CLI-RES = CLI-1 (CC: 10..* COM-2)

Cet exemple nous permet de déceler les possibilités de mise en place d'un langage de requête incrémental reposant sur les notions de variable entité et d'ensemble dérivé.

2. Ensemble dérivé et héritage

On peut également affirmer que tout ensemble dérivé d'une variable entité **hérite** des définitions qui dépendent de cette dernière.

Exemple : Soit un modèle qui, pour un client de numéro N calcule le montant de toutes ses commandes et le montant de ses commandes dont les frais sont nuls.

Donnée :
N : entier ; numéro du client

Résultats :
MONTANT-DU $_{CLI}$: entier ; montant dû par le client CLI
MONTANT-DU-0 $_{CLI}$: entier ; montant des commandes du client CLI dont les frais sont nuls

Grandeurs internes :
CLI : CLIENT ; client de numéro N
COM : COMMANDE ; commande du client CLI
COM-O : COMMANDE ; commande du client CLI dont les frais sont nuls
LC : LIGNE ; ligne de la commande COM
PRO : PRODUIT ; produit concerné par la ligne de commande LC
MONTANT-COM $_{COM}$: réel ; montant de la commande COM
MONTANT-LIGNE $_{LC}$: réel ; montant de la ligne LC

Règles de domaine :
CLI = CLIENT(:NCLI=N)
COM = COMMANDE(CC:CLI)
LC = LIGNE(CL:COM)
COM-0 = COM(:FRAIS=0)

Règles du problème :

$$\text{MONTANT-DU}_{CLI} = \sum_{COM} \text{MONTANT-COM}_{COM} + \text{ETAT}_{CLI}$$

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LIGNE}_{LC} + \text{FRAIS}_{COM}$$

$$\text{MONTANT-LIGNE}_{LC} = Q_{LC} * \text{PRIX}_{PRO}$$

$$\text{MONTANT-DU-0}_{CLI} = \sum_{\mathbf{COM\text{-}0}} \text{MONTANT-COM}_{\mathbf{COM\text{-}0}} + \text{ETAT}_{CLI}$$

Dans cette dernière expression, exprimant la valeur des commandes sans frais du client CLI, le domaine de LC est redéfini en fonction de COM-O en lieu et place de COM.
Ainsi, COM-0 hérite des définitions de la variable entité LC, mais aussi du calcul des grandeurs internes MONTANT-COM_{COM} et $\text{MONTANT-LIGNE}_{LC}$

II.3. Etude de cohérence

Cette section est consacrée à l'exposé de règles destinées à vérifier la cohérence d'un modèle. Un modèle est cohérent lorsqu'il n'existe **aucune contradiction entre ses objets**. Les objets d'un modèle, ce sont les grandeurs. Pour entrer en contradiction, les grandeurs doivent être mises en rapport. Les grandeurs sont "réunies" dans les règles du problème. C'est donc essentiellement au niveau de ces dernières que va porter l'étude de cohérence.

Ces règles de cohérence sont généralement vérifiées à postériori pour contrôler l'état des spécifications. L'exposé de la démarche méthodologique (partie III) indiquera les moments recommandés pour procéder à ces vérifications.

Des incohérences peuvent être détectées **au niveau d'une seule règle**. La première vérification étudiée consistera en un contrôle de l'utilisation des dimensions. Le GDV du modèle sera un outil primordial pour effectuer ce contrôle. Dans l'analyse des concepts, nous avons défini la relation de dépendance entre variables entité, mais nous n'avons donné aucune méthode pour créer ce GDV. C'est pourquoi, dans le premier point de cette étude de cohérence, nous exposerons une démarche d'analyse des règles de domaine des variables entité afin d'établir le GDV d'un modèle. Cette démarche est complexe et le lecteur pressé pourra faire fi de celle-ci en supposant que l'on est capable de créer un GDV. Cependant, cette partie constitue un des aspects le plus original de notre travail de recherche et le lecteur intéressé par le domaine ne pourra l'éviter. Nous avons en plus agrémenter l'exposé théorique de nombreux exemples pour en faciliter la compréhension.

Ainsi, dans un premier temps, nous présenterons notre méthode de création du GDV d'un modèle (II.3.1.).

Dans le deuxième point, nous proposerons un certain nombre de règles de cohérence sur l'utilisation des indices dans les règles du problème. Nous utiliserons le GDV pour la cohérence des variables entité (II.3.2.1.). Nous énoncerons ensuite des règles de validation concernant les grandeurs dimensionnées par des variables simples (II.3.2.2) et les grandeurs simples (II.3.2.3.). Nous évoquerons également la possibilité d'un contrôle de type (II.3.2.4.).

Enfin, nous terminerons l'analyse de la cohérence au niveau d'une règle par l'étude de deux types de règles plus particulières : les règles à définitions multiples (II.3.2.5.) et les règles de récurrence (II.3.2.6.).

L'analyse de cohérence au niveau d'une seule règle constitue la première étape de notre étude de validation. La seconde a pour objectif de détecter des incohérences **au niveau de l'ensemble des règles** d'un modèle (II.3.3.). Nous tenterons d'établir, par exemple, dans quelles conditions deux règles sont distinctes, non contradictoires...

II.3.1. Création du graphe de dépendances entre variables entité.

Lors de l'analyse des variables entité, nous avons défini la relation de paramètrage qui peut exister entre celles-ci. Nous proposons à présent une méthode d'analyse des règles de domaine afin d'établir le GDV d'un modèle. Cette étape de construction est indispensable pour effectuer un contrôle efficient de l'emploi des variables entité dans un modèle.

Pour chaque règle de domaine définissant une variable entité, on établit la dépendance avec les variables entité qui apparaîssent en partie droite. Après avoir effectué cette opération pour chaque règle, on est en mesure de construire le GDV du modèle.

La méthode d'analyse se divise en **quatres grandes étapes** qui ont pour but de déceler la dépendance qui existe dans une règle de domaine, ainsi que sa classe fonctionnelle.

La première étape transforme la règle de domaine de manière à en "retirer" facilement les **conditions d'association simples** (ne contenant aucun opérateur logique). La deuxième étape, à partir de la syntaxe des règles de domaine, repère les relations de dépendance dans les conditions d'association simples. La troisième étape recompose les liens détectés à l'étape précédente en tenant compte des opérateurs logiques mis en évidence lors de la première étape. A partir de ces liens recomposés, à l'aide du schéma de la base de données, la quatrième étape établit la classe fonctionnelle de la dépendance analysée. Nous conseillons très vivement au lecteur de se replonger quelque peu dans la définition syntaxique et sémantique des règles de domaine (II.2.1.2.2.) avant d'entamer cette méthode d'analyse.

A la suite de ces quatres grandes étapes, nous analyserons les règles de domaine de l'étude de cas n°1 et établirons son GDV.

La plupart des exemples que nous allons employer sont basés sur le schéma entité/association de la figure 2.4. Nous ne nous attarderons pas sur ce schéma. Il est cependant important pour la suite de connaître les identifiants de chacun des types d'entités.

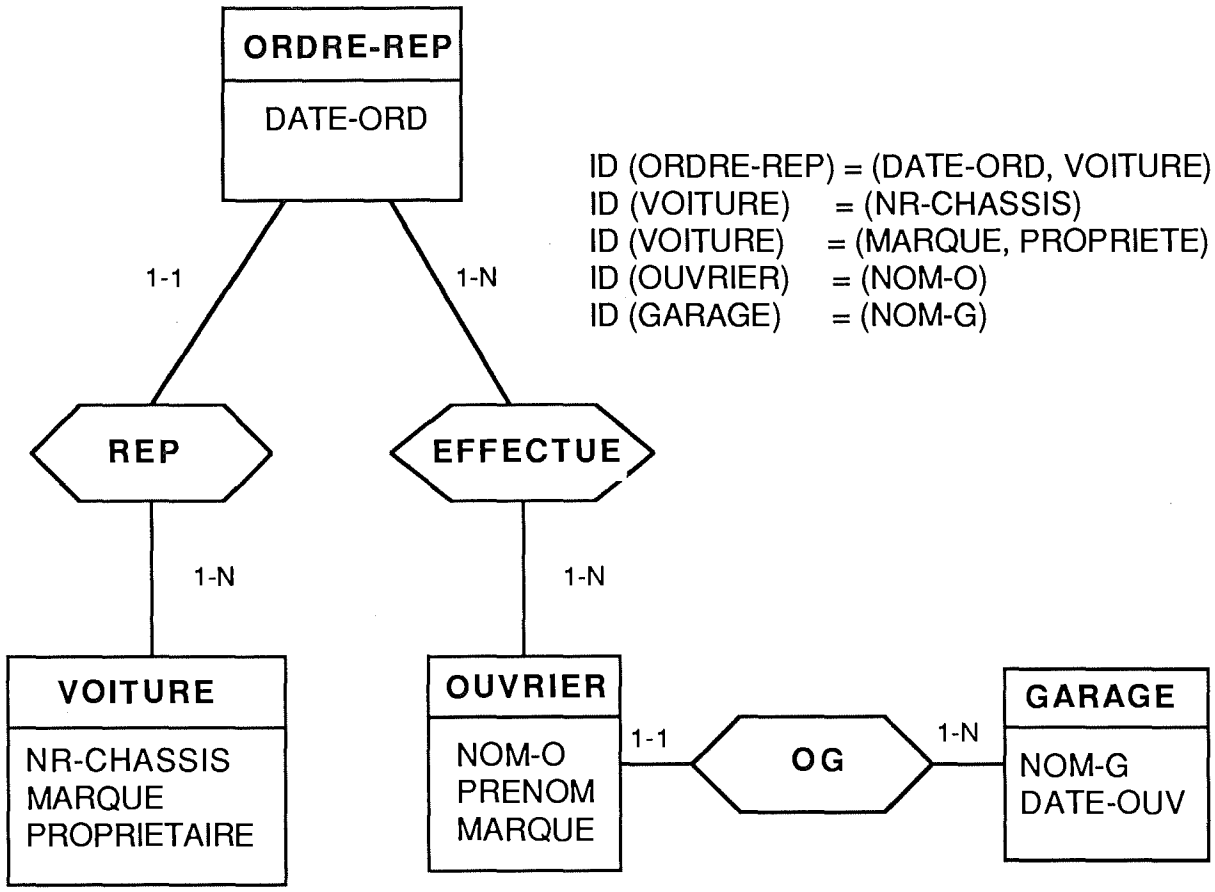


Figure 2.9.

II.3.1.1.<u>PREMIERE ETAPE</u> : Décomposition et distribution de la condition de sélection en conditions d'association simples.

De manière intuitive, la première étape a pour but de "retirer" tous les opérateurs logiques situés dans les conditions d'association pour les ramener vers le "haut" de l'expression.`

Exemple : OUV = OUVRIER(effectue:ORDRE-REP((:DATE-ORD=X) et (rep:VOI)))

Cette règle de domaine désigne l'ensemble des ouvriers qui ont effectué un ordre de réparation sur la voiture VOI à la date X.

La première étape a pour objectif de ramener l'opérateur "et", contenu dans la condition d'association portant sur ORDRE-REP, au niveau de la condition d'association portant sur OUVRIER. Après décomposition, on obtiendra

OUV = OUVRIER ((effectue:ORDRE-REP(:DATE-ORD=X)) et
(effectue:ORDRE-REP(rep:VOI)))

Ces deux règles de domaine ne sont pas sémantiquement équivalentes, mais cela ne nous importe pas à ce stade.

Lorsque cette opération est réalisée, dans les règles de domaine ou apparaîssent les deux types d'opérateurs logiques, on **distribue** les opérateurs logiques "et" par rapport aux opérateurs "ou" . Cet étape est possible car ils ont été rassemblés au plus "haut niveau" de la règle par l'opération de décomposition.

En résumé, de manière intuitive, cette étape devra transformer la condition de sélection afin qu'elle se présente de la manière suivante (en omettant un certain nombre de parenthèses) :

$$Csel = (Csel_1 \text{ ou } Csel_2 \text{ ou.....ou } Csel_M)$$

où $Csel_i$ ne contient aucun opérateur logique "ou".

Chaque $Csel_i$ sera alors décomposée en Cas_{ik},condition d'association simple, ne comportant aucun opérateur logique :

$$Csel_i = (\ Cas_{i1} \text{ et } Cas_{i2} \text{ et... } \text{...et } Cas_{iN})$$

où Cas_{ik} est une condition d'association simple.

A la suite de cette courte présentation intuitive, nous pouvons entamer une description plus formelle de cette première étape

La première étape poursuit **deux objectifs** :

1. décomposer la condition de sélection en conditions d'association ne contenant dans leur condition de sélection aucun opérateur logique.
2. toute condition de sélection qui est liée à une autre par un opérateur logique "et" ne pourra contenir une condition de sélection possédant un opérateur "ou".

Cette transformation portera sur **deux types de conditions de sélection** :

- celles qui définissent une règle de domaine : elles se rattachent directement au type d'entités de la variable entité définie. On a une expression du type : "Ve = Typeentité Csel",
  où Ve est la variable entité définie
  Typeentité est le type de Ve et sera appelé le **type d'entités initial**,
  Csel est la condition de sélection analysée.
- celles qui définissent un ensemble de valeurs de la base de données à l'intérieur d'une condition d'appartenance. On a alors une expression du type : "Attribut (: Typeentité Csel),
  où Typeentité est le **type d'entités initial**
  Csel est la condition de sélection analysée.

Le second objectif suppose la réalisation du premier : le processus se divisera donc en deux sous-étapes.

Sous-étape 1 : **Décomposition**

Une condition d'association se présente sous la forme "(A : Ens Cond)".
Soit Cas1, la première condition d'association où Cond est une condition de sélection du type (Csel1 opl Csel2) où Csel1 et Csel2 sont des conditions de sélection et opl est un opérateur logique (et/ou).

On effectue l'**opération de décomposition** suivante sur Cas1 :

Cas1 = (A : Ens (Csel1 opl Csel2))
devient
((A: Ens Csel1) opl (A:Ens Csel2)) ce qui peut s'écrire
(Csel1' opl Csel2')

Deux cas sont alors possibles :

1. Si A est un type d'associations entre le type d'entités initial et Ens, alors on peut effectuer la même opération sur les premières conditions d'association de Csel1 et Csel2 qui ont la même forme que Cas1.
2. Si A n'est pas un type d'associations entre le type d'entités initial et Ens, alors Cas1 constitue la partie Cond d'une autre condition d'association, soit Cas2. Cas1 ne peut faire partie d'une condition de sélection avec opérateur logique car on a supposé que Cas1 était la première condition d'association qui présentait cette caractéristique.
   On a en fait le schéma suivant :
   Cas2 = (B : Ens2 Cas1)
   = (B : Ens2 (Csel1' opl Csel2'))
   On peut dès lors appliquer l'opération de décomposition à Cas2.
   On applique cette transformation de manière itérative jusqu'à ce que le type d'associations lie la condition d'association obtenue au type d'entités initial.
   Dès que l'on a atteint cet état, on peut effectuer la même opération sur les premières conditions d'association de Csel1 et Csel2 qui ont la même forme que Cas1.

Lorsque cette étape est terminée, toutes les conditions d'association apparaissant dans la condition de sélection n'ont **plus aucun opérateur logique**. De plus, toutes ces conditions d'association **se rapportent au type d'entités initial.**

Exemple : VOI = VOITURE (rep:ORDRE-REP(effectue:OUVRIER((:PRENOM= "JEAN") et (og:GAR)))
GAR est une variable définie sur le type d'entités GARAGE.
Cette règle de domaine désigne toutes les voitures réparées par les ouvriers du garage GAR dont le prénom est Jean.
Soit Cas1 la première condition d'association rencontrée dont la partie Cond est une expression booléenne de conditions de sélection.
Cas1 = (effectue:OUVRIER((:PRENOM="JEAN") et (og:GAR)))
devient après décomposition
((effectue:OUVRIER(:PRENOM="JEAN")) et
(effectue:OUVRIER(og:GAR)))
"effectue" n'est pas un type d'associations entre VOITURE (le type d'entités initial) et OUVRIER. Cas1 joue donc le rôle de Cond pour une autre condition d'association (Cas2). On a ainsi :
Cas2 = (rep:ORDRE-REP Cas1)
devient après décomposition
((rep:ORDRE-REP(effectue:OUVRIER(:PRENOM="JEAN")))
et
(rep:ORDRE-REP(effectue:OUVRIER(og:GAR)))).

"rep" est bien un type d'associations entre VOITURE et OUVRIER.
On peut dès lors rechercher les conditions d'association de Csel1 (:PRENOM="JEAN") et Csel2 (og:GAR) dont la partie Cond est une expression booléenne.
Ce n'est pas le cas dans cet exemple et la décomposition s'arrête donc à cet état.
Cet exemple met bien en évidence l'état final dans lequel on arrive :
- plus d'opérateurs logiques dans les conditions d'association
- toutes les conditions d'association se rapportent au type d'entités initial.

Il est clair que cette décomposition (consistant en la distribution des opérateurs logiques) s'accompagne d'une **dégradation sémantique** de la Csel. En effet, dans l'exemple, la deuxième règle de domaine désigne un ensemble qui inclut le premier. Cependant, cette dégradation ne gêne pas dans l'analyse que l'on se propose de mener.

Sous-étape 2 : **Distribution**

Cette distribution s'effectue sur des conditions de sélection décomposées. Elle s'inspire de la forme OU de la loi distributive de l'algèbre de Boole.
Cette loi peut s'écrire : si A, B et C sont des propositions,

((A ou B) et C) = ((A et C) ou (B et C))

Dans notre cas,nous appliquons cette loi aux conditions de sélection. Pour toute condition de sélection qui se présente sous les formes :

$((C_{21}$ ou $C_{22})$ et $C_3)$ ou $(C_2$ et $(C_{31}$ ou $C_{32}))$

où C_2, C_3, C_{21}, C_{22}, C_{31}, C_{32} sont des conditions de sélection,
on effectue l'**opération de distribution** suivante :

$((C_{21}$ et $C_3)$ ou $(C_{22}$ et $C_3))$ ou $((C_2$ et $C_{31})$ ou $(C_2$ et $C_{32}))$

Exemple : ORD=ORDRE-REP(((:DATE-ORD=X) ou (:DATE-ORD=Y)) et
(rep:VOI))
Cette règle de domaine désigne les ordres de réparation effectués à la date X ou à la date Y et qui concernaient la voiture VOI (variable entité définie sur VOITURE).
Cette règle devient après distribution
ORD=ORDRE-REP(((:DATE-ORD=X) et (rep:VOI)) ou
((:DATE-ORD=Y) et (rep:VOI)))

Reprenons un dernier exemple où les deux sous-étapes sont mêlées.
Exemple : OUV=OUVRIER((effectue:ORDRE-REP(rep:VOITURE((: NR-
CHASSIS=X) ou (NR-CHASSIS=Y)))) et
(:DATE-ORD=Z))
Cette règle désigne les ordres de réparation concernant les voitures de numéro de chassis X ou Y et qui ont été effectué à la date Z.
Cette règle de domaine devient après décomposition et distribution :

```
OUV=OUVRIER(((effectue:ORDRE-REP(rep:VOITURE(:NR-
                    et                    CHASSIS=X)))
          (effectue:ORDRE-REP(:DATE-ORD=Z)))
                    ou
          ((effectue:ORDRE-REP(rep:VOITURE(:NR-
                    et                    CHASSIS=Y)))
          (effectue:ORDRE-REP(:DATE-ORD=Z))))
```

Nous pouvons illustrer le résultat intuitif présenté au début de cette première étape.
La condition de sélection finale se présente sous la forme :

(Csel1 ou Csel2)

où Csel1, Csel2 sont des conditions de sélection ne contenant aucun opérateur logique "ou".
Csel1 et Csel2 se décomposent alors en conditions d'association ne comportant aucun opérateur logique.
On a par exemple : Csel1= (Cas11 et Cas12)
où Cas11 = (effectue:ORDRE-REP(rep:VOITURE (:NR-CHASSIS=X))),
Cas12 = (effectue:ORDRE-REP(:DATE-ORD=Z))

II.3.1.2. <u>DEUXIEME ETAPE</u> : Analyse d'une condition d'association simple

Cette étape a pour objectif de déceler la présence d'une dépendance dans une condition d'association simple. Pour ce faire, nous allons classer les conditions d'association en huit grandes catégories. Cette classification sera très importante pour l'établissement de la classe fonctionnelle, objet de la quatrième étape.

Cette répartition des conditions d'association simples se base sur la définition syntaxique et sémantique des règles de domaine. De nouveau, nous conseillons au lecteur de s'y reporter.

Chaque catégorie sera illustrée par un exemple. Il s'agira de règles de domaine qui ne contiennent qu'une condition d'association simple afin de ne pas compliquer outre mesure.

Si on se rapporte à la définition des règles de domaine, une variable entité peut apparaître à trois endroits différents. Nous ne reprendrons pas la syntaxe à ce niveau, mais pour permettre au lecteur de s'y référer, nous utiliserons les références qui ont été placées en marge droite lors de la présentation de ce point (II.2.1.2.2.).

Une variable entité peut apparaître dans une condition d'association où elle joue le rôle de type d'entités (3). Une variable peut également intervenir dans une condition d'appartenance, soit en tant que dimension d'une grandeur d'un modèle (6), soit dans la condition de sélection définissant un ensemble de valeurs (8). Nous percevons déjà le plan de la classification effectuée : variable entité dans une condition d'association (cas n°1), variable entité dans une condition d'appartenance (cas n°2), avec la distinction entre grandeur indicée (2.1.) et condition de sélection (2.2.).

Soit **Ve**, la variable entité définie par la règle de domaine dont on a extrait la condition d'association simple analysée.

Cas n°1 : **aucune condition d'appartenance n'apparaît dans la condition d'association**

Pour qu'une variable entité apparaisse dans une condition d'association, elle ne doit être suivie d'aucune condition de sélection. Comme nous analysons des conditions d'association simples, la variable entité apparaîtra dans la **dernière condition d'association**. C'est pourquoi, nous distinguons deux cas selon le contenu de la dernière condition d'association.

Règle 1.1. :

Si le dernier ensemble spécifié dans la condition d'association est une variable entité V Alors V ⟶ Ve

Dans ce cas, à une valeur de V correspond au plus un domaine de valeurs pour Ve. Nous retrouvons la relation de paramètrage entre variables entité (II.2.1.3.).

Exemple : ORD = ORDRE-REP (rep : VOI)

La règle de domaine désigne l'ensemble des ordres de réparation qui concernent la voiture VOI.

Le dernier ensemble spécifié dans la condition d'association est la variable entité VOI. On obtient donc :

VOI ⟶ ORD

Règle 1.2. :

Si le dernier ensemble spécifié dans la condition d'association est un type d'entités Alors Ve est dite indépendante.

Exemple : VOI = VOITURE (rep : ORDRE-REP)

La règle de domaine désigne l'ensemble des voitures qui ont subi au moins une réparation.

Le dernier ensemble spécifié dans la condition d'association est le type d'entités ORDRE-REP. VOI est donc indépendante.

Cas n°2 : **Une condition d'appartenance apparaît dans la condition d'association**

Dans l'introduction de cette étape, nous avons distingué deux cas d'apparition de variables entité dans une condition d'appartenance : soit comme indice d'une grandeur, soit dans une condition de sélection définissant un ensemble de valeurs. On peut distinguer ces deux cas en analysant l'opérateur de la condition d'appartenance. Si c'est un opérateur de comparaison entre valeurs, la

variable entité éventuelle devra jouer le rôle d'indice. Si c'est un opérateur ensembliste, la variable entité éventuelle apparaîtra dans la condition de sélection.

2.1. La première Rel rencontrée est du type (<,=,>,≥,≤).
Soit **E1 Rel E2**, la condition d'appartenance.

Règle 2.1.1. :

Si E1 est un attribut du type d'entités de Ve et E2 est une grandeur non indicée par une variable entité Alors Ve est dite indépendante.

Exemple : VOI=VOITURE(:MARQUE="RENAULT")
E1 = MARQUE ET E2 = "RENAULT", une grandeur non indicée par une variable entité. VOI est donc indépendante.

Règle 2.1.2. :

Si E1 est un attribut du type d'entités de Ve et E2 est une grandeur indicée par une variable entité V Alors V ⟶ Ve

Exemple : GAR = GARAGE(:NOM-G=NOM-O $_{OUV}$)
La règle de domaine désigne les garages dont le nom est celui de l'ouvrier OUV.
E1 = NOM-G et E2 = NOM-O $_{OUV}$. La variable entité GAR est paramètrée selon OUV, ce qui s'écrit :

OUV ⟶ GAR

Règle 2.1.3.

Si E1 est attribut d'un type entités ENT associé à Ve et E2 est une grandeur non indicée par une variable entité Alors Ve est dite indépendante.

A une valeur de V correspond au plus une valeur de la grandeur qu'elle dimensionne. De plus, pour chaque valeur de cette grandeur, nous avons un domaine différent pour Ve. Nous retrouvons la relation de dépendance fonctionnelle existant entre V et Ve par l'intermédiaire de la grandeur indicée par V.
Exemple : ORD=ORDRE-REP(rep:VOITURE(:NR-CHASSIS= "34RT56Y"))
La règle de domaine désigne les ordres de réparation qui concerne la voiture dont le numéro de chassis est "34RT56Y".
ENT = VOITURE, E1 = NR-CHASSIS et E2 est la valeur "34RT56Y" (donc E2 n'est pas une grandeur indicée par une variable entité).
ORD est donc indépendante.

Règle 2.1.4. :

Si E1 est un attribut d'un type d'entités ENT associé à Ve et E2 est une grandeur indicée par une variable entité V Alors V ⟶ Ve

A une valeur de V correspond au plus une valeur pour la grandeur indicée par V. Chaque valeur de cette grandeur définit au plus un domaine pour Ve. Ve est donc paramètrée selon V par l'intermédiaire de la grandeur indicée.

Exemple : OUV=OUVRIER(effectue:ORDRE-REP(:DATE-ORD=DATE-OUV $_{GAR}$)

Cette condition d'association simple désigne les ouvriers qui ont effectué une réparation le jour d'ouverture du garage GAR.

ENT = ORDRE-REP, E1 = DATE-ORD et E2 = DATE-OUV $_{GAR}$, une grandeur indicée par une variable entité. La variable entité OUV est donc paramètrée selon GAR :

GAR ⟶ OUV

2.2. La première relation rencontrée Rel est du type (in,not-in,==).

Soit **E1 Rel Attribut(:Ens Csel)** avec Ens l'ensemble d'entités dont l'attribut est relié à E1 par Rel.

Règle 2.2.1. :

Si le domaine défini par "Ens Csel" est paramètré selon une variable entité V. (sur Ens porte une condition de sélection (Csel) sur laquelle on reporte l'analyse) Alors V ⟶ Ve

Exemple : ORD=ORDRE-REP(:DATE-ORD in DATE-OUV(:GARAGE(:NOM-G= NOM-O $_{OUV}$))))

Ens=GARAGE est paramètré selon OUV selon la règle 2.1.2. On a donc :

OUV ⟶ ORD

2.2.2. L'ensemble des autres cas est inintéressant pour les deux étapes suivantes (car Ens n'est paramètré par aucune variable entité et comme Ve est définie sur Ens, elle n' aucune chance de dépendre d'une autre variable entité)

II.3.1.3. TROISIEME ETAPE : Recomposition

Cette étape a pour but d'établir les diverses relations entre une variable entité et les différentes variables entité référencées dans sa règle de domaine.

La première étape décompose la règle de domaine en conditions d'association simples. La deuxième étape analyse chacune de ces conditions. A partir des dépendances décelées lors de cette deuxième phase, on recompose en une dépendance finale, liant la variable entité définie et celles qui la déterminent, selon les opérateurs logiques (et/ou) mis en évidence lors de la première étape.

Toutes les dépendances découvertes lors de la deuxième étape doivent être réunies. En effet, si deux variables entité apparaissent dans la règle de domaine de la variable définie, celle-ci est paramètrée conjointement par ces deux dimensions. A un couple de valeurs de ces deux paramètres correspond au plus un domaine de valeur pour la variable entité définie.

Exemple : ORD=ORDRE-REP((rep:VOI) et (effectue:OUV))

La règle de domaine désigne tous les ordres de réparation effectués par l'ouvrier OUV sur la voiture VOI.

Etape 1 : il n'y a aucune décomposition à effectuer.

On obtient les conditions d'associations simples :

Cas_1 =(rep:VOI) et

Cas_2 =(effectue:OUV)

Etape 2 : - analyse de Cas_1

La condition d'association est conforme à la règle 1.1.

Cela entraîne :

VOI ⟶ ORD

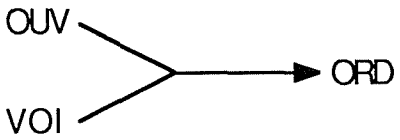
- analyse de Cas_2

La condition d'association respecte également la règle 1.1.

Cela entraîne :

OUV ⟶ ORD

Etape 3 : la variable entité ORD est paramètrée selon les variables entité VOI et ORD. On recompose donc comme suit :



Présentons à présent cette étape de recomposition de manière plus générale.

Suite à la décomposition de la première étape, la Csel se présente comme suit (en omettant un certain nombre de parenthèses) :

$$Csel = (Csel_1 \text{ ou } Csel_2 \text{ ou... } ...\text{ou } Csel_M)$$

Toute $Csel_i$ avec i dans {1..M} est une expression booléenne de conditions d'association reliées par des opérateurs "et".

On a ainsi :

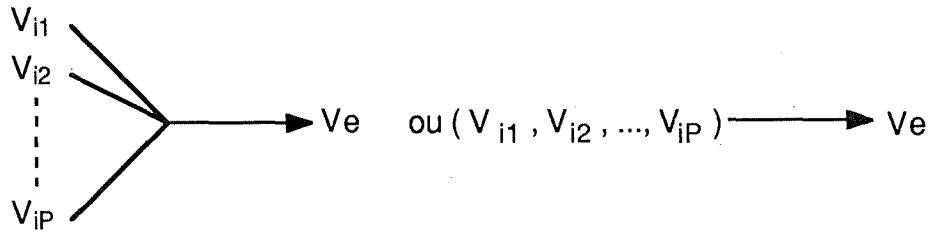
$$Csel_i = (Cas_{i1} \text{ et } Cas_{i2} \text{ et... } ...\text{et } Cas_{iN})$$

où Cas_{ik} avec k dans {1..N} est une condition d'association simple (sans opérateur logique).

Ce sont ces Cas $_{ik}$ qui sont analysées lors de la deuxième étape. Pour un certain nombre d'entre elles (soit P ce nombre), on a établi :

$V_{ik} \longrightarrow Ve$

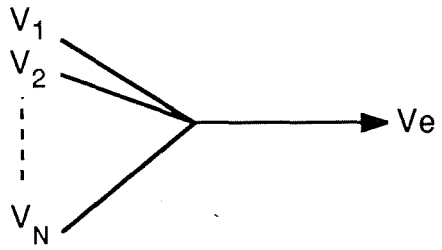
Comme ces conditions d'association sont reliées par un opérateur "et", Ve est paramètrée conjointement selon tous les V_{ik} avec k dans {1..N}. Ce qui donne :



Soit $V_i = V_{i1}, V_{i2}, ..., V_{iP}$ avec i dans {1..M}

Une condition de sélection décomposée est une expression booléenne d'opérateurs "ou". Cela entraîne que Ve est paramètrée selon l'ensemble de variables entité V_1 ou l'ensemble V_2 ou... ...ou l'ensemble V_N.

Cependant, le domaine de Ve est paramètré conjointement selon toutes les variables entité de sa règle de domaine. Nous avons donc :



Remarque : on pourrait se demander pourquoi avoir distribué les conditions de sélection. Cette phase ne montrera son intérêt que dans la quatrième étape d'établissement du cardinal de la dépendance.

II.3.1.4. QUATRIEME ETAPE : établissement de la classe fonctionnelle

A la suite de la troisième étape, on dispose de la relation entre la variable entité définie et celles dont elle dépend. Il ne reste plus qu'à étiqueter cette dépendance.

Lorsque nous avons défini la notion de dépendance, nous avons distingué deux classes fonctionnelles de relation : relation 1 et relation N. Nous avons défini cette notion par rapport au cardinal du domaine de la variable entité. La classe fonctionelle est 1 si le domaine de la variable entité définie contient au plus une valeur pour chaque ensemble de valeurs de ses paramètres.

Si ce domaine contient au plus une valeur, alors la règle de domaine **identifie** le type de la variable entité définie.

Nous pouvons dès lors donner une définition équivalente de la notion de classe fonctionnelle :

Si une variable entité Ve est paramètrée selon une variable entité V et Si sa règle de domaine **identifie** son type Alors la classe fonctionnelle de la dépendance de Ve vis-à-vis de V sera 1. Elle sera N sinon.

La détermination de la classe fonctionnelle fait appel au mécanisme d'identification de type d'entités. La quatrième étape sera donc très liée au **schéma de la base de données**.

Nous allons établir la classe fonctionnelle en deux étapes qui se basent sur la structure d'arbre établie lors de l'étape précédente. La première s'occupe des branches "et" de l'arbre et la seconde établit la classe fonctionnelle pour l'arbre "ou".

Dans un premier point, nous allons donc établir le cardinal de la relation pour chaque Csel ¡ (condition de sélection où n'apparaissent que des opérateurs logiques "et").

Dans un second point, nous résoudrons le problème de la recomposition pour les opérateurs logiques "ou".

Classe fonctionnelle d'une condition de sélection où n'apparaissent que des opérateurs logiques "et".

Rappelons que la condition de sélection analysée a été décomposée, c'est-à-dire que toutes les conditions d'association apparaissant dans celle-ci ne contiennent aucun opérateur logique et se rapportent toutes au type d'entités initial (type de la variable entité définie).

A la suite des trois premières étapes, nous connaissons les différentes variables entité paramètres de la variable définie. Il reste à voir si la règle de domaine identifie le type d'entités initial. La condition de sélection analysée peut s'écrire (en omettant un certain nombre de parenthèses) :

$$Csel = (\ Cas_1 \text{ et } Cas_2 \text{ et...} \quad ...\text{et } Cas_N)$$

où Cas_k est une condition d'association simple.

Pour toute Cas_k se rapportant au type d'entités initial, on va vérifier si la condition d'association est capable d'identifier ce type d'entités, soit à elle seule, soit en liaison avec d'autres Cas_k.

A partir des différentes Cas_k, nous allons constituer un vecteur de types d'entités et d'attributs. Si ce vecteur VI (appelé **vecteur d'identification**) est un identifiant strict ou non strict du type d'entités initial, alors la relation est une relation 1, sinon c'est une relation N.

Exemple : Soit VOI une variable entité définie sur voiture.
Soit X une grandeur quelconque.
ORD=ORDRE-REP((rep:VOI) et (:DATE-ORD=X))
Cette règle de domaine désigne tous les ordres de réparation effectués sur la voiture VOI à la date X.

Etape 1 : on obtient les conditions d'association simples suivantes :
Cas_1 = (rep:VOI)
Cas_2 = (:DATE-ORD=X)
Csel = (Cas_1 et Cas_2)

Etape 2 : Pour Cas_1, on a établi

VOI ⟶ ORD

Pour Cas_2, on a que ORD est indépendante

Etape 3 : il n'y a aucune recomposition à effectuer

Etape 4 : on constitue le vecteur d'identification VI
(VOITURE, DATE-ORD).
VI est un identifiant du type d'entités initial ORDRE-REP.
On obtient donc :

$$VOI \xrightarrow{1} ORD$$

Pour **constituer le vecteur d'identification,** on analyse les Cas_k de la condition de sélection les unes après les autres. Pour connaître les valeurs à y introduire, on se base sur la classification des conditions d'association simples effectuées lors de la deuxième étape.

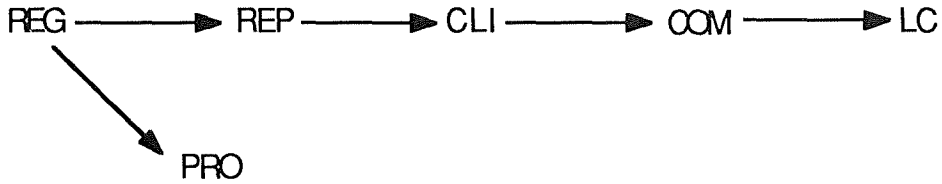
Soit **Ve** la variable entité définie par Csel, la condition de sélection analysée.

Si Cas_k est conforme à la **règle 1.1**, alors Ve est paramètrée selon V. On place dans VI, le type d'entités de V. On y place aussi tous les types d'entités des variables entité qui sont ancêtres de V dans le graphe de dépendances. En effet, Ve est également paramètrée par les ancêtres de V qui sont susceptibles d'identifier son type.

Exemple : (étude de cas n°2)

VR=VENTEREG(vp:PRODUIT(pl:LC))

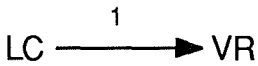
A la suite de l'analyse des règles de domaine précédente, on a établi le GDV suivant :



A la suite des trois premières étapes d'analyse de la règle de domaine de VR on a :

La condition de sélection contient une seule condition simple qui est conforme à la règle 1.1. On place alors dans le vecteur d'identification le type d'entités de LC (LIGNECOM) et les types d'entités de ses ancêtres dans le graphe. On obtient alors le vecteur d'identification suivant : VI = (LIGNECOM, COMMANDE, CLIENT, REPRESENTANT, REGION).

Or, (LIGNECOM, REGION) est identifiant de VENTEREG. Le véritable identifiant de VENTEREG est le couple (PRODUIT, REGION). Mais le chemin (LIGNECOM, PRODUIT) est de classe fonctionnelle N-1, et après composition des associations, on peut considérer (LIGNECOM, REGION) comme identifiant de VENTEREG. Cela entraîne



Si Cas $_k$ est conforme à la **règle 2.1.2.**, alors on place dans VI le type d'entités de V et les types d'entités de ses ancêtres.

De plus, que ce soit pour les règles 2.1.1. ou 2.1.2., la condition d'association se présente comme suit : "(:E1 Rel E2)" où E1 est un attribut du type d'entités initial et E2 une valeur unique. Cette condition d'association est susceptible d'identifier le type de Ve si - E1 est identifiant ou fait partie d'un identifiant composé du type d'entités initial et

- Rel est l'opérateur d'égalité "=".

Dans ce cas, on adjoint au vecteur d'identification l'attribut E1.

Exemple : nous pouvons reprendre l'exemple introductif de cette étape.

Soit VOI une variable entité définie sur voiture.

Soit X une grandeur quelconque.

ORD=ORDRE-REP((rep:VOI) et (:DATE-ORD=X))

A la suite des trois premières étapes, on a que ORD est paramètrée selon VOI.

Cas$_1$ = (rep:VOI) est conforme à la règle 1.1. En supposant que la variable entité VOI n' a pas d'ancêtres, on introduit VOITURE dans le vecteur d'identification.

Cas$_2$ = (:DATE-ORD=X) est conforme à la règle 2.1.1. On adjoint l'attribut DATE-ORD à VI, car il fait partie d'un identifiant composé de ORDRE-REP.

Finalement, on obtient le vecteur (VOITURE, DATE-ORD) qui est un identifiant de ORDRE-REP. Cela entraîne

Si Cas_K est conforme à la **règle 2.1.4.**, alors on place dans VI, le type d'entités de V et les types d'entités de ses ancêtres.

De plus, si Cas_K est conforme à la règle 2.1.3. ou 2.1.4., la condition d'association analysée comporte une condition d'association du type "(:E1 Rel E2)". Mais celle-ci n'est pas rattachée au type d'entités initial, mais à un autre type d'entités (associé au type initial) soit ENT.

Cette condition d'association est susceptible d'identifier le type d'entités de Ve si ENT(:E1 Rel E2) désigne une entité unique et ENT fait partie d'un identifiant du type initial.

Si E1 est un attribut identifiant de ENT et Rel est l'opérateur "=", alors ENT(:E1 Rel E2) est une entité unique et le type d'entités ENT peut être incorporé au vecteur d'identification.

Si E1 fait partie d'un identifiant composé de ENT et Rel = "=", alors ENT n'est pas une entité unique, mais elle pourrait le devenir avec l'aide d'autres conditions d'association simples. On va construire un vecteur d'identification propre à ENT, soit Y_{ENT}. Ce vecteur va permettre de savoir s'il n'est pas possible d'identifier ENT grâce à d'autres conditions d'association simples. On introduit dans Y_{ENT} l'ensemble des types d'entités déjà retenus dans VI. Si ce nouvel ensemble constitue un identifiant du type ENT, alors ENT est introduit dans VI.

Sinon, elle est mise à l'écart jusqu'à ce que

- soit un nouveau type d'entités rentre dans VI. Il est également introduit dans Y_{ENT}.
- soit une nouvelle Cas_K est conforme à la règle 2.1.3. ou 2.1.4. et sa condition d'appartenance porte également sur ENT. On ajoute à Y_{ENT} E1 si c'est un attribut faisant partie d'un identifiant de ENT et si Rel est l'opérateur "=".

Si le nouveau vecteur Y_{ENT} obtenu identifie ENT, alors celle-ci est introduite dans VI. Sinon elle reste à l'écart.

Les exemples 1 et 2 qui seront présentés à la suite mettent en oeuvre cette notion de vecteur "auxiliaire" Y_{ENT}.

Si Cas_K est conforme à la **règle 2.2.1.**, alors on place dans VI, le type d'entités de V et les types d'entités de ses ancêtres.

Si Cas_K ne rentre dans aucun cas précité, le vecteur d'identification VI reste inchangé.

A la fin de cette procédure, on a construit le vecteur d'identification du type d'entités de Ve. Il ne reste plus qu'à voir si VI constitue un identifiant du type d'entités de Ve.

Nous proposons à présent quelques **exemples** pour se familiariser avec le principe d'identification.

Exemple 1 : GAR=GARAGE(:NOM="XXXX")
ORD=ORDRE-REP((rep:VOITURE((:MARQUE="RENAULT") et
(:PROPRIETAIRE="LUC")))
et
(:DATE-ORD=DATEOUV GAR))

Cette règle de domaine désigne les ordres de réparation qui concernent la voiture de marque RENAULT appartenant à LUC et qui

ont été effectués lors de l'ouverture du garage GAR. Procédons à l'analyse de cette règle de domaine.

Etape 1 : La règle décomposée se présente comme suit :

ORD=ORDRE-REP(((rep:VOITURE(:MARQUE="RENAULT")) et
(rep:VOITURE(:PROPRIETAIRE="LUC"))) et
(:DATE-ORD=DATEOUV GAR))

Etape 2 : seule la dernière condition d'association simple "(:DATE-ORD=DATEOUV GAR)" contient une variable entité. Elle est conforme à la règle 2.1.2. Nous obtenons la relation suivante :

GAR ⟶ ORD

Etape 3 : il n'y a aucune recomposition à effectuer, GAR étant la seule variable entité présente dans la règle de domaine.

Etape 4 :

La première condition d'association est conforme à la règle 2.1.3. Dans ce cas ENT = VOITURE, E1 = MARQUE et Rel= "=". MARQUE fait partie d'un identifiant composé de VOITURE. On crée ainsi un vecteur $Y_{VOITURE}$ dans lequel on incorpore MARQUE.

La deuxième condition d'association est également conforme à la règle 2.1.3. De plus, sa condition d'appartenance porte également sur VOITURE.

Dans ce cas, on a E1 = PROPRIETAIRE et Rel = "=". PROPRIETAIRE fait partie d'un identifiant de VOITURE et est incorporé à $Y_{VOITURE}$.

Le vecteur $Y_{VOITURE}$ identifie à présent le type d'entités VOITURE qui peut donc être introduit dans le vecteur d'identification de la règle. On a VI=(VOITURE).

La troisième règle de domaine est conforme à la règle 2.1.2. On a E1 = DATE-ORD et Rel = "=". DATE-ORD est incorporé au vecteur d'identification.

On obtient après l'analyse des trois conditions simples :

VI = (VOITURE,DATE-ORD). VI identifie le type d'entités initial ORDRE-REP. Cela entraîne

GAR $\xrightarrow{1}$ ORD

Exemple 2 : GAR=GARAGE(:NOM="XXXX")

OUV=OUVRIER (effectue:ORDRE-REP
((rep:VOITURE((:MARQUE="RENAULT") et
(:PROPRIETAIRE="LUC")))
et
(:DATE-ORD = DATEOUV GAR)))

Cette règle désigne les ouvriers qui ont effectué un ordre de réparation sur une voiture de marque RENAULT et de propriétaire LUC à lade d'ouverture du garage GAR.

Etape 1 : cette règle se décompose comme suit :

OUV=OUVRIER
(((effectue:ORDRE-REP(rep:VOITURE(:MARQUE="RENAULT"))) et
(effectue:ORDRE-REP(rep:VOITURE(:PROPRIETAIRE="LUC")))) et
(effectue:ORDRE-REP(:DATE-ORD=DATEOUV GAR)))

Etape 2 : seule la dernière condition d'association simple "(effectue:ORDRE-REP (:DATE-ORD=DATEOUV GAR)" contient une variable entité. Elle est conforme à la règle 2.1.4. Nous obtenons la relation suivante :

GAR ⟶ OUV

Etape 3 : /

Etape 4 :

Après l'analyse de la première condition d'association simple, VI est vide et on a créé $Y_{VOITURE}$ = (MARQUE)

Suite à l'analyse de la seconde condition d'association, VI reste vide et $Y_{VOITURE}$ devient (MARQUE,PROPRIETAIRE).

VOITURE peut dès lors être introduit dans VI.

La troisième condition est conforme à la règle 2.1.4. On crée un vecteur $Y_{ORDRE-REP}$. On y introduit DATE-ORD, mais aussi VOITURE qui fait partie de VI.

$Y_{ORDRE-REP}$ (DATE-ORD, VOITURE) identifie ORDRE-REP qui est incorporé au vecteur VI.

VI = (ORDRE-REP, VOITURE) ne constitue pas un identifiant de OUVRIER.

Cela entraîne

GAR $\overset{N}{\longrightarrow}$ OUV

Exemple 3 : (inspiré de l'étude de cas n°2)

Supposons que le prix des produits par région varie tous les mois. Nous ajoutons un attribut MOIS à VENTEREG et un attribut DATE à COMMANDE pour pouvoir calculer le prix de celle-ci.

On a les règles de domaine suivantes et on veut établir la classe fonctionnelle de la dernière de ces règles.

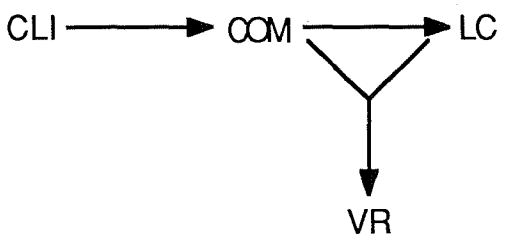
CLI=CLIENT(:NOM='YYYY')

COM=COMMANDE(cc:CLI)

LC=LIGNECOM(lc:COM)

VR=VENTEREG((vp:PRODUIT(pl:LC))et(:MOIS=Fmois($DATE_{COM}$)))

On obtient la hiérarchie suivante :

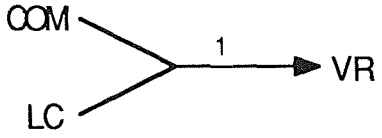


La règle de domaine qui définit VR est déjà décomposée.

La première condition d'association respecte la règle 1.1. On introduit dans VI le type d'entités de LC et celui de tous ses ancêtres. VI devient (LIGNECOM,COMMANDE,CLIENT).

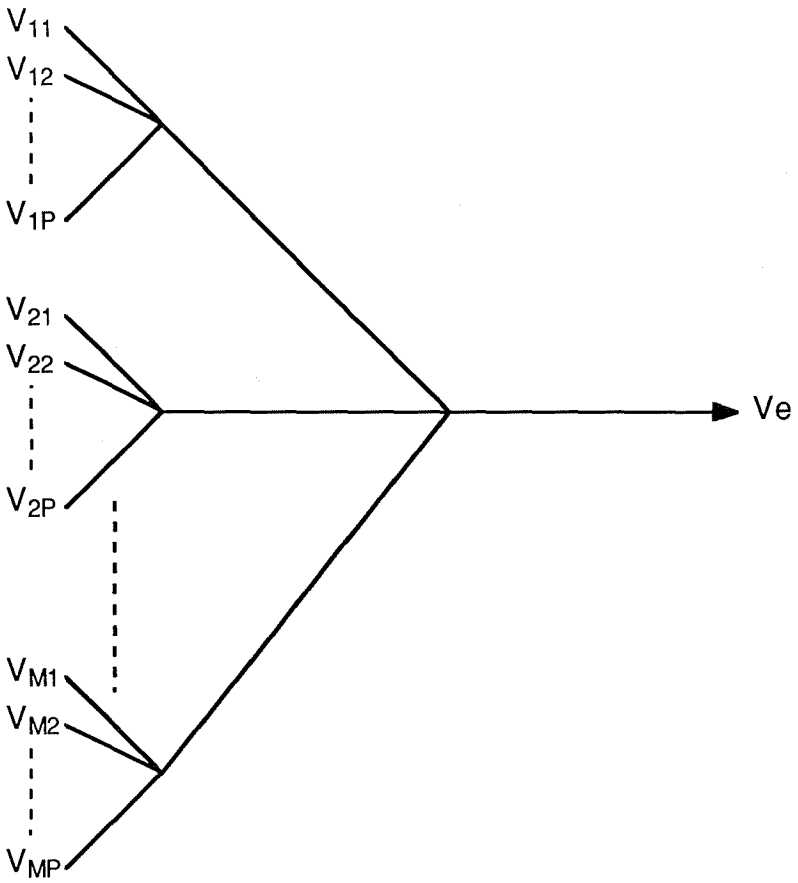
La seconde condition est conforme à la règle 2.1.2. et l'attribut MOIS est introduit dans VI.

VI=(LIGNECOM,COMMANDE,CLIENT,MOIS) identifie VENTEREG.
En effet, ID(VENTEREG)=(REGION, PRODUIT, MOIS) et
les chemins (CLIENT,REGION) (LIGNECOM,PRODUIT) sont de classe fonctionnelle N-1. Cela entraîne



Classe fonctionnelle pour une condition de sélection où apparaît des opérateurs logiques "ou".

A la suite de l'étape trois, nous avons recomposé la dépendance de la variable Ve vis-à-vis de ses paramètres. Nous obtenons un schéma du type :

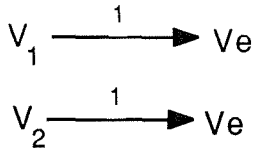


Par le point précédent, nous sommes capable d'étiqueter les relations :
$(V_{i1} , V_{i2} , ..., V_{iP}) \longrightarrow Ve$

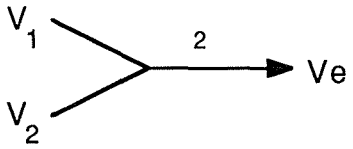
Il ne reste plus qu'à identifier la classe fonctionnelle de la dépendance finale. Comme les ensembles V_i sont des paramètres liés par l'opérateur "ou" , on peut affirmer que la dépendance finale sera "égale" à la **somme des dépendances** déjà établies.

Supposons une variable entité Ve paramètrée selon deux variables entité V_1 et V_2 qui appartiennent à des conditions de sélection (respectivement C_1 et C_2) reliées par un opérateur "ou". Le domaine de valeurs de Ve correspond à l'union de deux domaines, celui déterminé par la première condition de sélection C_1 et celui déterminé par C_2. Son cardinal sera donc égal à la somme du nombre des éléments des deux domaines. Donc, **la classe fonctionnelle sera toujours égale à N** pour une règle de domaine où apparaîssent des opérateurs "ou".

Cependant, il est possible de fournir une information supplémentaire à l'utilisateur dans certains cas. Reprenons la variable entité Ve et ses deux paramètres liés par un opérateur "ou" V_1 et V_2.
Supposons également que l'on a établi pour les deux conditions de sélection C_1 et C_2 les dépendances suivantes :



Le domaine de valeurs de Ve correspond à l'union du domaine désigné par C_1 et celui désigné par C_2. Or, par la classe fonctionnelle des deux relations ci-dessus, ces deux domaines contiendront au plus une valeur pour Ve. On obtient donc :

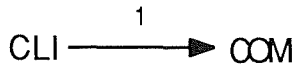


Ainsi, de manière générale, si une des relations pour les conditions de sélection à opérateurs "et" est de classe N, alors la dépendance finale est de classe N.

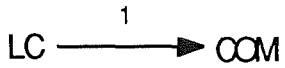
Si toutes les relations inférieures sont de classe 1, alors la dépendance finale est de classe M, c'est-à-dire le nombre de conditions de sélection ne comportant que des opérateurs logiques "et". Dans ce cas, le domaine de la variable entité définie contiendra au plus M éléments. Ce M constitue une information intéressante pour l'utilisateur. Cependant, au niveau de la classe fonctionnelle, cette valeur M sera assimiliée à une relation N.

Exemple : COM=COMMANDE((CL:LC) ou (:NCOM=NCLI CLI))

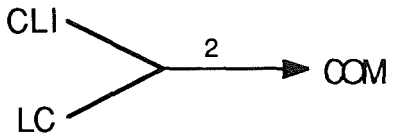
L'analyse de la première condition d'association a révélé la dépendance suivante :



L'analyse de la seconde condition de sélection donne la dépendance :



En suivant la règle de pour les opérateurs "ou" que nous venons de voir, on obtient :



Pour corroborer nos dires, déplions une grandeur dimensionnée par COM (Grd). Supposons également que les valeurs possibles pour CLI sont C1 et C2 et pour LC, LC1 et LC2.

$$Grd_{COM} = \left| \begin{array}{l} Grd_{COMMANDE((CL:LC1)\ ou\ (:NCOM=NCLI_{C1}))} \\ Grd_{COMMANDE((CL:LC1)\ ou\ (:NCOM=NCLI_{C2}))} \\ Grd_{COMMANDE((CL:LC2)\ ou\ (:NCOM=NCLI_{C1}))} \\ Grd_{COMMANDE((CL:LC2)\ ou\ (:NCOM=NCLI_{C2}))} \end{array} \right|$$

où

$$Grd_{COMMANDE((CL:LC2)\ ou\ (:NCOM=NCLI_{C2}))} = \left| \begin{array}{l} Grd_{CO1,LC2} \\ Grd_{CO2,C2} \end{array} \right|$$

où CO1 est la commande qui correspond à LC2 par la première condition de sélection et CO2 est la commande qui correspond au client C2 par la seconde condition de sélection.

II.3.1.5. Développement d'un exemple complet

Analysons l'ensemble des règles de domaine de l'étude de cas n°1

1. REV = REVUE(:TITRE=NOMR)

- *Première étape* : la condition de sélection se limite à une condition d'association simple.

- *Deuxième étape* :
  Règle : (règle 2.1.1.) : une condition d'appartenance apparaît dans la condition d'association simple.
  Cap = "TITRE=NOMR" et
  Rel = "="et
  E1 = TITRE et
  E2 = NOMR est une grandeur non indicée par une Ve
  cela entraîne que REV est indépendante.

- *Troisième étape* : /

- *Quatrième étape* : /

2. ABO = ABONNEMENT((aa:ANNEE(:AN ≥ A-D)) et (ar:REV))

- *Première étape* : la condition de sélection se décompose en deux conditions d'association simples.
  Cas1 = (aa:ANNEE(:AN ≥ A-D))
  Cas2 = (ar:REV)
  Csel = (Cas1 et Cas2)

- *Deuxième étape* :
  1. Analyse de Cas1
     Règle : (règle 2.1.3.)
     Une Cap "AN ≥ A-D" apparaît dans Cas1 et
     Rel = "≥" et
     E1 = AN
     E2 = A-D
     cela entraîne que ABO est indépendante.
  2. Analyse de Cas2
     Règle : (règle 1.1.)
     Aucune Cap n'apparaît dans Cas2 et
     le dernier ensemble spécifié dans Cas2 est la variable entité REV
     cela entraîne

REV ⟶ ABO

- *Troisième étape* : /

- *Quatrième étape* :
  VI = (REVUE)
  VI n'identifie pas ABONNEMENT, cela entraîne

REV —N→ ABO

3. P = PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV))) et (:DATEDEB ≥ A-D))

- *Première étape* :
  Cas1 = (concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV)))
  Cas2 = (:DATEDEB ≥ A-D)
  Csel = (Cas1 et Cas2)

- *Deuxième étape* :
  1. Analyse de Cas1
     Règle : (règle 1.1.)
     Aucune Cap n'apparaît dans Cas1 et le dernier élément spécifié dans Cas1 est la variable entité REV
     cela entraîne

     REV ⟶ P

  2. Analyse de Cas2
     Règle : (règle 2.1.1.)
     Une Cap "DATEDEB ≥ A-D" apparaît dans Cas1 et
     Rel = "≥" et
     E1 = DATEDEB
     E2 = A-D
     cela entraîne que P est indépendante.

- *Troisième étape* : /

- *Quatrième étape* :
  VI = (REVUE)
  VI n'identifie pas PRET, cela entraîne

  REV $\xrightarrow{N}$ P

4. A = ANNEE(:AN=Fannée(DATEDEB $_P$))

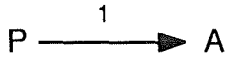
- *Première étape* : la Csel se limite à une Cas simple

- *Deuxième étape* :
  Règle : (règle 2.1.2)
  Une Cap "AN=Fannée(DATEDEB $_P$)" apparaît dans la Cas et
  Rel = "=" et
  E1 = AN et
  E2 = DATEDEB $_P$
  Cela entraîne

  P ⟶ A

- *Troisième étape* : /

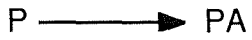
- *Quatrième étape* :
  VI = (PRET, AN, REVUE)
  VI identifie ANNEE, cela entraîne



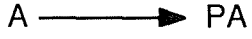
5. PA = ANNEEXEM((ea:EXEMPLAIRE(concerne:P)) et (aae:A))

- *Première étape* :
  Cas1 = (ea:EXEMPLAIRE(concerne:P))
  Cas2 = (aae:A)
  Csel = (Cas1 et Cas2)

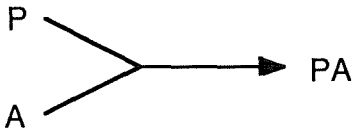
- *Deuxième étape* :
  1. Analyse de Cas1
     Règle : (règle 1.1.)
     Aucune Cap n'apparaît dans Cas1 et le dernier ensemble spécifié dans Cas1 est la variable entité P
     cela entraîne



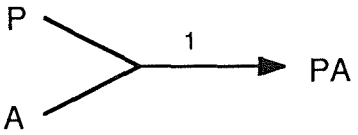
  2. Analyse de Cas2
     Règle : (règle 1.1.)
     Aucune Cap n'apparaît dans Cas2 et le dernier ensemble spécifié dans Cas2 est la variable entité A
     cela entraîne



- *Troisième étape* :
  Cas1, Cas2 sont reliées par l'opérateur "et", cela entraîne que PA dépend conjointement de P et de A :



- *Quatrième étape* :
  VI = (PRET, ANNEE, REVUE)
  VI identifie ANNEEXEM, car le couple (EXEMPLAIRE,ANNEE) est le "véritable identifiant" et le chemin (PRET,EXEMPLAIRE) est de classe fonctionelle N-1.
  Cela entraîne



6. ABON = ABONNE(beneficie:P)

- *Première étape* : la Csel se limite à une Cas simple.

- *Deuxième étape* :
  Règle : (règle 1.1.)

Aucune Cap n'apparaît dans le Csel et le dernier ensemble spécifié dans le Csel est la variable entité P
cela entraîne

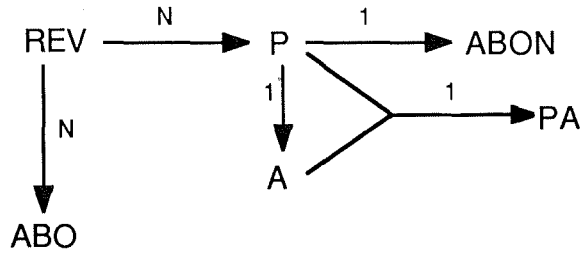
P ⟶ ABON

- *Troisième étape* : /

- *Quatrième étape* :
VI = (PRET, REVUE)
VI identifie ABONNE, cela entraîne

P —1→ ABON

A la suite de cette analyse, nous pouvons composer ces diverses relations pour creer le GDV du modèle du calcul de la rentabilité d'une revue.



II.3.2. Cohérence au niveau d'une règle

Dans une règle peuvent intervenir des grandeurs dimensionnées et des grandeurs simples. En ce qui concerne les grandeurs dimensionnées, un premier type de validation peut être mis en place pour **contrôler l'utilisation des dimensions**. Des rapports existent entre la ou les dimensions d'une grandeur expliquée par une règle et la ou les dimensions apparaissant en partie droite de celle-ci. Il faut veiller à l'absence de contradiction entre chaque dimension de la grandeur expliquée par une règle et chacune des dimensions apparaissant en partie droite de celle-ci.

Une règle a la forme générale suivante :

$$Grandeur2_{Dim2} = h\ (Grandeur1_{Dim1})$$

avec : - h est une expression contenant la grandeur $Grandeur1_{DIM1}$ (plusieurs grandeurs dimensionnées peuvent intervenir dans la partie droite de la règle, cela ne change pas l'idée générale).
- Dim1 est l'ensemble des dimensions de Grandeur1 qui est
  soit une grandeur mixte
  soit une grandeur dimensionnée par une variable entité
  soit une grandeur dimensionnée par une ou plusieurs variables simples.
- Dim2 est l'ensemble des dimensions de Grandeur2 qui est
  soit une grandeur mixte
  soit une grandeur dimensionnée par une variable entité
  soit une grandeur dimensionnée par une ou plusieurs variables simples.

Nous allons analyser les rapports entre chaque dimension de Dim2 et chaque dimension de Dim1 en les examinant deux à deux. Le point II.3.2.1. classifiera les relations entre variables entité et relèvera un certains nombre de règles de cohérence au moyen du graphe de dépendances des variables entité établi au point précédent. Le point II.3.2.2. fera de même avec les variables simples. Pour ce qui est de l'étude des liens entre variables simples et variables entité, elle n'est d'aucun intérêt puisqu'il n'existe aucun rapport entre ces deux types de dimensions.

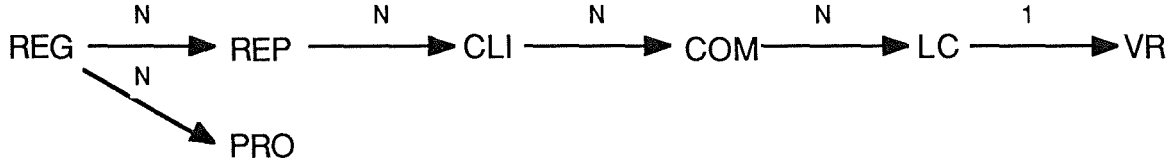
Dans le point II.3.2.3., nous regarderons plus en détail le cas des grandeurs simples.

Après l'analyse de la cohérence au niveau des dimensions, nous nous occuperons de la **compatibilité** des types (entier, booléen, ...) des grandeurs (point II.3.2.4.).

Pour clôturer, nous étudierons la cohérence des règles à définitions multiples (point II.3.2.5.) et des règles de récurrence (point II.3.2.6.). Ces règles nécessitent une étude plus approfondie que celle faite au niveau des dimensions et des types.

II.3.2.1. Les grandeurs dimensionnées par une variable entité

Le graphe de dépendances entre variables entité va nous permettre de contrôler l'usage des variables entité en tant qu'indice de grandeurs. Afin de mieux saisir les mécanismes proposés, nous présenterons des exemples inspirés de la deuxième étude de cas dont voici le graphe de dépendances :



Deux types de grandeurs sont dimensionnées par une variable entité : les grandeurs dimensionnées par une variable entité et les grandeurs mixtes.

Toute grandeur interne ou résultat dimensionné par une variable entité est expliqué par une règle dans laquelle apparaît une ou des autre(s) grandeur(s) dimensionnée(s) par une variable entité. Au vu des relations entre variables entité établies dans le graphe de dépendances, certains principes sont à respecter pour garder un modèle cohérent.

Comme cela a déjà été dit précédemment, nous examinerons les rapports entre la variable entité de la grandeur expliquée par une règle (soit Ve2) avec celle(s) de la partie droite de cette règle (soit Ve1). Suivant la position des deux variables entité dans le graphe, nous distinguerons quatre situations :

- Ve2 est ancêtre de Ve1,
- Ve2 est descendant de Ve1,
- Ve2 est égale à Ve1,
- pas de chemin entre Ve1 et Ve2, mais un ancêtre commun.

Pour chacun des cas, des mécanismes de vérification de cohérence peuvent être trouvés.

Ve2 est l'ancêtre de Ve1

Ve1 est donc paramètrée selon Ve2. Les différents cas que nous allons distinguer dans cette rubrique sont basés sur la classe fonctionnelle de la dépendance qui existe entre ces deux variables. Celle-ci est égale au poids du chemin reliant Ve2 à Ve1 : il peut être de poids 1, N ou égal à un puissance de N supérieure à un.

Si, entre Ve1 et Ve2, il existe un **chemin de poids N**, à une occurrence de Ve2 correspond plusieurs occurrences de Ve1. Dans ce contexte, on ne peut "égaler" Grandeur1 $_{Dim1}$ à Grandeur2 $_{Dim2}$. Par conséquent, Grandeur1 $_{Dim1}$ doit obligatoirement être dans une fonction d'agrégation dont l'indice d'agrégation est soit Ve1, soit un ancêtre de Ve1 qu'il lui est relié par un chemin de poids 1. La dimension d'agrégation a le même cardinal que le domaine de valeurs de Ve1.

Exemples : - (Etude de cas n°2)

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LC}_{LC}$$

Le montant d'une commande est égal à la somme du montant des lignes de celle-ci. A une commande COM correspond plusieurs lignes de commande LC ; il faut donc sommer. La fonction Σ et la grandeur MONTANT-LC sont indicées par la même variable entité (LC). La dimension COM est l'ancêtre de LC ; il y a bien un chemin de poids N entre ces deux dimensions.

- (figure 2.5. : exemple de grandeurs mixtes)

$$\text{MONTANT-REV}_{AN,REV} = \sum_{N} \text{MONTANT-NUM}_{AN,N} + \text{AMENDE}_{AN,REV}$$

La grandeur expliquée par cette règle est mixte. La dimension N est paramètrée selon REV par un chemin de poids N. Il est donc logique de retrouver la grandeur MONTANT-NUM $_{AN,N}$ dans une fonction d'agrégation portant N en indice.

- (Etude de cas n°2)

$$\text{MONTANT-COM}_{COM} = \sum_{LC} (\text{PRIX}_{VR} * \text{QUANTITE}_{LC})$$

On obtient cette règle en remplaçant dans le premier exemple MONTANT-LC $_{LC}$ par sa règle de définition. La fonction Σ et PRIX sont indicées par deux variables entité reliées par un chemin de poids 1 (LC et VR). A une vente de région VR correspond une ligne de commande LC. On peut donc sommer car une quantité a son prix, qui est unique dans la région considérée. Les domaines de VR et de LC ont le même cardinal. La dimension COM est ancêtre de LC et VR. Le chemin allant de COM à VR et de COM à LC est de poids N.

Si, entre les deux dimensions, il y a un **chemin de poids 1**, à une occurrence de Ve2 correspond une occurrence de Ve1. Grandeur2 $_{Dim2}$ et Grandeur1 $_{Dim1}$ seront égalées car Ve1 et Ve2 ont le même nombre d'éléments. On ne peut pas utiliser une fonction d'agrégation dont l'indice porte sur Grandeur1$_{Dim1}$.

Exemple : (Etude de cas n°2)

$$\text{MONTANT-LC}_{LC} = \text{PRIX}_{VR} * \text{QUANTITE}_{LC}$$

Le montant d'une ligne de commande est égal à la multiplication de la quantité commandée d'un produit par son prix dans la région considérée. La dimension LC est l'ancêtre de VR et le chemin les reliant est de poids 1.

Si, entre les deux dimensions, il y a un **chemin de poids égal à une puissance de N supérieur à un**, elles doivent être reliées à l'aide de plusieurs fonctions d'agrégation[1]. Ce genre de règles est mathématiquement correcte mais son côté pratique est discutable. Pour éviter de créer des règles trop complexes pouvant entraîner des erreurs de conception, il faut décomposer la règle en utilisant d'autres grandeurs internes dimensionnées.

[1] Il s'agit simplement d'emboîter des fonctions d'agrégation.

Exemple : (Etude de cas n°2)
Dans ce modèle, on n'écrit pas :

$$\text{MONTANT-CLI}_{CLI} = \sum_{COM} \sum_{LC} \text{MONTANT-LC}_{LC}$$

Théoriquement, cette règle est bonne. A un client CLI correspond plusieurs commandes COM qui comprennent chacune plusieurs lignes de commande LC, d'où la composition de deux fonctions d'agrégation. Toutefois, pour des raisons pédagogiques, on décompose en créant une grandeur interne MONTANT-COM_{COM} pour rendre la règle acceptable dans notre modèle :

$$\text{MONTANT-CLI}_{CLI} = \sum_{COM} \text{MONTANT-COM}_{COM}$$

$$\text{MONTANT-COM}_{COM} = \sum_{LC} \text{MONTANT-LC}_{LC}$$

Suite à cette restriction, si Grandeur1_{Dim1} participe à une agrégation portant Ve1 en indice, Ve2 et Ve1 doivent être obligatoirement reliées par un chemin de poids N.

Ve2 est descendant de Ve1

Les dépendances du graphe de dépendances entre variables entité ont été qualifiées de fonctionnelles (cfr point II.2.1.3.). Dans ce cas, à une occurrence de Ve2 ne correspond qu'une seule occurrence de Ve1. Nous nous retrouvons dans la même situation que lorsque Ve2 est ancêtre de Ve1 et qu'elles sont reliées par un chemin de poids 1. Grandeur1_{Dim1} peut donc apparaître ou non dans une fonction d'agrégation. Si elle appartient à une agrégation, l'indice de celle-ci ne peut ni être égal à Ve1 ni être un ancêtre relié à Ve1 par un chemin de poids 1.
Exemples : (Etude de cas n°2)

- $\text{NOUVQUOTA}_{REP} = (1+(\text{FACTEURPUB}_{REG} + \text{FACTEURECO}_{REG})/2) * \text{BASECALCUL}_{REP}$

  Des grandeurs dimensionnées par REG (ancêtre de REP) peuvent intervenir dans l'expression ci-dessus. Il n'est pas incohérent de dire que les nouveaux quotas sont calculés à partir de la publicité de la région du représentant.

- $\text{MONTANT-CLI}_{CLI} = \sum_{COM} (\text{MONTANT-COM}_{COM} + \text{FRAIS}_{REG})$

  On suppose que chaque région a ses propres frais de commande (FRAIS_{REG} attribut du type d'entités REGION). Il y a une seule valeur de FRAIS_{REG} qui intervient autant de fois qu'il y a de commandes pour le client CLI de la région REG.

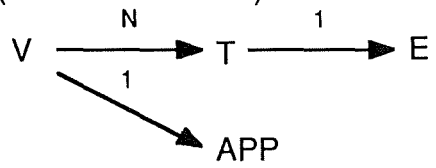
Ve2 est égale à Ve1

Il est évident que, par cette égalité, Grandeur1 $_{Dim1}$ peut apparaître ou non dans une fonction d'agrégation. Mais, en aucun cas, si elle appartient à une agrégation, l'indice de celle-ci ne peut être égal à Ve1 ou être un ancêtre relié à Ve1 par un chemin de poids 1. Il serait absurde de sommer sur Ve1 puisqu'à une occurrence de Ve2 ne lui correspond évidement qu'une seule occurrence de Ve1.
Exemples : (Etude de cas n°2)

- MONTANT-CLI $_{CLI} = \sum_{COM}$ (MONTANT-COM $_{COM}$ + FRAIS $_{CLI}$)
  Chaque client engendre des frais différents (FRAIS $_{CLI}$ attribut du type d'entités CLIENT). Dans ce cas-ci, les frais sont les mêmes pour toutes les commandes COM que le client CLI a passées. On ne somme pas sur CLI.
- BASE2 $_{REP}$ = QUOTA $_{REP}$ + SUPNOUVPRO $_{REG}$ /NBREP $_{REG}$
  La grandeur interne BASE2 $_{REP}$ est expliquée par le quota dimensionné par REP également.

Pas de chemin entre Ve1 et Ve2 mais un ancêtre commun Ve3

Dans ce cas, il faut remonter à l'ancêtre Ve3 dans le graphe de dépendances entre variables entité. A une occurrence de Ve2 ne correspond qu'une seule occurrence de Ve3 (le graphe définit des dépendances fonctionnelles). Par conséquent, la forme générale "Grandeur2 $_{Dim2}$ = h (Grandeur $_{Dim1}$)" peut s'écrire "Grandeur2 $_{Dim3}$ = h (Grandeur1 $_{Dim1}$)" avec Dim3 égale à Dim2 où on a remplacé Ve2 par Ve3. Remarquons qu'au niveau de la vérification de cohérence basée sur le graphe de dépendances, cela ne change rien, bien qu'au niveau sémantique la règle a un tout autre sens. Mais cela importe peu dans le cas présent. On applique les trois cas analysés précédemment pour vérifier cette nouvelle règle.
Exemple : (Etude de cas n°3)



avec Ve1 = APP
Ve2 = T
Ve3 = V
La règle : QCONS/km $_T$ = CONS-VIDE $_{APP}$ + CONS-CHARGE $_{APP}$ * CHARGE-MOY $_V$
APP et T ne sont pas reliées par un chemin mais elles ont un ancêtre commun V. On peut transformer la règle comme suit :
QCONS/km $_V$ = CONS-VIDE $_{APP}$ + CONS-CHARGE $_{APP}$ * CHARGE-MOY $_T$
D'après ce qu'on a dit ci-dessus, on ne peut sommer entre V et APP (car il y a un chemin de poids 1 entre ces deux dimensions). Donc, on ne peut voir une grandeur dimensionnée par APP dans une fonction d'agrégation la portant en indice (cfr Ve2 est ancêtre de Ve1).

II.3.2.2. Les grandeurs dimensionnées par des variables simples

Ce point concerne les grandeurs dimensionnées par une ou plusieurs variables simples ainsi que les grandeurs mixtes. Les variables simples ne figurent nullement dans le graphe de dépendance entre variables entité. Il n'existe pas de liens sémantiques entre variables simples comme pour les variables entité. Nous ne disposons donc d'aucun support permettant de déduire des principes de vérification.

Néanmoins, les deux mécanisme présentés ci-dessous sont obligatoires pour assurer la cohérence d'une règle. Ils sont dictés par le bon sens.

Si l'on agrège sur une des variables simples dimensionnant Grandeur1 $_{Dim1}$, celle-ci ne se retrouvera plus comme dimension de Grandeur2 $_{Dim2}$.

Exemples : - (figure 2.4.)

$$\text{MONTANT-COM}_{CO} = \sum_{PR} \text{MONTANT-LC}_{CO,PR} * \text{FRAIS}_{CO}$$

La fonction d'agrégation porte sur PR. Cette variable simple n'est plus dimension de MONTANT-COM $_{CO}$. A une commande CO correspond plusieurs produits PR.

- (figure 2.5.)

$$\text{COUT-PAIEMENT}_{REV} = \sum_{AN} \text{MONTANT-REV}_{AN,REV}$$

Le coût d'une revue sur plusieurs années est égal à la somme des montants annuels de cette revue. On fait la Σ sur AN ; elle disparaît des indices en partie gauche.

Si les variables simples de Grandeur1 $_{Dim1}$ ne sont pas les indices sur lesquels portent les fonctions d'agrégation, elles se retrouvent comme dimensions de Grandeur2 $_{Dim2}$.

Exemples : - (figure 2.4.)

$$\text{MONTANT-LC}_{CO,PR} = Q_{CO,PR} * \text{PRIX}_{PR}$$

MONTANT-LC est égal à la quantité du produit fois le prix de celui-ci. En partie gauche de cette règle se trouvent toutes les variables simples de l'expression puisqu'elles ne font l'objet d'aucune agrégation.

- (figure 2.5.)

$$\text{MONTANT-REV}_{AN,REV} = \sum_{N} \text{MONTANT-NUM}_{AN,N} + \text{AMENDE}_{AN,REV}$$

la variable AN est dimension de MONTANT-REV. Elle n'est pas l'indice d'une fonction d'agrégation.

II.3.2.3. Les grandeurs simples

Nous distinguons deux cas : lorsqu'une grandeur simple apparaît en partie droite d'une règle de définition et lorsqu'elle est expliquée par une règle.

Une grandeur simple peut apparaître **en partie droite d'une règle**. Comme elle n'est pas dimensionnée, aucune vérification n'est possible. Son emploi dans n'importe quel type d'expressions ne rend pas la règle incohérente.

Exemple : (Etude de cas n°1)

DUREE-DEPASS $_P$ = DATERET $_P$ - DATEDEB $_P$ - DUREE

Le dépassement (c'est-à-dire le retard lorsqu'on ne rend pas un prêt à temps) est égal à la soustraction de la durée du prêt (date de retour moins date de prêt) et de la durée maximale d'un prêt. Une grandeur simple (DUREE) intervient dans la règle de définition d'une grandeur dimensionnée par une variable entité.

La règle de définition d'une grandeur simple peut contenir des grandeurs dimensionnées sur lesquelles on effectue des agrégations.

Exemples : - (figure 2.4.)

Si on souhaite calculer le chiffre d'affaire (CA) comme la somme de toutes les commandes passées, on obtient :

$$CA = \sum_{CO} MONTANT\text{-}COM_{CO}$$

La variable simple n'a plus de raison de dimensionner CA puisqu'on a regroupé toutes les commandes en les sommant.

- (figure 2.3.)

Si on reprend le modèle calculant le montant de la facture d'un client, en prenant pour CLI tous les clients, on obtient :

CLI = CLIENT

$$CA = \sum_{CLI} MONTANT\text{-}CLI_{CLI}$$

La variable entité a disparu. CLI n'a pas d'ancêtre dans le graphe de dépendances entre variables entité.

En général, on peut affirmer qu'une grandeur simple peut être expliquée par une règle où apparaissent des grandeurs dimensionnées par une variable entité ou par une variable simple. Ces grandeurs sont dans une fonction d'agrégation dont l'indice est égal à leur dimension. En effet, il faut que ces dimensions ne soient pas en partie gauche de la règle. La fonction d'agrégation permet de les faire disparaître.

Pour être complet, nous devons ajouter que, dans le cas d'une dimension variable entité, celle-ci n'a pas d'ancêtre. Si elle avait un ancêtre, il devrait se trouver comme dimension de la grandeur expliquée, ce qui est impossible.

II.3.2.4. Cohérence des types

Une vérification simple à effectuer et qui n'a pas encore été envisagée est la cohérence des types dans une règle (qu'elle soit à définition unique, à définitions multiples, récursive ou récurrente).

Dans un premier temps, les types des grandeurs employés dans la partie à droite du signe d'égalité d'une règle doivent être compatibles. La **compatibilité de types** est basée sur l'affirmation suivante : "les opérations arithmétiques (+,-,*,/,...) ne peuvent être appliquées qu'à des grandeurs de type numérique (réel, entier, ...)."

Exemple : (Etude de cas n°1)

La grandeur MONTANT-PRET $_P$ n'est pas définie par la règle :

MONTANT-PRET $_P$ = MONTANT $_{PA}$ * RED $_{ABON}$

On ne peut multiplier une valeur logique avec une valeur numérique. On a donc défini une grandeur interne REDUCTION $_P$ de type réel assurant la cohérence de la règle.

Deuxièmement, les types de la partie droite d'une règle doivent être compatibles avec celui de la partie gauche. Dans notre exemple, on peut affirmer que si la partie droite est une multiplication de grandeurs de type réel, la partie gauche (MONTANT-PRET $_P$) sera aussi de type réel.

Pour conclure, nous pouvons faire deux remarques :

- la vérification de la compatibilité des types dans les conditions d'une règle à définitions multiples est semblable au cheminement proposé.
- le but principal de notre étude de cohérence est d'avertir l'utilisateur de la présence éventuelle d'erreurs. Ici, en aucun cas, il ne s'agit d'avertissement. Les principes donnés sont des conditions indispensables pour assurer la cohérence sémantique d'un modèle.

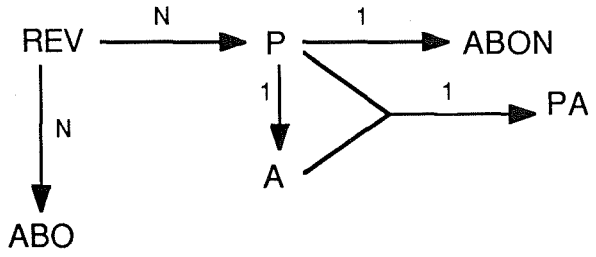
II.3.2.5. Cohérence des règles à définitions multiples

Quelle sorte de mécanismes de cohérence doit-on envisager face à des règles à définitions multiples ? Prenons la règle suivante de la première étude de cas :

AMENDE-PRET $_P$ = 0 si DEPASS-DATE $_P$ = faux
DUREE-DEPASS $_P$ * AMENDE $_A$ si DEPASS-DATE $_P$ = vrai

Il apparaît clairement que cette règle utilise des grandeurs dimensionnées. Les mécanismes de vérification de cohérence des trois points précédents concernant les grandeurs et leurs dimensions sont applicables à ces règles.

Plus particulièrement dans l'exemple, les deux conditions ne posent pas de problème : elles ne contiennent qu'une seule grandeur dimensionnée par une variable entité (DEPASS-DATE $_P$) qui est un booléen. Où il faut faire attention, c'est dans la deuxième expression définissant AMENDE-PRET $_P$: DUREE-DEPASS $_P$ * AMENDE $_A$. Il s'agit de grandeurs dimensionnées par une variable entité. Pour effectuer un contrôle rigoureux, nous avons besoin du graphe de dépendances entre variables entité :



Deux vérifications sont à faire :

- entre AMENDE-PRET $_P$ et DURE-DEPASS $_P$: il s'agit des deux mêmes variables entité. Dans le point II.3.2.1., il est dit qu'en aucun cas DUREE-DEPASS $_P$ ne peut figurer dans une fonction d'agrégation portant P en indice. Ici, il n'y a pas de problème.
- entre AMENDE-PRET $_P$ et AMENDE $_A$: P est ancêtre de A et le chemin les reliant est de poids 1. Donc, AMENDE $_A$ ne peut être utilisée dans une fonction d'agrégation portant A en indice car à une occurrence de P correspond une occurrence de A.

Lorsqu'on peut affirmer qu'une règle à définitions multiples est cohérente du point de vue de ses dimensions et de ses types, il faut encore vérifier la **couverture** et le **non-recouvrement**.

On parlera de problème de couverture lorsqu'une grandeur expliquée par une règle à définitions multiples ne reçoit pas toujours une valeur parce qu'aucune condition n'a été vérifiée.

Le non-recouvrement contrôle que plusieurs conditions ne peuvent être vérifiées en même temps afin que la grandeur expliquée ne prenne qu'une seule valeur.

Lorsqu'il y a un problème de couverture, on avertit l'utilisateur mais on ne l'oblige pas à corriger sa règle. La grandeur aura une valeur inconnue se propageant à travers le modèle. Par contre, le critère de non-recouvrement est indispensable pour assurer la cohérence d'un modèle.

Avant d'analyser de manière détaillée chacun de ces deux critères, il faut poser les limites de notre étude de cohérence.

II.3.2.5.1. Limites de l'étude de cohérence

L'étude de cohérence développée, pour le contrôle de couverture et de non-recouvrement des règles à définitions multiples, a un champ d'application restreint. Parfois, **la forme des conditions** dans de telles règles et **les corrélations entre grandeurs** ne permettent pas toujours d'assurer la mise en oeuvre de la vérification de cohérence. Le principe sur lequel est basée la vérification de couverture et de non-recouvrement restreint encore le champ d'application de notre analyse. Nous avons repéré trois types de limites.

Premièrement, lorsqu'on essaye d'établir si la grandeur en partie gauche de la règle est bien expliquée par une règle de définition, il n'est pas toujours possible de déterminer avec précision dans quel contexte la condition sera vérifiée. Prenons un exemple pour montrer le problème.

Exemple : A,B,C sont des grandeurs.

On a la règle à définitions multiples : A = 10 si B ≤ C
20 si B > C

Dans cette règle, on ne peut savoir si A recevra toujours une valeur.

La difficulté réside dans le fait que la valeur de C ne sera connue qu'à l'exécution. Or, notre étude de cohérence intervient au moment de la conception d'un modèle. On ne peut utiliser les principes de vérification lorsque, dans une condition, il existe une comparaison entre grandeurs (différentes de constantes).

Deuxièmement, il est possible de rencontrer, dans des conditions, des grandeurs corrélées. Suivant la valeur prise par une des grandeurs, l'autre verra son domaine de valeurs limité.

Exemple : A,B,C sont des grandeurs. B et C sont corrélées car, lorsque C est supérieure à zéro, B est obligatoirement supérieure à 0 (le contraire étant également vrai). On a la règle :

A = 10 si B ≤ 0
20 si B > 0 et C > 0

Si B a la valeur 3 et C la valeur -2, aucune condition n'est vérifiée. On ne couvre pas tous les cas.

Cette règle semble donc incohérente au niveau de la couverture. Or, d'après la corrélation entre B et C, on peut dire qu'il est impossible d'avoir un jeu de valeurs où B est positif et C négatif. La règle est donc cohérente au niveau de la couverture. La corrélation entre grandeurs peut fausser les résultats de l'étude de cohérence.

Troisièmement, pour satisfaire notre but pédagogique et pour fournir un contrôle de cohérence le plus efficace possible, deux restrictions sont apportées aux règles à définitions multiples :

- dans une condition, on n'autorise qu'un seul opérateur logique (ou/et).
- dans l'ensemble des conditions d'une règle à définitions multiples, seulement deux grandeurs distinctes pourront intervenir.

Si ces deux restrictions ne sont pas respectées, cela ne veut pas dire pour autant que le modèle sera faux, mais il sera complexe et la vérification de couverture et de non-recouvrement que nous proposons sera impossible.

Dès à présent, montrons par un exemple comment l'utilisateur a la possibilité de contourner ces limites. Dans la plupart des cas, elles ne sont pas gênantes car il est possible de définir des grandeurs internes intermédiaires rassemblant des grandeurs utilisées dans des conditions complexes.

Exemple : Transformons la définition de la grandeur ECONOMIE $_{REG}$ dans l'étude de cas n°2 :

ECONOMIE $_{REG}$ = bonne si CROISSANCE $_{REG} \geq 0,04$ ou
CHOMAGE $_{REG} < 0,076$
= moyenne si FACTEURPUB $_{REG} < 0,005$
= mauvaise si (CROISSANCE $_{REG} < 0,04$ ou
CHOMAGE $_{REG} \geq 0,076$) et
FACTEURPUB $_{REG} \geq 0,005$

Cette règle n'est pas autorisée pour deux raisons :

- premièrement, la deuxième condition introduit une troisième grandeur différente des deux premières.
- deuxièmement, la troisième condition utilise deux opérateurs logiques.

Modifions l'exemple ci-dessus pour le rendre acceptable. Pour ce faire, la grandeur interne CROISCHO $_{REG}$ est définie :

CROISCHO $_{REG}$ = vrai si CROISSANCE $_{REG} \geq 0,04$ ou
CHOMAGE $_{REG} < 0,07$
= faux si CROISSANCE $_{REG} < 0,04$ ou
CHOMAGE $_{REG} \geq 0,076$

Cela donne la règle correcte suivante pour ECONOMIE $_{REG}$ en utilisant CROISCHO $_{REG}$:

ECONOMIE $_{REG}$ = bonne si CROISCHO $_{REG}$ = vrai
= moyenne si FACTEURPUB $_{REG} < 0,005$
= mauvaise si FACTEURPUB $_{REG} \geq 0,005$ et
CROISCHO $_{REG}$ = faux

Cette règle ne viole plus les restrictions de notre modèle.

II.3.2.5.2. Couverture de tous les cas

Une grandeur définie par une règle à définitions multiples prend des valeurs différentes suivant la condition vérifiée. Les grandeurs intervenant dans les conditions définissent un ensemble de cas à envisager. Un contrôle de cohérence peut être effectué pour examiner si tous les cas possibles sont bien pris en compte.

Le but de cette étude est de fournir un mécanisme rigoureux de vérification de la couverture de tous les cas. Prenons un exemple pour bien visualiser le problème.

Exemple : B = 10 si A > 100
20 si A < 0

La grandeur B aura une valeur pour toutes les valeurs de A strictement supérieures à 100 et strictement inférieures à 0. Mais, sachant que A peut prendre n'importe quelle valeur entière, que se passe-t-il si elle prend sa valeur dans l'intervalle [0,100] ? B sera alors indéfini. Le contrôle détecte ce genre de problème.

Tout d'abord, mettons de côté le cas trivial où l'on a dans les conditions d'une règle à définitions multiples une condition par défaut (emploi du SINON). Automatiquement, cette clause assure la couverture de tous les cas.

Exemple : (Etude de cas n°2)

FACTEURPUB $_{REG}$ = PUB $_{REG}$ /100000 si PUB $_{REG} > 7000$ et
ECONOMIE $_{REG}$ <> mauvaise
-0.015 si PUB $_{REG} < 5000$ et
ECONOMIE $_{REG}$ <> bonne
0 sinon

Idée générale

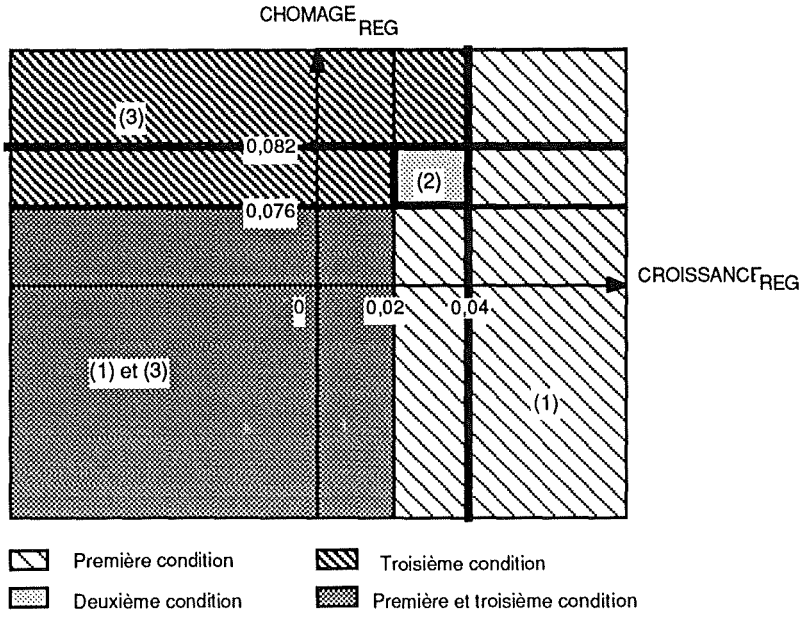
La vérification de couverture est basée sur la technique des plans dans un espace à deux dimensions. L'idée est assez simple : à chaque grandeur intervenant dans les conditions d'une règle, on assigne un axe de l'espace. Puisque nous travaillons dans un espace à deux dimensions, on comprend aisément la restriction qui consiste à n'accepter que deux grandeurs dans les conditions. Les deux axes représentent l'ensemble des valeurs des deux grandeurs. Pour réaliser cette technique, il faut faire attention au type des grandeurs. S'il s'agit de réels ou d'entiers, cela ne pose aucun problème (la découpe des axes est habituelle). Par contre s'il s'agit de booléens ou de grandeurs prenant leur valeur dans un domaine du style (bon, moyen, mauvais), il faut prendre certaines conventions dans la découpe des axes. Par exemple, les nombres positifs représentent VRAI et les négatifs FAUX.

Au départ, le plan représente tous les cas à couvrir. La première condition définit une portion du plan qui est couverte. Celle-ci est marquée (ou retirée de l'espace restant à couvrir) pour montrer qu'elle a bien été couverte, et ainsi de suite. On procède de la sorte jusqu'à la fin de la règle. A ce moment, deux cas sont possibles :

- soit tout le plan est couvert et la règle à définitions multiples est cohérente du point de vue de la couverture.
- soit des portions de plan ne sont pas encore couvertes ; il y a donc un problème.

Tout le problème de cette technique réside dans l'apparition d'un opérateur logique dans les conditions. Il pose des difficultés quant à la définition des régions couvertes. Par la suite, nous regarderons plus en détail ce problème dans un développement détaillé de la technique. Mais, avant cela, il nous semble intéressant de l'illustrer par un exemple significatif.

Exemple : ECONOMIE $_{REG}$ = bonne si CROISSANCE $_{REG} \geq 0.04$ ou
CHOMAGE $_{REG} < 0.076$ **(1)**
moyenne si $0.02 \leq$ CROISSANCE $_{REG} < 0.04$ et
$0.076 \leq$ CHOMAGE $_{REG} < 0.082$ **(2)**
mauvaise si CROISSANCE $_{REG} < 0.02$ ou
CHOMAGE $_{REG} \geq 0.082$ **(3)**



Explication détaillée

Maintenant que le principe général est donné, nous allons envisager une explication plus mathématique du processus. Pour pallier l'aspect quelque peu opaque de ce développement, nous avons essayé de présenter dans la mesure du possible des graphiques.

Dans un premier temps, nous examinerons le cas trivial de la présence d'une seule grandeur dans l'ensemble des conditions. Ensuite, nous attaquerons le cas général de deux grandeurs présentes dans les conditions d'une règle à définitions multiples.

Lorsqu'une seule grandeur intervient dans l'ensemble des conditions, il n'est pas nécessaire d'utiliser l'espace à deux dimensions. Pour chaque condition, on cherche l'ensemble des valeurs couvertes pour cette grandeur (EV). A partir de l'ensemble de toutes les valeurs possibles pour la grandeur (E), nous allons déterminer, condition après condition, le nouvel ensemble de valeurs restant à couvrir en lui enlevant l'ensemble des valeurs couvertes. A la fin de la règle, si l'ensemble des valeurs restant à couvrir est égal au vide (Ø), le critère de **couverture** est vérifié.

Exemple : (Etude de cas n°1)

Soit la règle : REDUCTION $_{P}$ = 0,5 si RED $_{ABON}$ = vrai
1 si RED $_{ABON}$ = faux

E = ensemble des valeurs possibles de RED $_{ABON}$ = (vrai,faux)

Première condition : EV = (vrai)
E devient (E \ EV) = (faux)

Deuxième condition : EV = (faux)
E devient (E \ EV) = Ø

On obtient l'ensemble vide. Pour toutes les valeurs que peut prendre RED $_{ABON}$, il existe donc une condition qui couvre cette valeur.

Nous nous sommes demandé comment représenter un ensemble de valeurs[1] d'une grandeur. Si le type de la grandeur est entier ou réel, nous utiliserons la notation mathématique consacrée. Ainsi l'ensemble des nombres positifs est représenté par [0,+∞[. Si le type de la grandeur est de cardinal fini, un ensemble de valeurs sera désigné par une suite de valeurs entre parenthèses. Ainsi, l'ensemble des booléens se note (vrai,faux).

Tout au long de ce chapitre, nous allons utiliser une notation particulière pour synthétiser les principes explicités. En voici les éléments les plus fréquents ; les autres seront définis au fur et à mesure de l'exposé :

- E ou tout sigle commençant par E symbolise ce qui reste à couvrir. Il représente une portion du plan ou un ensemble de portions.
- EV ou tout sigle commençant par EV symbolise ce qui est couvert par une condition. Il représente une portion de plan ou un ensemble de portions.
- Ens1 \ Ens2 signifie qu'on enlève de l'ensemble de valeurs Ens1 l'ensemble des valeurs Ens2. Cette opération ne s'exécute que sur des ensembles de valeurs d'une grandeur. Le résultat sera toujours un ensemble de valeurs.
- Ens1 ∩ Ens2 signifie qu'on prend les valeurs comprises dans les deux ensembles (intersection). Ici, Ens1 et Ens2 représentent des portions du plan.

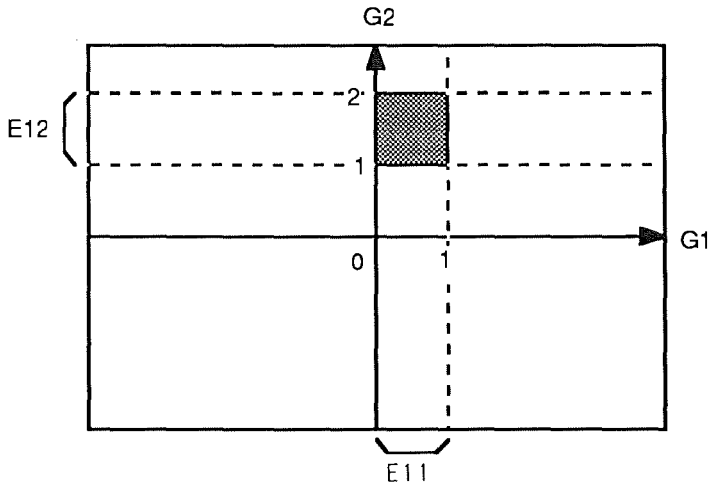
---

[1] Lorsqu'on parle d'ensemble de valeurs, il s'agit de portion, de zone de l'espace à deux dimensions. Ces trois vocables ont la même signification.

Examinons le cas plus complexe de l'intervention de plusieurs grandeurs dans l'ensemble des conditions d'une règle à définitions multiples. Il faut, à ce moment, travailler avec l'espace à deux dimensions. Le travail consistera, pour chaque condition, à trouver les portions de plan couvertes par celle-ci et de les enlever de l'ensemble des zones restant à couvrir. A tout instant, lors de l'analyse des conditions, nous disposons des éléments suivants :

- les deux grandeurs figurant dans les conditions (nommées G1 et G2 pour plus de facilités),
- l'ensemble des zones non encore couvertes qu'on appelle E et qui contient soit tout l'espace si aucune condition n'a encore été analysée, soit n zones. Chaque zone a la forme : Ei = Ei1 ∩ Ei2 qui représente l'intersection de deux ensembles de valeurs (Ei1 pour G1 et Ei2 pour G2).

Exemple : E11 = [0,1] et E12 = [1,2] définissent la zone de l'espace E11 ∩ E12 qui se représente graphiquement :



Ceci étant dit, trois types de conditions sont appréhendables. Premièrement, une condition sans opérateur logique se présente. L'ensemble des valeurs couvertes sera soit un ensemble de valeurs pour G1, soit un ensemble de valeurs pour G2. Supposons que la condition couvre un ensemble de valeurs pour G1 (on l'appelera EV1). La condition couvre EV1 quelle que soit la valeur prise par G2. Pour trouver les zones restant à couvrir, il suffit pour chaque élément de E de soustraire à Ei1 l'ensemble EV1. Le processus sera identique pour G2. Plus formellement, on peut écrire :

Si EV = EV1,

alors E = (...,Ei1 ∩ Ei2,...) devient E'= (...,(Ei1 \ EV1) ∩ Ei2,...).

Si EV = EV2,

alors E = (...,Ei1 ∩ Ei2,...) devient E'= (...,Ei1 ∩ (Ei2 \ EV2) ,...).

Deuxièmement, une condition avec un opérateur logique 'ou' se présente. Celle-ci définit deux ensembles de valeurs couvertes (EV1,EV2). Le 'ou' signifie que, si G1 a une valeur appartenant à EV1 et quelle que soit la valeur de G2, la condition est vérifiée et inversement pour G2. Par conséquent, à chaque élément de E, on va soustraire EV1 à Ei1 et EV2 à Ei2. On aura :

EV = (EV1,EV2)

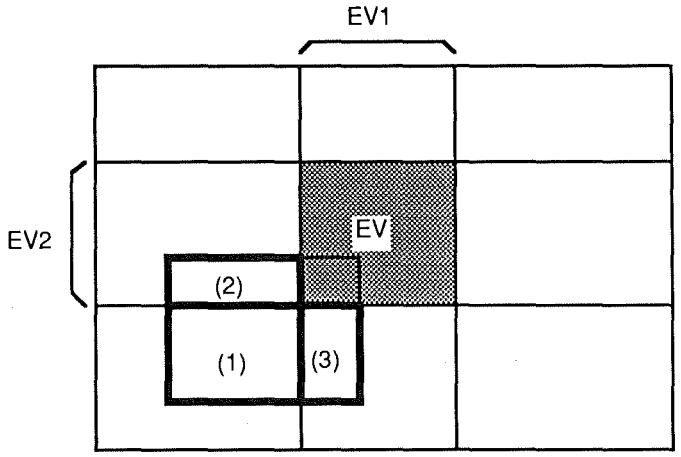
E = (...,Ei1 ∩ Ei2,...) devient E'= (...,(Ei1 \ EV1) ∩ (Ei2 \ EV2),...)

Troisièmement, une condition avec un opérateur logique 'et' se présente. Il s'agit du cas le plus difficile. La condition n'est vérifiée que pour certaines valeurs de G1 et certaines valeurs de G2. Elle définit une portion de plan représentée par l'intersection de EV1 et EV2. On a d'abord :

$EV = EV1 \cap EV2$

$E = (...,Ei1 \cap Ei2,...)$

Comment transformer les zones de E pour qu'il ne contienne plus que les zones non encore couvertes ? Le dessin de cette situation va nous permettre de trouver la solution :



Si on utilise le même principe que pour une condition avec un opérateur 'ou', on obtient : $(Ei1 \setminus EV1) \cap (Ei2 \setminus EV2)$ qui représente la zone (1) de notre dessin. Cependant, il faut encore couvrir les zones (2) et (3). Le dessin ci-dessus nous permet de les exprimer :

$(Ei1 \setminus EV1) \cap EV2$ pour (2),

$EV1 \cap (Ei2 \setminus EV2)$ pour (3).

Le point suivant va traiter un exemple complet.

Traitement d'un exemple

Prenons la règle à définitions multiples de l'étude de cas n°2 :

ECONOMIE $_{REG}$ = bonne si CROISSANCE $_{REG} \geq 0,04$ ou
CHOMAGE $_{REG} < 0,076$ **(1)**

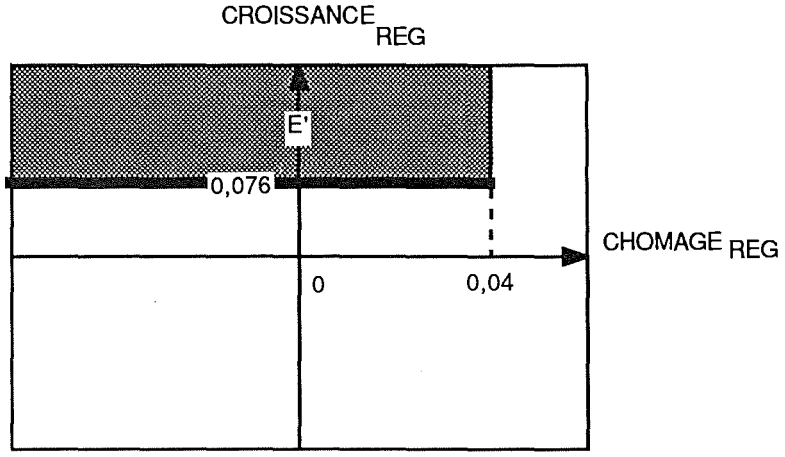
moyenne si $0,02 \leq$ CROISSANCE $_{REG} < 0,04$ et
$0,076 \leq$ CHOMAGE $_{REG} < 0,082$ **(2)**

mauvaise si CROISSANCE $_{REG} < 0,02$ ou
CHOMAGE $_{REG} \geq 0,082$ **(3)**

et analysons-la, condition après condition. Tout d'abord, fixons ce qui est connu :

- G1 désigne CROISSANCE $_{REG}$ et G2 CHOMAGE $_{REG}$,
- E représente tout l'espace. G1 et G2 étant de type réel, E peut s'écrire $(]-\infty,+\infty[\cap]-\infty,+\infty[)$.

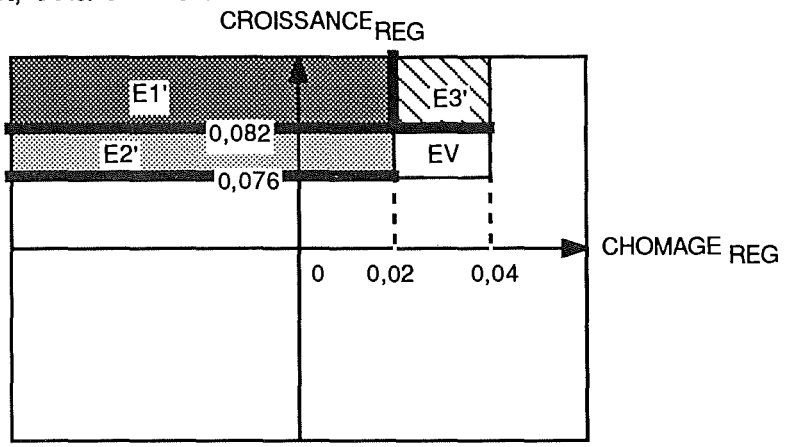
Première condition **(1)** : un opérateur 'ou' est repéré.
EV = (EV1,EV2) = ([0,04,+∞[,]-∞,0,076[)
E = (E11 ∩ E12) = (]-∞,+∞[∩]-∞,+∞[)
Calculons E' (ce qu'il reste à couvrir) :
E'= ((E11 \ EV1) ∩ (E12 \ EV2))
= ((]-∞,+∞[\ [0,04,+∞[) ∩ (]-∞,+∞[\ ]-∞,0,076[))
= (]-∞,0,04[∩ [0,076,+∞[)
Graphiquement[1] , cela donne :



Deuxième condition **(2)** : un opérateur 'et' est repéré.
EV = (EV1 ∩ EV2) = ([0,02,0,04[∩ [0,076,0,082[)
E = (E11 ∩ E12) = (]-∞,0,04[∩ [0,076,+∞[)
Calculons E' (ce qu'il reste à couvrir) :
E'= (E1',E2',E3')
avec E1'= (E11 \ EV1) ∩ (E12 \ EV2)
= (]-∞,0,04[\ [0,02,0,04[) ∩ ([0,076,+∞[\ [0,076,0,082[)
=]-∞,0,02[∩ [0,082,+∞[
E2'= (E11 \ EV1) ∩ EV2
= (]-∞,0,04[\ [0,02,0,04[) ∩ [0,076,0,082[
=]-∞,0,02[∩ [0,076,0,082[
E3'= EV1 ∩ (E12 \ EV2)
= [0,02,0,04[∩ ([0,076,+∞[\ [0,076,0,082[)
= [0,02,0,04[∩ [0,082,+∞[

[1] Les lignes plus épaisses, employées dans les graphiques, signifient que la borne est comprise dans la portion hachurée.

Graphiquement, cela donne :



Troisième condition **(3)** : un opérateur 'ou' est repéré.
EV = (EV1,EV2) = (]-∞,0,02[,[0,082,+∞[)
E = (E1,E2,E3)
= (]-∞,0,02[∩ [0,082,+∞[,]-∞,0,02[∩ [0,076,0,082[,
[0,02,0,04[∩ [0,082,+∞[)
Calculons E' (ce qu'il reste à couvrir) :
E'= (E1',E2',E3')
avec E1'= (E11 \ EV1) ∩ (E12 \ EV2)
= (]-∞,0,02[\ ]-∞,0,02[) ∩ ([0,082,+∞[\ [0,082,+∞[)
= Ø ∩ Ø = Ø
E2'= (E21 \EV1) ∩ (E22 \ EV2)
= (]-∞,0,02[\ ]-∞,O,O2[) ∩ ([0,076,0,082[\ [0,082,+∞[)
= Ø ∩ [0,076,0,082[= Ø
E3' = (E31 \EV1) ∩ (E32 \ EV2)
= ([0,02,0,04[\ ]-∞,O,O2[) ∩ ([0,082,+∞[\ [0,082,+∞[)
= [0,02,0,04[∩ Ø = Ø
E'= (Ø,Ø,Ø)
Tout l'espace a donc été couvert.

II.3.2.5.3. Vérification de non-recouvrement

Problème

Il faut empêcher que des conditions se recouvrent, c'est-à-dire que, pour des mêmes valeurs de grandeurs intervenant dans celles-ci, on puisse choisir deux ou plusieurs expressions définissant la grandeur expliquée par la règle. Un exemple simple nous montre plus exactement le problème.

Exemple : (Etude de cas n°1)

En transformant quelque peu la règle de définition de la grandeur REDUCTION p, on obtient :

REDUCTION $_P$ = 0,5 si RED $_{ABON} \geq 0$

1 si RED $_{ABON} \leq 0$

Dans le cas où RED $_{ABON}$ = 0, REDUCTION $_P$ peut prendre les valeurs 0,5 et 1.

Dans le point suivant, une idée générale du principe de résolution est présentée. Ensuite une explication plus détaillée est proposée avant de traiter un exemple complet.

Idée générale

Intuitivement, pour vérifier le non-recouvrement, il est nécessaire de regarder les conditions d'une règle à définitions multiples et de les comparer deux à deux pour trouver un éventuel recouvrement.

Comme nous travaillons dans l'espace à deux dimensions, les conditions représentent des zones de l'espace. Pour comparer deux conditions, on vérifie qu'elles ne recouvrent pas des parties semblables du plan. Si c'est le cas, alors il y a recouvrement. Reprenons l'exemple ci-dessus : la première condition couvre l'ensemble de valeurs pour RED $_{ABON}$ $[0,+\infty[$ et la deuxième l'ensemble $]-\infty,0]$. $\{0\}$ est commun aux deux conditions. Il y a donc un recouvrement pour certaines valeurs de RED $_{ABON}$.

Explication détaillée

Avant de commencer, il est nécessaire d'effectuer quelques modifications sur certains types de conditions afin de permettre la comparaison.

Si une seule grandeur intervient, aucune transformation n'est nécessaire. Chaque condition représente un ensemble de valeurs. La comparaison est immédiate. Il suffit de vérifier qu'aucun ensemble n'ait un sous-ensemble de valeurs commun avec une autre condition.

Si deux grandeurs interviennent dans les conditions, trois types de conditions se présentent. Pour chacune d'elles, donnons la forme générale de la zone du plan qu'elle recouvre :

- condition avec une seule grandeur :
  EV = EV1 ou EV = EV2
  avec EV = portion de l'espace couverte par la condition,
  EV1 = ensemble de valeurs pour la première grandeur,
  EV2 = ensemble de valeurs pour la deuxième grandeur.
- condition avec un opérateur logique "ou" : EV = (EV1,EV2).
- condition avec un opérateur logique "et" : EV = EV1 $\cap$ EV2.

Ces types de conditions couvrent des zones exprimées de façons différentes. Pour permettre une comparaison, il faut ramener celles-ci à une forme standard. On choisit la troisième forme présentée (condition avec un opérateur logique "et") comme standard car il est assez aisé de ramener les deux autres formes à celle-ci.

Nous devons donc exprimer chaque portion du plan couverte par une intersection d'ensembles de valeurs.

Transformons d'abord une condition avec une seule grandeur. Cette condition définit un ensemble de valeurs pour une des grandeurs quelle que soit la valeur de l'autre. La portion du plan couverte est cet ensemble de valeurs (défini par la condition) en intersection avec l'ensemble de toutes les valeurs possibles pour la grandeur non présente dans la condition. Plus concrètement,

Si EV = EV1, alors EV devient EV = EV1 $\cap$ E2.

Si EV = EV2, alors EV devient EV = E1 $\cap$ EV2.

avec E1 = ensembles de toutes les valeurs possibles pour la première grandeur.

E2 = ensembles de toutes les valeurs possibles pour la deuxième grandeur.

Transformons maintenant une condition avec un opérateur logique "ou". La condition spécifie, pour chaque grandeur, une portion du plan couverte quelle que soit la valeur de l'autre grandeur. Ces ensembles de valeurs couvertes pour chaque grandeur par la condition peuvent être transformés en deux portions du plan constituées par l'intersection de l'ensemble des valeurs couvertes pour une grandeur avec l'ensemble de toutes les valeurs possibles pour l'autre grandeur. Concrètement, on peut écrire :

EV = (EV1,EV2) devient EV = (EV',EV")

avec EV' = EV1 $\cap$ E2 et EV" = E1 $\cap$ EV2.

Lorsque toutes les transformations sont opérées, on compare chaque condition deux à deux, zone par zone. Si deux zones ont une portion d'espace commune, on conclut à l'existence d'un recouvrement. Sinon, il n'y a pas de problème. Comment comparer deux zones ?

Supposons les zones (EV1 $\cap$ EV2) et (EV3 $\cap$ EV4), il suffit de regarder ce que représente la portion de plan : (EV1 $\cap$ EV3) $\cap$ (EV2 $\cap$ EV4). Si elle est vide, il n'y a pas de recouvrement entre les deux zones.

Le traitement d'un exemple complet va éclairer cette explication.

Traitement d'un exemple

Reprenons la règle à définitions multiples déjà analysée du point de vue de la cohérence de couverture :

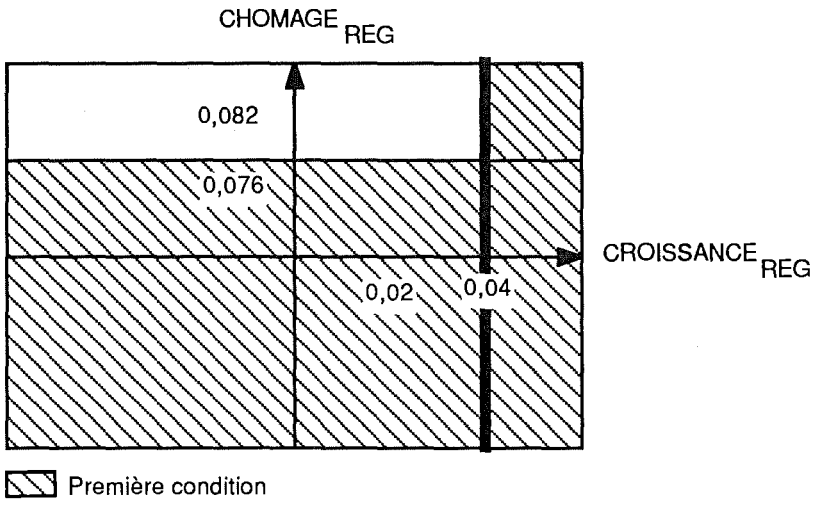
ECONOMIE $_{REG}$ = bonne si CROISSANCE $_{REG} \geq 0{,}04$ ou
CHOMAGE $_{REG} < 0{,}076$ **(1)**

moyenne si $0{,}02 \leq$ CROISSANCE $_{REG} < 0{,}04$ et
$0{,}076 \leq$ CHOMAGE $_{REG} < 0{,}082$ **(2)**

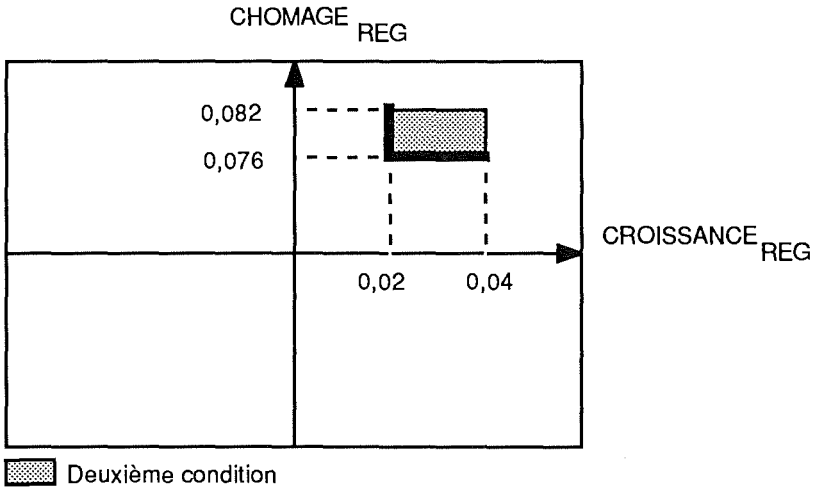
mauvaise si CROISSANCE $_{REG} < 0{,}02$ ou
CHOMAGE $_{REG} \geq 0{,}082$ **(3)**

On a CROISSANCE $_{REG}$ la première grandeur et CHOMAGE $_{REG}$ la deuxième.
E1 = $]-\infty,+\infty[$ et E2 = $]-\infty,+\infty[$

Transformation de la première condition **(1)** :
EV1 = ([0,04,+∞[,]-∞,0,076[) devient EV1 = (EV11,EV12)
avec EV11 = [0,04,+∞[∩]-∞,+∞[
EV12 =]-∞,+∞[∩]-∞,0,076[
Graphiquement, voici les zones couvertes par la première condition :

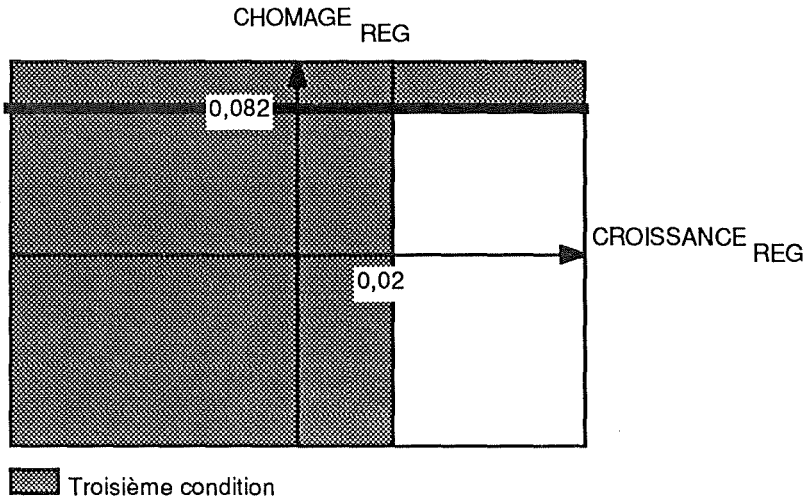


Deuxième condition **(2)** :
EV2 = [0,02,0,04[∩ [0,076,0,082[
Graphiquement, on couvre :



Transformation de la troisième condition **(3)** :
EV3 = (]-∞,0,02[,[0,082,+∞[) devient EV3 = (EV31,EV32)
avec EV31 =]-∞,0,02[∩]-∞,+∞[
EV32 =]-∞,+∞[∩ [0,082,+∞[

Graphiquement, on a la zone :



Comparaison de la première condition avec la deuxième :

- EV11 avec EV2, on obtient :

  $([0,04,+\infty[\cap [0,02,0,04[) \cap (]-\infty,+\infty[\cap [0,076,0,082[) = \emptyset \cap [0,076,0,082[= \emptyset$

- EV12 avec EV2, on obtient :

  $(]-\infty,+\infty[\cap [0,02,0,04[) \cap (]-\infty,0,076[\cap [0,076,0,082[) = [0,02,0,04[\cap \emptyset = \emptyset$

Comparaison de la première condition avec la troisième :

- EV11 avec EV31, on obtient :

  $([0,04,+\infty[\cap]-\infty,0,02[) \cap (]-\infty,+\infty[\cap]-\infty,+\infty[) = \emptyset \cap]-\infty,+\infty[= \emptyset$

- EV12 avec EV31, on obtient :

  $(]-\infty,+\infty[\cap]-\infty,0,02[) \cap (]-\infty,0,076[\cap]-\infty,+\infty[) =]-\infty,0,02[\cap]-\infty,0,076[$

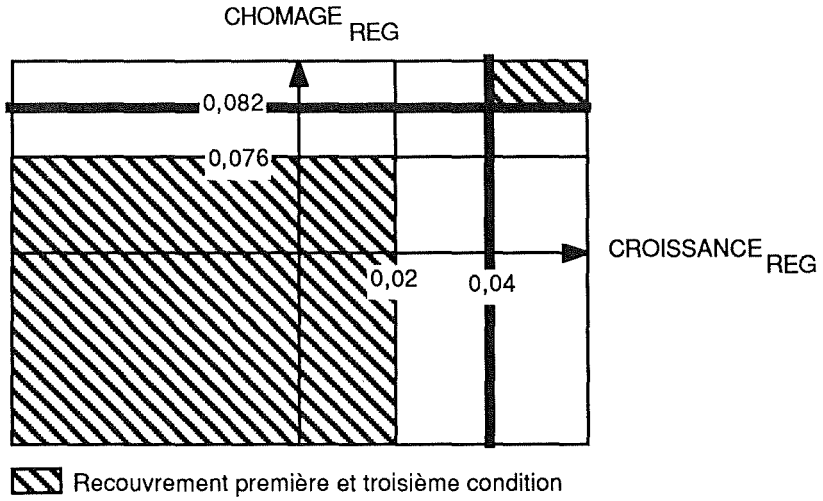
- EV11 avec EV32, on obtient :

  $([0,04,+\infty[\cap]-\infty,+\infty[) \cap (]-\infty,+\infty[\cap [0,082,+\infty[) = [0,04,+\infty[\cap [0,082,+\infty[$

- EV12 avec EV32, on obtient :

  $(]-\infty,+\infty[\cap]-\infty,+\infty[) \cap (]-\infty,0,076[\cap [0,082,+\infty[) =]-\infty,+\infty[\cap \emptyset = \emptyset$

La comparaison fait ressortir des zones que les deux conditions couvrent. Voici le graphe représentant cette situation :



Comparaison de la deuxième condition avec la troisième :

- EV2 avec EV31, on obtient :

  ([0,02,0,04[∩]-∞,0,02[) ∩ ([0,076,0,082[∩]-∞,+∞[) = Ø ∩ [0,076,0,082[= Ø

- EV2 avec EV32, on obtient :

  ([0,02,0,04[∩]-∞,+∞[) ∩ ([0,076,0,082[∩ [0,082,+∞[) = [0,02,0,04[∩ Ø = Ø

II.3.2.6. Cohérence des règles de récurrence

Ces règles particulières nécessitent un contrôle de cohérence spécifique. Une règle de récurrence exprime que la valeur d'une grandeur dépend d'une ou plusieurs de ses valeurs précédentes ou suivantes. Pour créer ce type de règles, on utilise la notion de paramètre externe. Les règles de récurrence ont été définies au point II.2.3.1., nous conseillons au lecteur de s'y reporter pour revoir les trois formes de règles.

De par la définition, on perçoit que des contrôles supplémentaires seront à effectuer concernant le paramètre externe ou encore le sens de la règle de récurrence.

La vérification proposée se divise en quatre étapes. La première a pour objectif de "déceler" les règles de récurrence dans un modèle. La deuxième contrôle la complétude de la règle en vérifiant l'initialisation par rapport au sens et au pas de celle-ci. La troisième s'intéresse aux divers types de grandeurs qui peuvent être présents dans la récurrence. La quatrième étape consiste à appliquer à la règle les principes de cohérence étudiés dans toute cette section.

Première étape

Une règle de récurrence n'est pas toujours repérable du premier coup d'oeil. En effet, dans certains cas, il n'apparaît pas explicitement dans la règle de définition d'une grandeur qu'elle dépend de ses valeurs précédentes ou suivantes. Pour des raisons légitimes, l'utilisateur a "camouflé" la règle de récurrence (en employant, par exemple, des grandeurs internes).

Exemple : (Etude de cas n°3)

$QRES_T = QINIT_T + QEMP_T - QCONS_T$

$QINIT_{T(1)} = QRES\text{-}INIT$

$QINIT_{T(i)} = QRES_{T(i-1)}$, i = (2..n)

En regardant ces règles séparément, il n'y a rien de particulier. Il n'y a aucune règle où la grandeur expliquée dépend de ses valeurs précédentes ou suivantes.

Toutefois, en remplaçant dans la règle de définition de $QINIT_{T(i)}$, $QRES_{T(i-1)}$ par sa règle, on s'aperçoit qu'il s'agit d'une règle de récurrence. On obtient :

$QINIT_{T(1)} = QRES\text{-}INIT$

$QINIT_{T(i)} = QINIT_{T(i-1)} + QEMP_{T(i-1)} - QCONS_{T(i-1)}$, i = (2..n)

En fait, on a défini une grandeur interne intermédiaire ($QRES_T$) pour simplifier le problème.

Pratiquement, il y a possibilité de règle de récurrence, lorqu'une grandeur expliquée a un paramètre externe portant sur sa dimension. Si la récurrence n'est pas visible, on remplace les grandeurs de la partie droite par leur règle de définition. Ce processus de remplacements s'arrêtera une fois qu'on a une règle de récurrence, c'est-à-dire que la grandeur expliquée dépend d'une de ses valeurs précédentes ou suivantes.

On essaye, dans la mesure du possible, de ramener la récurrence sous une des trois formes vues : la grandeur expliquée dépend de ses valeurs précédentes, de ses valeurs suivantes ou des deux.

Deuxième étape

Lorsque la règle de récurrence a sa forme standard, une première vérification se fait sur l'initialisation. On repère de quel type de règles il s'agit et, ensuite, on effectue la vérification. Celle-ci est relativement évidente puisqu'il suffit de regarder le pas employé.

Si la grandeur définie dépend d'une ou plusieurs de ses valeurs précédentes, on vérifie l'initialisation de la grandeur pour toute les valeurs du paramètre externe inférieures ou égales au pas utilisé.

Si elle dépend d'une ou plusieurs de ses valeurs suivantes, on vérifie l'initialisation pour toutes les valeurs du paramètre supérieures ou égales au cardinal du domaine de valeur de sa dimension moins le pas utilisé.

Si elle dépend de ses valeurs suivantes et précédentes, on vérifie l'initialisation pour toutes les valeurs du paramètre définies dans les deux formes précédentes.

Exemple : (Etude de cas n°3)

QINIT $_{T(1)}$ = QRES-INIT

QINIT $_{T(i)}$ = QINIT $_{T(i-1)}$ + QEMP $_{T(i-1)}$ - QCONS $_{T(i-1)}$, (i = 2..n)

QINIT $_{T}$ est défini à partir de sa valeur précédente. Le pas est de 1. Il faut, dans l'initialisation, définir QINIT $_{T(1)}$.

Troisième étape

Dans une règle de récurrence, plusieurs types de grandeurs peuvent intervenir. On va les classifier et définir, si nécessaire, des contraintes sur leur intervention. On aura la présence :

- ***éventuelle*** de grandeurs autres que la grandeur expliquée par la récurrence. Ce sont soit :
  - des grandeurs non dimensionnées.
  - des grandeurs dimensionnées par une dimension différente de celle sur laquelle s'effectue la récurrence.
  - des grandeurs dimensionnées par la dimension de récurrence. Ces grandeurs dimensionnées sont soit des données ou des attributs de la base de données, soit définies par une règle normale, soit définies par une règle de récurrence. Dans ce dernier cas, il faut vérifier que la grandeur est bien définie pour toutes les valeurs du paramètre qu'elle prendra.

    Exemples : - A $_{T(1)}$= 1

    A $_{T(i)}$= A $_{T(i-1)}$ + 2 , i = (2 .. 5)

    B $_{T(1)}$ = 2

    B $_{T(i)}$ = B $_{T(i-1)}$ + A $_{T(i+1)}$, i = (2 .. 5)

    Il y a un problème dans la définition de B $_{T(i)}$ car, si i égal cinq, A $_{T(6)}$ n'est pas défini par la règle qui définit A.

    - (Etude de cas n°3)

    QEMP $_{T(i-1)}$ est un attribut de la base de données dimensionné par la dimension de récurrence (T). Il n'y a pas de vérification à faire car il existe bien pour tous les T(i).

    - (Etude de cas n°3)

    QCONS $_{T}$ est défini par la règle : QCONS/km $_{T}$ * DISTANCE $_{T}$. Elle est définie pour toutes les entités de l'ensemble désigné par la variable entité T.

- ***systématique*** de la grandeur expliquée par la règle de récurrence dans l'expression de celle-ci (on peut avoir plusieurs occurrences). Elle est paramètrée soit par (i-k) soit par (i+k).
  Exemple : $QINIT_{T(i)} = QINIT_{T(i-1)}+QEMP_{T(i-1)}-QCONS_{T(i-1)}$, (i=2..n)
  $QINIT_T$ apparaît en partie gauche de la règle et elle est paramètrée par (i-1).

Quatrième étape

Finalement, comme nous l'avons déjà dit, il s'agit d'opérer les vérifications au niveau des dimensions et des types des grandeurs (cfr points II.3.2.1. à II.3.2.4). Pour ce faire, on ne prendra pas en compte le paramètre externe.

II.3.3. Cohérence globale

II.3.3.1. Introduction

Dans cette section, nous allons nous intéresser à la cohérence non plus au niveau d'une seule règle, mais au niveau de l'**ensemble des règles** du modèle. Dans ce contexte, une règle ne sera pas incohérente par rapport à elle-même, mais par rapport au modèle lui-même.
Exemple : Définition d'une grandeur interne qui n'est jamais utiliséedans aucune autre règle.

A notre connaissance, dans la littérature, le modèle de spécification qui nous concerne ne fait l'objet d'aucune étude de cohérence. Pour élaborer des règles de cohérence, nous nous sommes inspirés des récents travaux réalisés en intelligence artificielle concernant la **vérification de la cohérence d'une base de connaissances** [LOISEAU,90]. Le domaine des bases de données déductives a également été une source d'inspiration importante ([GALLAIRE,84], [RIET,90]).
Nous avons relevé les divers types d'incohérences possibles. Nous les reprenons en les adaptant à notre modèle de spécification. Nous en ajoutons d'autres, propres à notre formalisme.

Nous proposons ainsi une liste de règles incohérentes par rapport au modèle. Pour chacune d'elles, nous soulevons les dangers qu'elle peut engendrer et, ainsi, l'intérêt de les repérer. Nous proposons ensuite une méthode pour déceler ces incohérences. Les méthodes proposées sont essentiellement basées sur le graphe de dépendances entre grandeurs.

II.3.3.2. Règles non distinctes

Nous distinguerons les règles non distinctes en surface et les règles non distinctes en profondeur.

II.3.3.2.1. Distinction de surface

Définition

Deux règles sont non distinctes en surface ssi Elles ont un membre identique

Exemple : $MONTANT_{COM} = \sum_{LC} Q_{LC} * PRIX_{PRO}$

$$COMMANDE_{COM} = \sum_{LC} Q_{LC} * PRIX_{PRO}$$

Deux cas peuvent se présenter :

1. Soit les membres de gauche sont identiques

Cette éventualité est directement rejetée par la définition même du modèle de spécification. "On n'admettra pas dans un modèle qu'une grandeur soit définie par plus d'une règle"[HAINAUT,90]. Cette incohérence peut être repérée dès la saisie des règles.

2. Soit les membres de droite sont identiques

Ce cas n'entre pas en conflit avec la définition du formalisme. Il se pourrait même que le modèle s'exécute sans erreur avec la présence de deux règles non distinctes en surface.

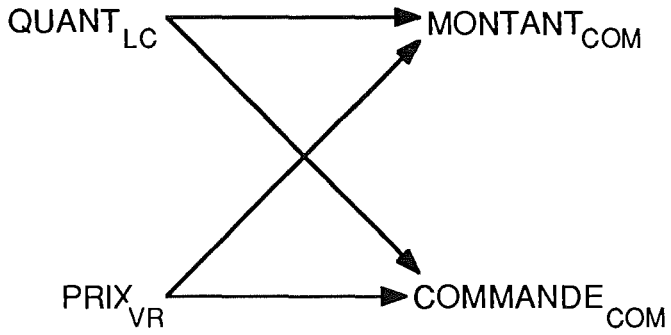
Intérêts et dangers

Lorsqu'une incohérence de ce type est repérée, il est important de la signaler à l'utilisateur afin d'uniformiser la définition des grandeurs internes et des résultats, ce qui augmente la lisibilité du modèle. On forcera ainsi l'utilisation d'un vocable unique dans les autres règles.
De plus, deux règles non distinctes peuvent révéler
- une erreur d'introduction de l'utilisateur.
- une erreur de conception plus grave (confusion entre deux grandeurs).

Contrôle

Nous ne nous intéressons ici qu'au deuxième cas. La source principale de vérification de cohérence est le graphe de dépendances.

Deux règles du modèle sont susceptibles d'être non distinctes si et seulement si elles ont les mêmes précédents. Dans ce cas, il suffit de comparer leur expression respective.
Exemple :



II.3.3.2.2. Distinction en profondeur

Définition

Deux règles sont non distinctes en profondeur ssi Par remplacements successifs des grandeurs qui les composent, les membres de droite sont identiques.

Exemple : R1 : D = 2 * E
R2 : B = D + C
R3 : A = 2 * E + C
R2 et R3 sont non distinctes en profondeur.

Il n'y a, de nouveau, pas de conflit avec le formalisme. Cependant, pour les mêmes raisons que nous avons citées au point précédent, il est souhaitable de faire disparaître une des deux règles.

Contrôle

Pour effectuer ce contrôle, il faut exprimer toutes les grandeurs internes et les résultats en fonction des données. Après cette transformation, si deux grandeurs sont définies par la même règle, alors elles sont non distinctes en profondeur. Pour réaliser cette contraction par remplacements successifs, le graphe de dépendances sera très utile.

II.3.3.3. Règle subsumante

II.3.3.3.1. Définition générale

Une règle est subsumée ssi Son déclenchement n'apporte aucune information supplémentaire

Exemple : une grandeur interne qui n'intervient dans aucune autre règle.

De nouveau, après cette définition générale, nous distinguerons la subsomption en surface et la subsomption en profondeur.

II.3.3.3.2. Subsomption en surface

Définition

Une règle est subsumée en surface ssi la grandeur interne apparaissant en partie gauche n'est utilisée dans aucune autre règle

Intérêts et dangers

L'apparition d'une règle subsumée peut être révélatrice d'erreurs graves:
- oubli de l'emploi de la grandeur interne dans une autre règle,
- erreur d'introduction de l'utilisateur (orthographe),
- cette grandeur interne est en fait un résultat.

Contrôle

On repère ce type d'incohérences grâce au graphe de dépendances. Il s'agit d'une grandeur qui n'a pas de descendant sans être un résultat.

II.3.3.3.3. Subsomption en profondeur

Définition

Une règle est subsumée en profondeur ssi la grandeur interne qu'elle définit n'intervient jamais dans la détermination d'un résultat

Exemple : R1 : D = A + B
R2 : E = D/4
R3 : F = G + H + A avec F résultat
R2 est subsumante en surface
R1 est subsumante en profondeur

Intérêts et dangers

La subsomption en profondeur est subordonnée à la subsomption en surface. Lorsque l'on a repéré une subsomption en surface, il est intéressant de découvrir l'éventuelle subsomption en profondeur qui peut en résulter. En effet, tout changement perpétré sur la règle subsumée en surface fera évoluer le statut des règles subsumées en profondeur.
Exemple : Si on décide de supprimer la grandeur, toutes les règles subsumées en profondeur qui lui sont reliées pourront être supprimées.

Contrôle

Il suffit de vérifier que la grandeur interne définie par la règle ne possède aucune grandeur résultat dans l'ensemble de ses descendants.

II.3.3.4. Règle indéclenchable

II.3.3.4.1. Définition générale

Une règle est indéclenchable ssi La grandeur définie par la règle peut ne pas prendre de valeur

Exemple : règle de définition où apparaît une grandeur interne qui n'est définie par aucune règle.

A la suite de cette définition générale, nous distinguerons l'indéclenchabilité de spécification et l'indéclenchabilité d'exécution.

II.3.3.4.2. Indéclenchabilité de spécification

Définition

Une règle est indéclenchable par spécification ssi La grandeur définie par la règle ne prend pas de valeurs quelles que soient les valeurs des données

A l'intérieur de ce point, nous distinguerons l'indéclenchabilité de spécification en surface et en profondeur.

Indéclenchabilité de spécification en surface

Définition

Une règle est indéclenchable par spécification en surface ssi Dans cette règle apparaît une grandeur n'appartenant ni à la base de données, ni aux données et qui n'a pas de règle de définition

Intérêts et dangers

Une incohérence de ce type révèle un problème grave car elle empêche l'exécution du modèle. Elle peut provenir soit de l'oubli de la déclaration d'une grandeur, soit de l'absence de règle de définition , soit d'une erreur d'introduction (orthographe)...

Contrôle

Une grandeur indéfinie n'appartient ni aux données ni à la base de données et ne possède aucun précédent.

Indéclenchabilité de spécification en profondeur

Définition

Une règle est indéclenchable par spécification en profondeur ssi Par remplacements successifs des grandeurs qui la composent, apparaît une grandeur n'appartenant ni aux données, ni à la base de données et qui n'a pas de règle de définition.

Intérêts et dangers

L'indéclenchabilité de spécification en profondeur est subordonnée à l'indéclenchabilité de spécification en surface. De manière analogue à la subsomption, lorsqu'une indéclenchabilité de surface est repérée, il est intéressant de découvrir celle en profondeur, car tout changement perpétré sur la règle en surface fera évoluer dans le même sens le statut des règles en profondeur.

Contrôle

Une grandeur indéfinie en profondeur possède un ancêtre qui n'a pas de précédent et qui n'appartient ni à la base de données, ni aux données du modèle.

II.3.3.4.3. Indéclenchabilité d'exécution

Définition

Une règle est indéclenchable d'exécution ssi la grandeur définie par la règle ne prend pas de valeurs pour certaines valeurs de données.

Dans notre modèle de spécification, cela correspond au problème de couverture dans le cas d'une règle à définitions multiples.
Exemple : A = 10 si B > 10
20 si B > 1 et B <= 10
B est défini dans l'intervalle [0,100]
A ne prend pas de valeur pour B = 0

Intérêts et dangers

Il est possible d'exécuter le modèle dans ces conditions en prévoyant une stratégie de propagation de la valeur indéfinie de la grandeur. Nous avons déjà envisagé ce problème au point II.2.6. De toute façon, quel que soit le choix, il est intéressant de signaler cette incohérence à l'utilisateur.

Contrôle

La vérification de ce genre d'anomalies se situe au niveau de la cohérence d'une règle. Elle a déjà été envisagée lors de l'étude de la couverture d'une règle à définitions multiples (point II.3.2.5.2.).

II.3.3.5. Règles contradictoires

Définition

Deux règles sont contradictoires ssi On peut trouver des conditions qui rendent contradictoires les conclusions des deux règles

Dans notre modèle de spécification, cela correspond au problème de non-recouvrement dans le cas d'une règle à définitions multiples.
Exemple : A = 10 si B > 0 et C > 0
= 20 si B > 0 ou C < 0
Pour B = 5 et C = 10, on peut avoir A = 10 ou A = 20. Il y a donc une contradiction au niveau des conditions employées dans cette définition multiple.

Intérêts et dangers

Il est impossible de laisser le choix au modèle entre deux valeurs pour une même grandeur car le modèle de spécification n'envisage aucune stratégie permettant le recouvrement avec l'introduction de grandeurs multivaluées et de facteurs d'incertitude [HOLSAPPLE,87].

Contrôle

Il est clair que la vérification de ce genre d'anomalies se situe au niveau de la cohérence d'une règle. C'est pour cette raison que la vérification de non-recouvrement a déjà été envisagée au point II.3.2.5.3.

II.3.3.6. Règle incohérente

Définition

Une règle, dans la base de règles, est incohérente ssi Il existe un ensemble de données réalistes qui permet d'obtenir une valeur qui ne respecte pas les contraintes d'intégrité du modèle

Exemple : Supposons la base de règles suivantes :

R1 : A = B + C
R2 : B = E + D
R3 : C = F / 5

De plus, on a la règle de contrainte suivante qui définit un domaine de valeurs sur A : $0 < A < 100$

Considérons l'ensemble des données suivant : (E=50, D=40 , F=55)

Après exécution du modèle, on obtiendra : A = 101.

On constate que A dépasse sa borne supérieure.

Intérêts et dangers

Repérer ce genre d'incohérence est tout à fait primordial. En effet, une contrainte du modèle exprime une propriété du réel perçu qui doit être respectée à tout moment. Si cette propriété est mise en défaut, il y a une erreur dans la détermination d'une ou plusieurs grandeurs.

Contrôle

La meilleure solution serait de repérer ce genre d'anomalies dès la spécification du modèle. Mais cet objectif est totalement utopique vu la diversité des contraintes d'intégrité et leur complexité.

Le contrôle devra donc être reporté au moment de l'exécution. On pourra le mettre en oeuvre à l'aide d'un mécanisme de "Event/Trigger", à la manière de la vérification des contraintes d'intégrité dans certains systèmes de gestion de bases de données [DITTRICH,86].

PARTIE III

DEMARCHE DE CONCEPTION

Partie III : Démarche de conception

Nous abordons dans cette partie le deuxième composant de notre méthodologie de conception d'applications d'aide à la décision : la démarche. Dans ce cadre, toute démarche est fondée sur des modèles et exploitée à l'aide d'outils logiciels.

Dans la partie II, nous avons tenté de définir le plus précisément possible les concepts fondamentaux du modèle de spécification. Cependant, il apparaît clairement que ce modèle, bien que de plus en plus précis et complet, n'a pas atteint sa pleine maturité. On pourrait donc penser qu'énoncer une démarche de conception est prématuré.

Toutefois, la construction d'un modèle directionnel repose sur un petit nombre de **principes intuitifs**. Ces principes ont été énoncés par Hainaut [HAINAUT,86a]. Nous repartons de ces principes pour définir quatre phases primordiales du processus de conception. Pour ces phases, nous tentons de donner une approche basée sur une **construction parallèle des graphes de dépendances.** Nous terminerons cette démarche par le traitement d'un court exemple.

La conception d'un modèle consiste à relever les données, les résultats, puis les relations qui permettent d'établir ces résultats à partir des données.

La conception est un processus créatif qui se heurtera à deux grands types de difficultés :

1. établir l'ensemble strictement suffisant et nécessaire des données.
2. établir les relations strictement nécessaires et suffisantes conduisant aux résultats à partir des données. En fait, on tente de créer un chemin des données vers les résultats : ce chemin est intuitivement donné par le graphe de dépendances entre grandeurs. L'aspect graphique pourra aider le concepteur à mieux découvrir les "bonnes" relations. Le graphe permettra de déceler plus facilement le parcours d'un chemin trop long ou sans issue. C'est pourquoi notre démarche en fait un usage intensif.

On peut adjoindre une troisième grande difficulté lorsque le modèle extrait des données de la base de données. En effet, il faut établir un lien entre les grandeurs du modèle et les entités de la base. Comme la démarche proposée est une démarche top-down, il faut tenter d'établir ce lien avant même de savoir quels attributs réels de la base de données seront utilisés pour déterminer la grandeur en question.

La démarche proposée tente au maximum de réduire ces trois grandes difficultés de conception.

III.1. Principes intuitifs

La démarche que l'on va suivre est basée sur quelques principes relativement intuitifs. Ces principes sont énoncés dans [HAINAUT,86]. Nous les reprenons pour en déduire nos phases de démarche.

Le premier principe est de **partir des résultats pour remonter vers les données**. Il est plus sûr, en effet, de se demander comment calculer un résultat que de se demander ce que l'on pourrait bien calculer d'utile (c'est-à-dire qui nous approche des résultats) à partir des données dont on dispose, d'autant plus que celles-ci sont généralement incomplètes, non indépendantes et/ou en surnombre.

Le deuxième principe consiste à n'**adopter que des règles très simples**, que l'on pourra facilement élaborer, justifier et expliquer. Ceci nous amènera en général à définir des grandeurs nouvelles, caractéristiques du domaine analysé, mais qui n'avaient pas été relevées en tant que données ou résultats. Ces grandeurs sont, par conséquent, du type interne (sauf grandeurs relevant du principe ci-dessous).

Le troisième principe est d'**élaborer, parallèlement à l'ensemble des règles, l'ensemble des données**. On partira d'un ensemble initial de données qui paraissent utiles lors d'une analyse préalable du domaine. Cet ensemble sera complété des grandeurs mises en évidence lors de l'élaboration des règles, et que l'on juge devoir faire partie des données.

Le quatrième principe définit la terminaison du processus. L'analyse s'arrête lorsque, dans les règles déjà construites, **chaque grandeur invoquée en partie droite est soit une donnée, soit une grandeur que l'on trouve en partie gauche d'une règle**. Les données qui n'ont été utilisées dans la partie droite d'aucune règle seront supprimées.

III.2. Démarche de conception

Maintenant que les principes sont posés, on peut en déduire une démarche. Celle-ci sera, par le premier principe une démarche top-down. Elle se déroule en quatre phases. Spécifions chacune d'elles.

III.2.1. Phase 1 : Définition des données et résultats

L'utilisateur doit avant toute chose définir précisément, en français, tous les résultats désirés. Pour chacun d'eux, s'il s'agit d'une grandeur dimensionnée, trois cas sont envisageables :

- la grandeur se rapporte directement à un type d'entités de la base de données. Il s'agit d'un attribut virtuel du type d'entités. La dimension est donc une variable entité définie sur celui-ci. Il faut encore spécifier sa règle de domaine, ce qui est parfois un processus complexe. C'est pourquoi cette conception fera l'objet d'un point spécial (III.2.5.).
- la grandeur n'a aucun rapport avec un type d'entités. Une ou plusieurs variables simples la dimensionnent. Il faut préciser l'ensemble des valeurs de chaque dimension.

- la grandeur est l'attribut d'une association virtuelle entre un type d'entités et une ou plusieurs variables simples. C'est donc une grandeur mixte vue comme un attribut virtuel répétitif du type d'entités de la variable entité. De nouveau, une définition des règles de domaine de chaque dimension doit être donnée.

Ensuite, le concepteur doit repérer un maximum de données qui seront nécessaires à l'initialisation du modèle. S'il s'agit de grandeurs dimensionnées, les trois cas du point précédent sont à consulter pour définir le domaine de valeurs de la dimension de la donnée.

III.2.2. Phase 2 : Règles de définition des grandeurs

Cette phase se prolongera tant qu'il existe un résultat ou une grandeur interne non encore expliquée par une règle simple. Pour la formuler, il est conseillé de d'abord définir les dépendances de la grandeur non expliquée avec d'autres, c'est-à-dire de construire le graphe de dépendances entre grandeurs. On peut écrire cela sous la forme :

Grandeur $_{DIM}$ = (grandeur1 $_{D1}$, grandeur2 $_{D2}$, ..., grandeurN $_{DN}$)

avec N petit (car on désire des formules simples) et DIM, D1, ..., DN des ensembles de dimensions éventuellement vides dans le cas de grandeurs simples.

Cette notation signifie que la grandeur expliquée verra apparaître dans sa règle de définition les grandeurs : grandeur1 $_{D1}$, grandeur2 $_{D2}$, ..., grandeurN $_{DN}$. Mais elle ne précise pas encore la relation mathématique entre celles-ci.

Cette dépendance fait appel à des grandeurs connues ou inconnues. Ce dernier cas pose le problème de savoir si grandeuri $_{Di}$ n'est pas une donnée non encore repérée. Si oui, on la met dans l'ensemble des données (à condition que ce ne soit pas un attribut réel d'une base de données). Sinon, on met grandeuri $_{Di}$ dans l'ensemble des grandeurs internes restant à expliquer. Cette grandeur devra alors faire l'objet d'une définition précise en français.

De plus, si grandeuri $_{Di}$ est dimensionnée, trois cas doivent être envisagés :

- Di contient une seule dimension se rapportant directement à un type d'entités de la base de données. Grandeuri $_{Di}$ est un attribut dimensionné par une variable entité qui doit être mise en relation[1] avec la variable entité éventuellement présente dans DIM.
- si grandeuri $_{Di}$ ne se rapporte pas à un type d'entités, elle est dimensionnée par une ou plusieurs variables simples. Celles-ci doivent être mises en relation[1] avec les variables simples éventuellement présentes dans DIM.
- Grandeuri $_{Di}$ peut être une grandeur mixte pour laquelle on combine les remarques des deux premiers cas.

Pour les grandeurs dont Di est non vide, de nouvelles dimensions ont peut-être été définies. Il convient d'en donner une définition précise en français et d'en expliquer le domaine de valeurs (règle de domaine).

---

[1] L'étude de cohérence du point II.3. peut aider à établir cette relation entre dimensions.

Pour terminer cette phase, il faut donner la formule mathématique précise et appliquer la **vérification de cohérence** au niveau de la règle pour en dégager tous les problèmes éventuels.

Il peut s'avérer que la formule mathématique soit un appel à un sous-modèle déjà défini dont l'élaboration est remise à plus tard.

Cette deuxième phase est terminée lorsqu'il ne reste plus de grandeurs non expliquées.

III.2.3. Phase 3 : Règles de contrainte

S'il est nécessaire d'imposer des règles de contrainte sur le modèle, on les exprime. On est parfois amené à définir dans de telles règles des grandeurs internes qu'on explique comme dans la deuxième phase.

III.2.4. Phase 4 : Suppression des données non utilisées

On élimine les données qui ne sont utilisées dans aucune règle.

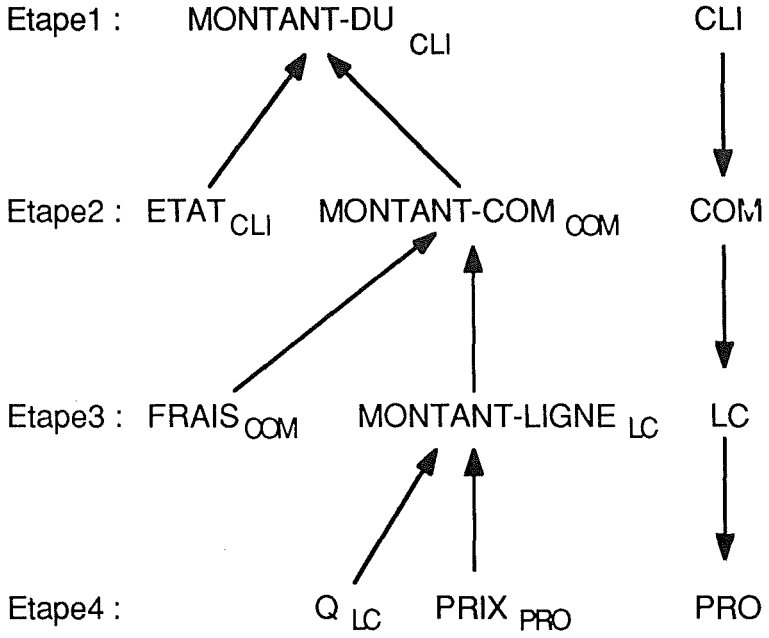
III.2.5. Règles de domaine

L'écriture des règles de domaine est un processus qui se situe dans les trois premières phases de conception d'un modèle. Les dimensions, si elles jouent un rôle essentiel dans le modèle de spécification, sont moins primordiales dans une démarche de conception. Il est clair que le graphe de dépendances au niveau des grandeurs est bien plus explicite de la solution que le graphe de dépendances entre variables entité (cette différence fondamentale entre grandeurs et dimensions a été mise en évidence au point II.2.2.).

Cependant, il n'est pas souhaitable de rejeter cette construction après l'établissement des règles du problème. Notre courte expérience du domaine nous a montré qu'il est préférable de mêler l'écriture des deux types de règles que de scinder le processus en deux.

Dans la plupart des cas, le graphe de dépendances entre variables entité se crée de manière parallèle au graphe de dépendances entre grandeurs. En effet, la dimension du résultat ne dépendra généralement pas d'une autre variable entité. De même la création de nouvelles variables entité et de leur règle de domaine se fera généralement à partir des variables entité déjà définies.

Exemple : Reprenons l'exemple introductif des clients de la figure 2.3.
En simplifiant, le concepteur du modèle est passé par quatre grandes étapes s' il a suivi notre démarche top-down.
La figure proposée ci-dessous montre bien le parallélisme qui existe entre la construction des deux graphes de dépendances.



Pour concevoir la règle de domaine qui définit une variable entité, on procédera comme suit :

- on recherche tous les éléments dont la variable entité doit dépendre. On obtient alors une expression du type :
  $Ve = (X_1, X_2, ..., X_N)$
  où Ve est la variable entité à définir
  X_i est soit une valeur,
  soit une grandeur,
  soit un type d'entités,
  soit une variable entité.
- on tente alors d'établir le lien entre les X_i et la variable Ve.
  Dans le cas où X_i est une valeur ou une grandeur, le lien consistera en une condition d'appartenance ; dans le cas où c'est un type ou une variable entité, le lien sera créé grâce à une condition d'association.

III.3. Traitement d'un exemple

Utilisons notre démarche pour concevoir un modèle répondant à l'énoncé de l'étude de cas n°1.

III.3.1. Phase 1 : Définition des données et résultats

Après la lecture de l'énoncé du problème, il apparaît que l'objectif de ce modèle est de confirmer si l'abonnement à une revue est intéressant pour une bibliothèque. Cela peut être réalisé en calculant le coût de la revue (**COUT-REV**). Si ce coût est supérieur à zéro, la revue n'est pas rentable.

La grandeur COUT-REV se rapporte directement au type d'entités REVUE de la base de données. En effet, elle peut être considérée comme un attribut virtuel d'une entité REVUE. COUT-REV est donc dimensionnée par une variable entité définie sur REVUE, soit REV, dont il convient de définir le domaine de valeurs.

On va calculer le coût pour une revue identifiée par son nom (**NOMR**). REV désigne donc la revue de nom NOMR. On peut écrire :

REV = (NOMR) ou plus explicitement REV=REVUE(:TITRE=NOMR)

Pour trouver cette règle de domaine, nous avons défini une nouvelle grandeur (NOMR). D'après l'énoncé, celle-ci peut être classée dans les données du modèle. Il s'agit une grandeur simple représentant le titre d'une revue.

Maintenant que le résultat est entièrement spécifié, il faut définir les autres données repérées dans l'énoncé et qui ne sont pas présentes dans la base de données. Nous disposons déjà du nom de la revue. Le résultat va être calculé sur un certain nombre d'années ; nous devons fixé l'année de début de l'analyse. Nous appelerons cette grandeur simple **A-D**. La définition de ces grandeurs termine la première phase de conception.

III.3.2. Phase 2 : Règles de définition des grandeurs

Cette phase durera tant qu'il restera des résultats ou des grandeurs internes non expliqués. Tout au long de cette deuxième phase, on construit les dépendances entre grandeurs. Pour voir comment la démarche construit le graphe de dépendances entre grandeurs, nous avons placés des références en marge droite lors de la définition de chaque grandeur. Ces références ont été reportées sur le graphe (figure 3.1., à la suite du traitement de l'exemple) afin de permettre au lecteur de percevoir le chemin choisi par les concepteurs. Nous conseillons donc au lecteur de consulter le graphe parallèlement à la démarche.

Pour chaque grandeur, nous allons tenter de découvrir les grandeurs dont elle dépend pour ensuite lui donner une règle de définition. A ce stade, **COUT-REV** est la seule grandeur dont on doit trouver la règle de définition. Le coût de la revue dépend du coût de paiement des abonnements à cette revue (COUT-PAIEMENT) et des rentrées (montant des prêts) qu'elle engendre (RENTREE). Ces deux grandeurs sont dimensionnées par REV : elles sont des attributs virtuels du type d'entités REVUE. On écrit la dépendance : COUT-REV $_{REV}$ = (COUT-PAIEMENT $_{REV}$,RENTREE $_{REV}$). Il est clair que le coût de la revue est la différence entre ces deux grandeurs. La règle de définition devient :

$$\text{COUT-REV}_{REV} = \text{COUT-PAIEMENT}_{REV} - \text{RENTREE}_{REV} \qquad (1)$$

Nous avons deux grandeurs internes qui doivent être expliquées COUT-PAIEMENT $_{REV}$ et RENTREE $_{REV}$.

COUT-PAIEMENT $_{REV}$ dépend du prix payé par la bibliothèque pour les abonnements à REV depuis l'année A-D. Ce prix (PRIX) se rapporte au type d'entités ABONNEMENT. Pour lier cette grandeur à la base de données, on crée la variable entité ABO désignant tous les abonnements à la revue REV depuis l'année A-D. On a la dépendance ABO=(REV,A-D). En tentant d'établir les liens entre ABO et les deux grandeurs dont elle dépend, on établit la règle de domaine suivante :

ABO = ABONNEMENT((aa:ANNEE(:AN ≥ A-D)) et (ar:REV))

PRIX $_{ABO}$ est un attribut réel du type d'entités ABONNEMENT. Il ne nécessite aucune règle de définition.

Le montant des paiements est égal à la somme de tous les prix des abonnements :

$$\text{COUT-PAIEMENT}_{REV} = \sum_{ABO} \text{PRIX}_{ABO} \qquad \textbf{(2)}$$

Il faut à présent développer la grandeur interne **RENTREE $_{REV}$**. Elle dépend des rentrées financières engendrées par chaque prêt concernant REV (RENTREE-PRET). Cette grandeur se rattache au type d'entités PRET ; on la dimensionne dès lors par une variable entité P définie sur PRET. Cette dimension désigne un prêt concernant la revue REV, ce prêt ayant été effectué après l'année A-D. On a la dépendance suivante : P=(A-D,REV). Pour être lié à la variable REV, le prêt P doit porter sur un exemplaire d'un numéro de la revue REV. Pour lier le prêt P à A-D et s'assurer qu'il lui est postérieur, on utilise l'attribut DATEDEB dans une condition d'appartenance. On obtient :

P = PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV))) et
(:Fannée(DATEDEB) ≥ A-D))

Pour calculer les rentrées d'une revue (RENTREE $_{REV}$), on doit additionner chaque rentrée de prêt P.

$$\text{RENTREE}_{REV} = \sum_{P} \text{RENTREE-PRET}_{P} \qquad \textbf{(3)}$$

La grandeur **RENTREE-PRET $_{P}$** dépend du montant du prêt P (MONTANT-PRET) et de l'amende *éventuelle* engendrée par un retard (AMENDE-PRET). Ces deux grandeurs sont dimensionnées par P. Ce sont des attributs virtuels du type d'entités PRET.

La grandeur RENTREE-PRET $_{P}$ est égale à la somme de ces deux grandeurs internes.

$$\text{RENTREE-PRET}_{P} = \text{MONTANT-PRET}_{P} + \text{AMENDE-PRET}_{P} \qquad \textbf{(4)}$$

Développons d'abord **MONTANT-PRET** $_P$. Elle dépend du montant du prêt (MONTANT) et de la réduction éventuelle obtenue par l'emprunteur (REDUCTION). **MONTANT** est un attribut réel du type d'entités ANNEEXEM (chaque année le montant du prêt pour un exemplaire est réajusté). Pour extraire les valeurs désirées de cet attribut, on le dimensionne par une variable entité, soit PA, désignant une ANNEEXEM liée à l'exemplaire du prêt P et dont l'année est égale à l'année du prêt P. On obtient la dépendance PA=(P,DATEDEB) ou, plus explicitement,

PA = ANNEEXEM((ea:EXEMPLAIRE(concerne:P)) et
(aae:ANNEE(:AN=Fannée(DATEDEB $_P$))))

REDUCTION est un attribut virtuel du type d'entités PRET. Il est dimensionné par P.

On peut écrire à présent la règle de définition de MONTANT-PRET.

$$\text{MONTANT-PRET}_P = \text{MONTANT}_{PA} * \text{REDUCTION}_P \qquad (5)$$

Il reste à définir **REDUCTION** $_P$ qui dépend d'une caractéristique de l'abonné de la bibliothèque contractant l'emprunt P. En effet, celui-ci peut disposer d'une réduction sur un prêt (RED).

RED se rapporte donc au type d'entités ABONNE dont il est un attribut réel. Pour extraire les bonnes valeurs d'attribut, on le dimensionne par une variable entité (ABON) désignant l'abonné qui a contracté le prêt P. On a :

ABON = (P) ou plus explicitement ABON = ABONNE(bénéficie:P)

Si l'abonné ABON bénéficie d'une réduction, il ne payera que la moitié du prix. On peut décrire cette situation par une règle à définitions multiples :

REDUCTION $_P$ = 0.5 si RED $_{ABON}$ = vrai
1 si RED $_{ABON}$ = faux (6)

Revenons à la grandeur **AMENDE-PRET** $_P$. L'emprunteur doit payer une amende uniquement s'il dépasse la durée du prêt. Pour déterminer s'il y a lieu de pénaliser l'abonné, il faut déterminer s'il y a eu un retard. La grandeur DEPASS-DATE nous donnera ce renseignement.

S'il y a eu retard, la grandeur AMENDE-PRET $_P$ dépend du montant de l'amende fixée chaque année (AMENDE) et du nombre de jours de retard du prêt (DUREE-DEPASS).

AMENDE se rapporte au type d'entités ANNEE, on la dimensionnée par A qui désigne l'année de la date de début (DATEDEB) d'un prêt P. AMENDE est un attribut réel du type d'entités ANNEE. On peut écrire la règle de domaine suivante pour A :

A = ANNEE(:AN=Fannée(DATEDEB $_P$))

DUREE-DEPASS et DEPASS-DATE sont dimensionnées par P car elles sont des attributs virtuels du type d'entités PRET.

Dans ce contexte, on a la règle suivante pour AMENDE-PRET $_P$:

AMENDE-PRET $_P$ = 0 si DEPASS-DATE $_P$ = faux
DUREE-DEPASS $_P$ * AMENDE $_A$ si DEPASS-DATE $_P$ = vrai
(7)

DUREE-DEPASS $_P$ dépend de la date de début du prêt (DATEDEB), de la date de fin de prêt (DATERET). Ces deux grandeurs sont des attributs réels du type d'entités PRET. Elles sont dimensionnées par P. Si on essaye de trouver une règle de définition pour DUREE-DEPASS $_P$, on découvre rapidement qu'il nous faut la durée permise pour un emprunt (DUREE). Cette information n'est pas présente dans la base de données et nous ne disposons d'aucun moyen pour la calculer. Il s'agit donc d'une donnée qui n'avait pas été répertoriée lors de la lecture de l'énoncé. Maintenant, nous pouvons expliquer DUREE-DEPASS $_P$ de la façon suivante :

$$\begin{aligned} \text{DEPASS-DATE}_P = \text{vrai si DATERET}_P - \text{DATEDEB}_P > \text{DUREE} \\ \text{faux si DATERET}_P - \text{DATEDEB}_P \leq \text{DUREE} \end{aligned} \qquad \textbf{(8)}$$

DUREE-DEPASS $_P$ dépend de DATEDEB, DATERET et DUREE. On définit cette dernière grandeur par la règle :

$$\text{DUREE-DEPASS}_P = \text{DATERET}_P - \text{DATEDEB}_P - \text{DUREE} \qquad \textbf{(9)}$$

Il ne reste plus de résultats ou de grandeurs internes non expliqués : la phase deux est donc terminée.

III.3.3. Phase 3 : Règles de contrainte

Aucune contrainte n'est nécessaire pour le fonctionnement du modèle.

III.3.4. Phase 4 : Suppression des données non utilisées

Toutes les données définies ont été utilisées.

III.3.5. Conclusion

Cet exemple nous montre comment utiliser la démarche de manière à assurer une conception rigoureuse. Mais il faut bien admettre que notre méthode est prototypique. Elle devrait être testée maintes fois par des utilisateurs finals avant d'atteindre une certaine maturité.

Toutefois, notre courte expérience a révélé qu'une démarche top-down basée sur la création de règles simples (par l'intermédiaire des grandeurs internes) permettait de bien appréhender les situations à modéliser.

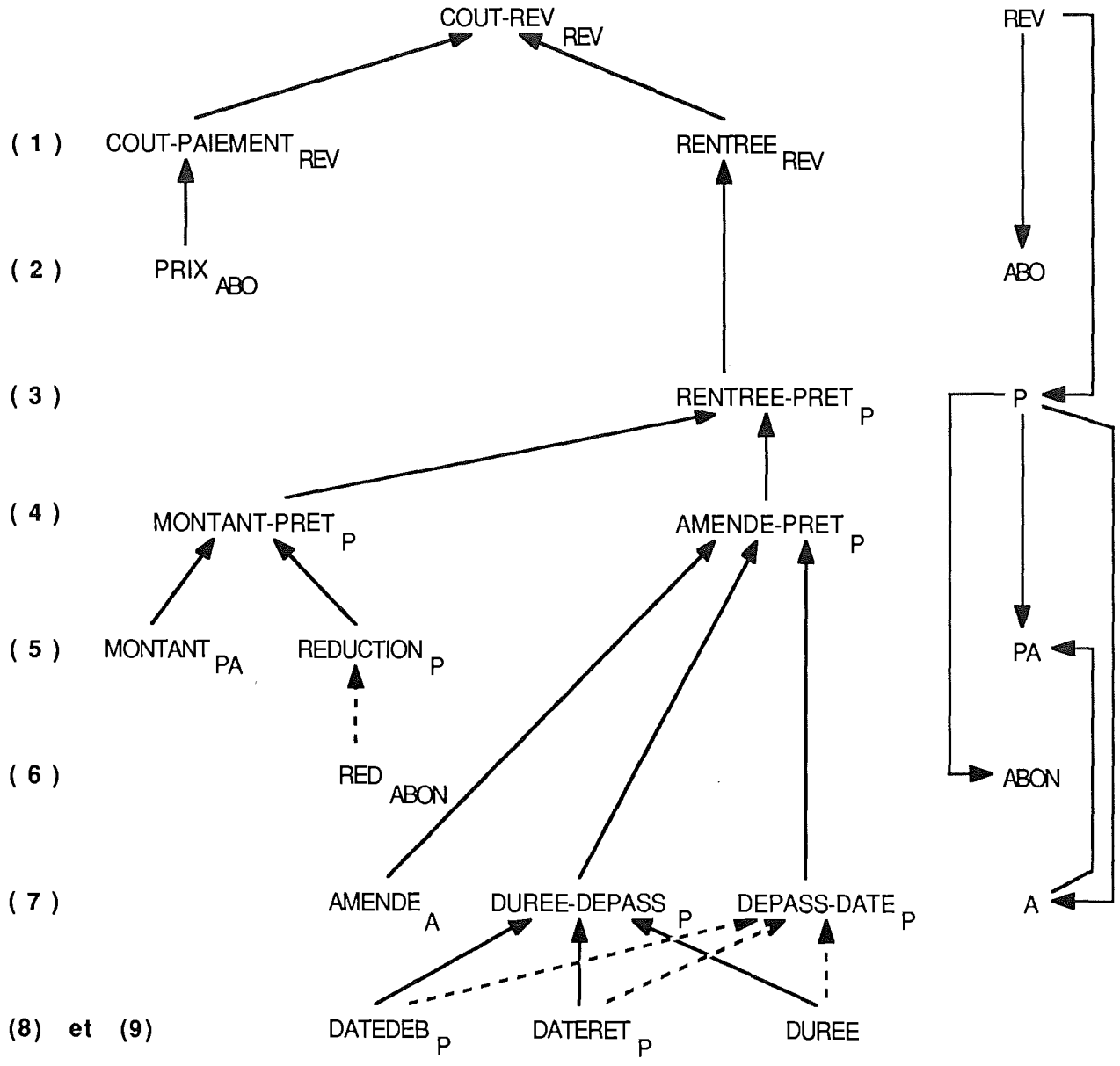


Figure 3.1. : Graphes de dépendances (étude de cas n°1)

PARTIE IV

OUTILS LOGICIELS

PARTIE IV : OUTILS LOGICIELS

Dans cette partie, nous nous sommes interrogés sur les outils logiciels qui pourraient aider l'utilisateur dans les différentes étapes de développement d'un modèle. Quels seraient les outils qui l'aideraient à accomplir sa tâche de la meilleure manière qui soit ? Actuellement, au niveau de la conception, il nous semble prématuré de parler de logiciels d'aide. La démarche proposée étant toujours à l'état prototypique, il est impossible, à ce stade, de parler d'outils d'aide à la conception de modèles.

Par contre, ce mémoire ayant défini de manière précise un noyau de concepts, il est possible d'envisager la mise en oeuvre de ceux-ci dans des systèmes d'aide à la décision.

Dans un premier point, nous envisageons trois grandes possibilités de mise en oeuvre d'un modèle selon l'outil utilisé. Nous tenterons pour chaque cas de donner des pistes pour une traduction systématique d'un modèle.

Dans un second point, nous présenterons les deux fonctions que nous avons mises au point et qui assurent une vérification de cohérence au niveau des règles du problème et des règles de domaine.

IV.1. Mise en oeuvre d'un modèle

L' objectif de ce mémoire est d'aider l'utilisateur à concevoir des applications d'aide à la décision. Pour ce faire, nous avons proposé un modèle de spécification. Mais si la mise en oeuvre des concepts développés pose des problèmes à l'utilisateur, tout le travail de recherche effectué dans les étapes antérieures du processus est inutile. C'est pourquoi ce point s'interroge sur les possibilités de traduction d'un modèle dans un système d'aide à la décision.

Notre but n'est pas d'analyser les détails d'une telle mise en oeuvre, mais plutôt de présenter succinctement quelques orientations possibles. Nous allons envisager trois types de systèmes afin de montrer le caractère général de notre modèle de spécification. Il s'agira d'un tableur, d'un système de traitement de problèmes généralisé (STPG) et d'un système de gestion de base de données classique.

IV.1.1. Représentation d'un modèle par un tableau

Nous supposons que le tableur analysé n'est couplé avec aucun gestionnaire de base de données. Le tableur ne fait donc partie d'aucun logiciel intégré (cette solution de mise en oeuvre sera étudiée au point suivant). Il est donc impossible d'extraire des valeurs d'une base de données. Seuls les modèles qui ne font aucun usage de variables entité pourront être représentés dans des feuilles électroniques.

Les principes de traduction proposés sont issus de [HAINAUT,86a].

L'implantation d'un modèle dans une feuille électronique sous la forme d'un tableau s'effectue en deux phases. La première est celle du choix de la position de chaque grandeur par l'**élaboration de la maquette** du tableau. La seconde consiste à **traduire les règles** en expressions acceptées par le tableur.

L'élaboration d'une maquette s'effectue sur une feuille de papier quadrillée. On regroupera les données ainsi que les résultats dans des portions de la feuille. On choisira soigneusement les titres, les commentaires, les unités, ainsi que toute information textuelle qui permet à l'utilisateur de se servir du tableau. Certains

fragments du modèle seront présentés sous forme tabulaire. Si plusieurs grandeurs sont fonction d'une dimension, on assignera à celles-ci des colonnes adjacentes, chaque ligne correspondant à une valeur de la variable dimension. Si une grandeur est fonction de deux dimensions, on lui consacrera un tableau dont les colonnes correspondent aux valeurs d'une dimension et les lignes aux valeurs de l'autre dimension.

Le principe du tableau est d'assigner une cellule ou un ensemble de cellules à chaque grandeur retenue dans le modèle. Il faut alors traduire les règles selon la syntaxe du langage du tableur. Comme, dans une formule, une grandeur est désignée par l'adresse de sa cellule, il est pratique de se construire une table des grandeurs qui indique l'adresse à laquelle chaque grandeur est implantée. Ainsi traduite, une règle devient exécutable.

Nous trouverons également des principes de traduction similaires dans [RONEN,89].

IV.1.2. Mise en oeuvre d'un modèle dans un logiciel intégré

Un tableur seul ne permet pas de mettre en oeuvre des modèles liés à une base de données. Il est donc intéressant de regarder les possibilités de traduction d'un modèle dans un STPG intégrant gestionnaire de bases de données et tableur.

La représentation d'un modèle dans un tel système s'effectue en **trois phases** : l'élaboration de la maquette, la mise en place des données de la bases de données et la traduction des règles.

L'**élaboration de la maquette** permet de fixer le format de la feuille électronique, de déterminer la place des grandeurs. Le procédé est identique à celui de la représentation d'un modèle par un tableau (cfr point précédent).

La mise en place des données de la base de données consiste à créer de petites **macro** (souvent des programmes) **allant chercher des valeurs d'un attribut dans la base de données** et important celles-ci dans la feuille électronique à l'endroit qui avait été assigné lors de l'élaboration de la maquette. C'est dans cette deuxième phase qu'apparaît clairement le lien entre le tableur et le gestionnaire de bases de données.

Pour finir, il faut **traduire les règles** du problème selon la syntaxe du langage du tableur.

Concrètement, comment traduit-on un modèle dans un logiciel intégré ? Au cours de notre stage en France, nous avons eu la possibilité d'utiliser le système GURU qui est l'exemple typique du logiciel intégré. Nous avons décidé de mettre en oeuvre la première étude de cas (étude de la rentabilité d'une revue). Dans la deuxième annexe, le lecteur trouvera le détail des trois phases de la représentation de l'étude de cas dans le système GURU.

Dés à présent, nous pouvons tirer un certain nombre de conclusions de cette démarche. Pour réaliser cette traduction, il a fallu se documenter pour résoudre les problèmes qui se présentaient surtout au niveau de la création des macro. Ce travail a pris quelques jours ce qui n'est pas négligeable pour une étude de cas somme toute assez simple. Notre manque d'expérience en la matière est une explication possible. De plus, le logiciel s'est montré "capricieux" pour tout ce qui concernait les transferts de données entre ses différents composants (exportation des données de la base de données vers le tableur). Des utilisateurs non expérimentés seront amenés à réaliser le même genre de travail.

Dans ce contexte, un générateur de code peut devenir une nécessité. Il serait utile pour construire systématiquement les macro destinées à aller chercher des

données dans la base de données et à traduire les règles. Cela permettrait à l'utilisateur de se détacher de la syntaxe utilisée pour développer les macro.

La production de requête pour extraire les données se fera à l'aide du langage de manipulation de données. Le formalisme des règles de domaine est défini de telle sorte que la production de requête est assez aisée. Il est possible de définir des règles de transformation afin de rendre chaque requête conforme au SGBD utilisé [HAINAUT,86]. On pourrait également concevoir la création d'un générateur de code automatisant entièrement la tâche.

Dans le même ordre d'idée, la traduction des règles du problème est relativement simple à cause du formalisme ne laissant la place à aucune interprétation et n'engendrant que des règles arithmétiques aisées à programmer.

IV.1.3. Mise en oeuvre d'un modèle dans un système de gestion de bases de données (SGBD) classique

Un système de gestion de données classique est caractérisé par un langage de manipulation de données (DML : data manipulation langage).

Pour réaliser des programmes d'application, on insère le DML dans un langage de programmation habituel.

Exemple : NDBS : langage hôte : Pascal
DML : data base handler
ORACLE : langage hôte : COBOL
DML : SQL (algèbre relationnel).

Cette caractéristique est souvent utilisée pour différencier SGBD classiques et SGBD modernes (SGBD orienté-objet, base de données déductives) [ULLMAN,88].

Dans ce cadre, la traduction d'un modèle s'effectue en deux phases : la **recherche des données** dans la base de données et la **traduction des règles** dans le langage hôte.

Ces deux phases pourraient être supportées par un générateur de code du type proposé au point précédent.

IV.2. Fonctions implémentées

A la lecture de ce mémoire, on se rend compte que la majeure partie du travail d'analyse a été consacrée à la mise en place de concepts stables pour le modèle de spécification. L'étude de cohérence fait également l'objet d'une analyse détaillée.

Pour donner un aspect plus concret à notre travail, nous avons entrepris de construire des fonctions de vérification de cohérence des règles de domaine et des règles du problème.

Avant de vérifier la cohérence d'une règle, il faut d'abord être capable d'analyser sa syntaxe. Les fonctions implémentées se diviseront donc en une analyse syntaxique et analyse de cohérence.

Pour décrire les fonctions implémentées, nous présenterons dans un premier temps, les quelques éléments théoriques sur lesquels nous nous sommes basés pour construire analyseurs syntaxiques et sémantiques.

Ensuite, dans les deux points suivants, nous analyserons respectivement le cas des règles du problème et celui des règles de domaine. Pour chacun de ces types de règles, nous proposons une syntaxe sous forme BNF (Backus-Naur-Form), pour ensuite y greffer une méthode d'analyse syntaxique (analyse syntaxique prédictive pour les règles de domaine et analyse par précédence d'opérateurs pour les règles de problème). Pour terminer, nous décrirons brièvement les modules d'analyse de règles.

IV.2.1. Eléments théoriques

Comme nous l'avons souligné au point précédent, un outil de mise-en-oeuvre du modèle de spécification devra être capable de générer du code afin de calculer les grandeurs d'un modèle. Nous entrons ainsi de plein pieds dans le domaine de la compilation [AHO,86], [UFFREDI,74].

La génération de code ne peut être envisagée à ce stade. Seule la vérification de cohérence peut être mise en oeuvre. Pour réaliser celle-ci, nous devons implémenter **la partie frontale d'un compilateur**. Elle est caractérisée par l'ensemble des phases de compilation qui dépendent principalement du langage source. Elle comprend normalement l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

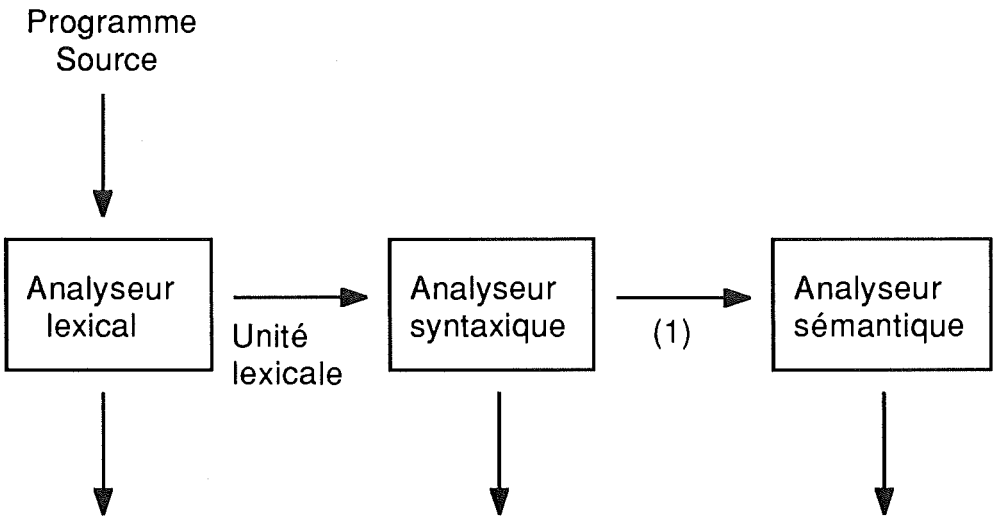
Chacune de ces phases est présente dans les fonctions implémentées. Détaillons quelque peu le rôle de chacune d'elles. La phase d'**analyse lexicale** lit les caractères formant la règle et les groupe en un flot d'unités lexicales, dont chacune représente une suite de caractères formant un tout logiquement cohérent, comme un identificateur, un mot-clé, un opérateur (+,*,/,...).

L'**analyse syntaxique** est le processus qui permet de déterminer si une suite d'unités lexicales peut être engendrée par une grammaire. Cette grammaire décrit la structure hiérarchique des expressions du langage (ici des règles du modèle).

Pour spécifier la syntaxe d'un langage, on utilise généralement des grammaires non contextuelles ou notation BNF (Backus-Naur-Form). Pour chacune des deux types de règles, nous proposerons une expression sous forme BNF.

L'**analyse sémantique** opère un certain nombre de contrôles pour assurer que l'assemblage des constituants d'une règle a un sens. Elle contrôle donc ainsi si une expression contient des erreurs sémantiques. Cette phase d'analyse contiendra notre vérification de cohérence.

On peut représenter l'interface entre ces trois composants de la manière suivante :



(1) Dans un certain nombre de cas, l'analyseur syntaxique transforme la chaîne d'entrée en une structure de données particulière, par exemple, un arbre syntaxique.

Figure 4.1. Structures générales des fonctions implémentées

Pour les deux types de règles, nous proposons une structure modulaire qui se confond avec l'architecture proposée à la figure 4.1. Nous pouvons à présent entamer les deux points suivants qui vont aborder pour chacun des deux types de règles : une notation BNF, une méthode d'analyse syntaxique propre et une structure modulaire inspirée des éléments théoriques proposés.

IV.2.2. Analyse des règles de domaine

Nous déduisons une notation BNF directement de la définition syntaxique et sémantique proposée au point II.2.1.2.2.

Rd ⟶ ident Csel

Csel ⟶ Cas | (Csel opl Csel)

Cas ⟶ (ident : ident) | (ident : ident Csel) | (:ident Cap)

Cap ⟶ Rel Ens

Ens ⟶ ident | Grddim | Fonction | ident (:ident Csel)

Grddim ⟶ ident (ident)

Fonction ⟶ Fx (ident) | Fx (Grddim)

Fx ⟶ Fannee | Fmois | Fjour | .....

opl ⟶ ou | et

Rel ⟶ > | < | = | ≥ | ≤ | <> | >< | == | in | not-in | not==

ident ⟶ lettre | lettre ident

lettre ⟶ majuscule | minuscule | _

La méthode d'analyse syntaxique utilisée porte le nom d'**analyse descendante prédictive**. On consultera [AHO,86] pour une étude détaillée et complète de cette méthode. Nous exposerons uniquement les grands principes et justifierons l'emploi de cette méthode pour l'analyse des règles de domaine.

A partir de la règle de domaine, l'analyse **descendante** effectue le construction descendante d'un arbre syntaxique. L'objectif est de construire l'arbre syntaxique d'une manière telle que la chaîne engendrée par l'arbre concorde exactement avec la chaîne d'entrée. Cette méthode lit la chaîne d'entrée de gauche à droite et construit parallèlement l'arbre syntaxique.

Suivant l'unité lexicale reconnue (appelée symbole de prévision (lookahead)), on choisit une production de la grammaire qui détermine la manière de construire l'arbre. Cette méthode est appelée **prédictive** car le symbole de prévision détermine de manière non ambiguë la production à utiliser (aucun retour en arrière ou saut en avant dans la chaîne d'entrée n'est donc nécessaire).

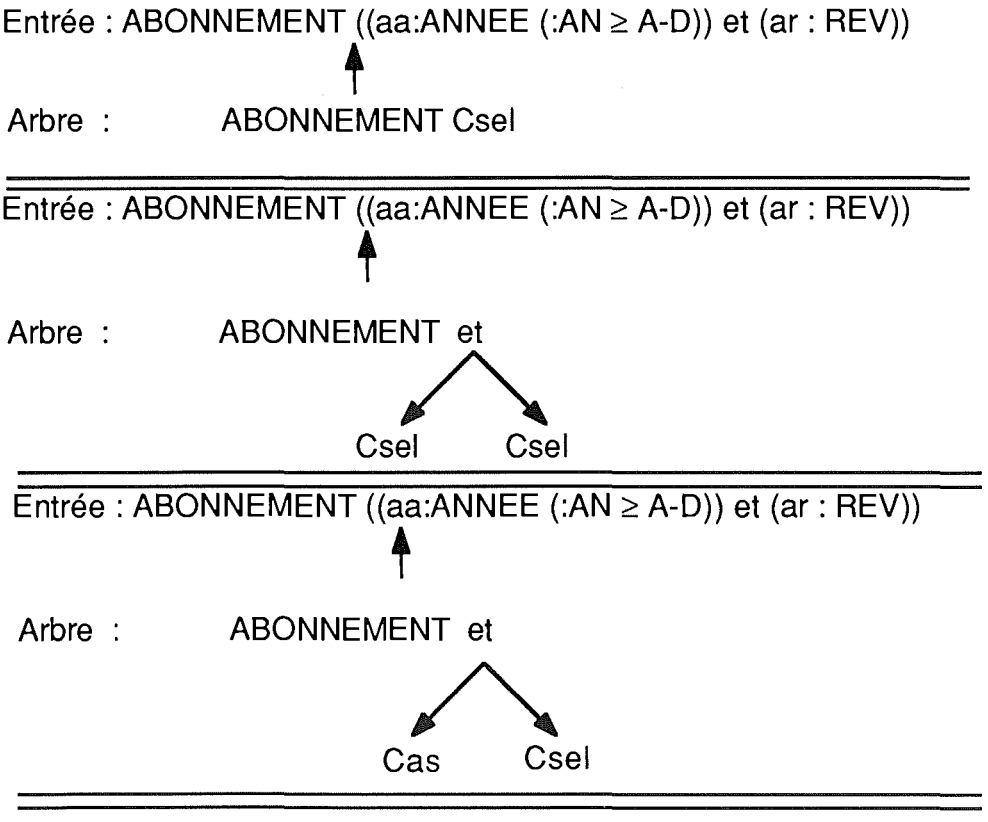
Notre grammaire est non ambiguë, car il est possible de déterminer de manière univoque le type de conditions de sélection que l'on va lire grâce au symbole de prévision. Si l'unité lexicale lue équivaut à '((', on a une expression booléenne de deux conditions de sélection. Si on lit '(:', on doit poursuivre par une condition d'association avec une condition d'appartenance et si on obtient l'unité lexicale '(lettre', on obtient une condition d'association sans condition d'appartenance.

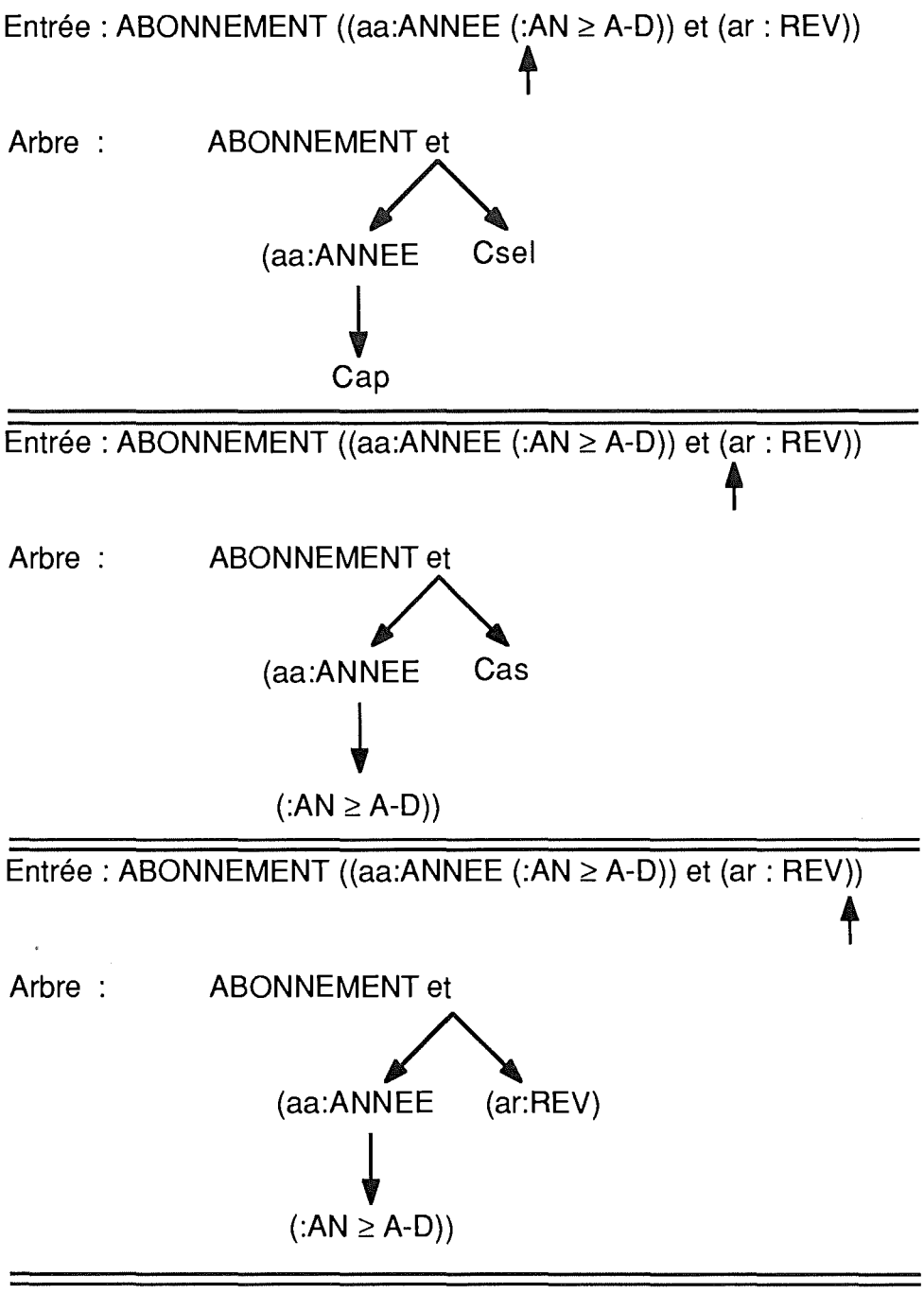
Prenons un exemple pour montrer la construction parallèle et sans équivoque de l'arbre syntaxique. L'arbre sera présenté de manière tout à fait informelle.

Exemple : (étude de cas n°1)

ABONNEMENT=((aa:ANNEE(:AN ≥ A-D)) et (ar:REV))

Dans les graphiques ci-dessus un pointeur sur la chaîne d'entrée permet de mettre en parallèle lecture et construction de l'arbre.





En adoptant la méthode d'analyse prédictive, on rassemble en une seule étape les phases d'analyse lexicale et d'analyse syntaxique. En fait, l'analyse lexicale se limite à l'accès à la prochaine unité lexicale permettant à l'analyseur syntaxique de construire l'arbre. Ce module se nomme VERIFSYNRD et a pour objectif de contrôler si une chaîne d'entrée respecte la BNF décrite ci-dessus.

Les règles de domaine sont liées au schéma de la base de données et sont donc porteuses de sémantique. Celle-ci a été définie précédemment au point II.2.1.2.2. Le module VERIFSEMRD effectue un contrôle sémantique et vérifie, par exemple, si un type d'associations appartient bien au schéma de la base de données, si l'attribut spécifié dans une condition d'appartenance appartient à l'entité citée précédemment, etc.

Un troisième module, GDV, a pour objectif de créer le graphe de dépendances entre variables entité du modèle afin de permettre la vérification de cohérence implantée par la fonction suivante. Ce module s'inspire des quatres étapes de création du GDV détaillées au point III.3.1.

IV.2.3. Analyse des règles du problème

Le modèle de spécification accepte deux types de règles de définition : celles à définition unique et celles à définitions multiples. Pour chacune d'elles, nous allons présenter leur forme BNF avant d'y greffer une méthode d'analyse syntaxique.

Runiq ⟶ Expr

Expr ⟶ Expr + Expr | Expr - Expr | Expr * Expr | Expr / Expr |
Σ Expr | Π Expr | - Expr | (Expr) | id

id ⟶ Réel | Entier | Nom

Nom ⟶ lettre | lettre Nom

lettre ⟶ majuscule | minuscule | _

Rmult ⟶ Ensdesol

Ensdesol ⟶ Expr si Cond | Expr si Cond Ensdesol

Cond ⟶ Contrainte ou Cond | Contrainte et Cond | Contrainte

Contrainte ⟶ Expr < Expr | Expr > Expr | Expr ≤ Expr | Expr ≥ Expr |
Expr = Expr

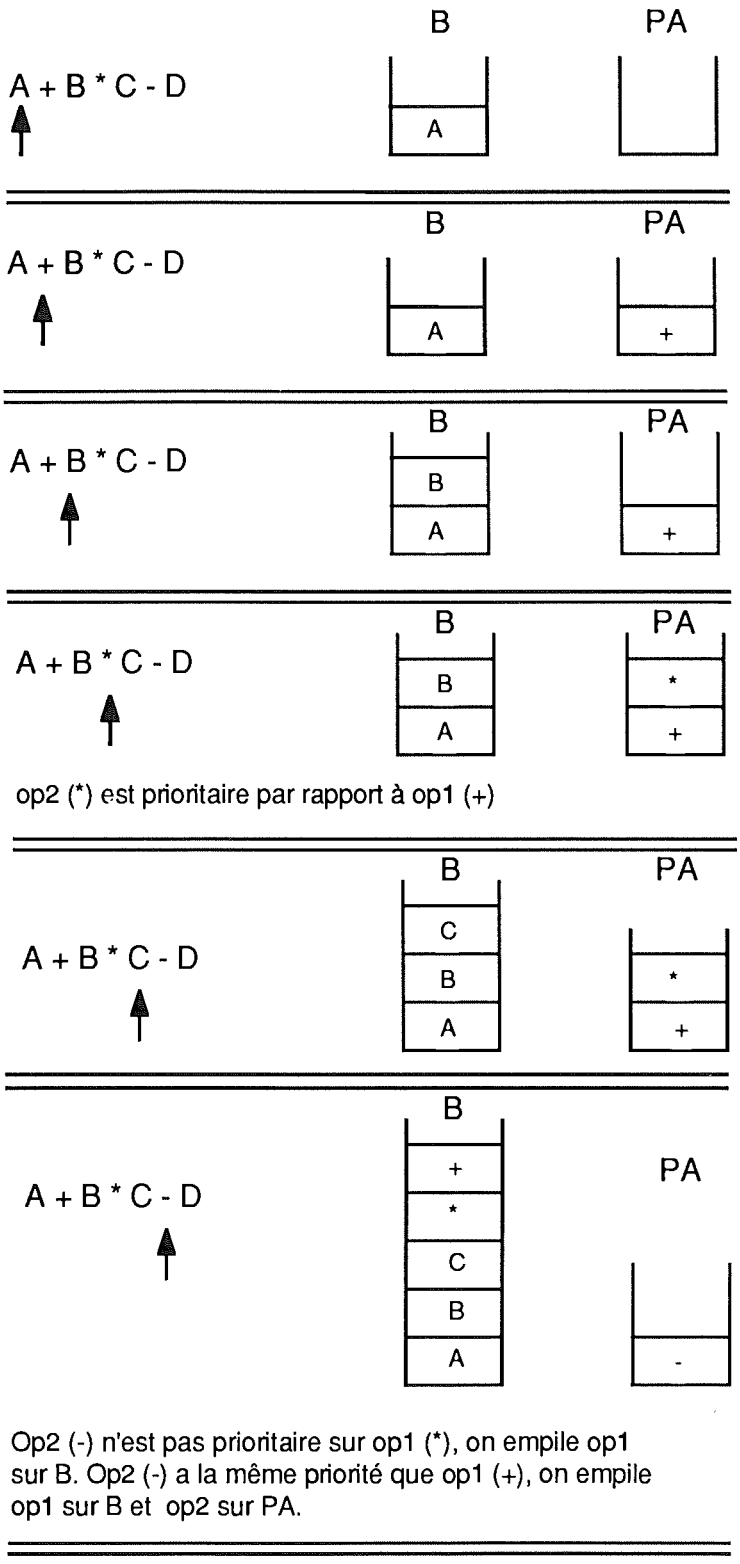
La méthode d'analyse syntaxique choisie pour les règles du problème porte le nom d'analyse par précédence d'opérateurs. On consultera [AHO,86] et [UFFREDI,74] pour une étude détaillée et complète de cette méthode.

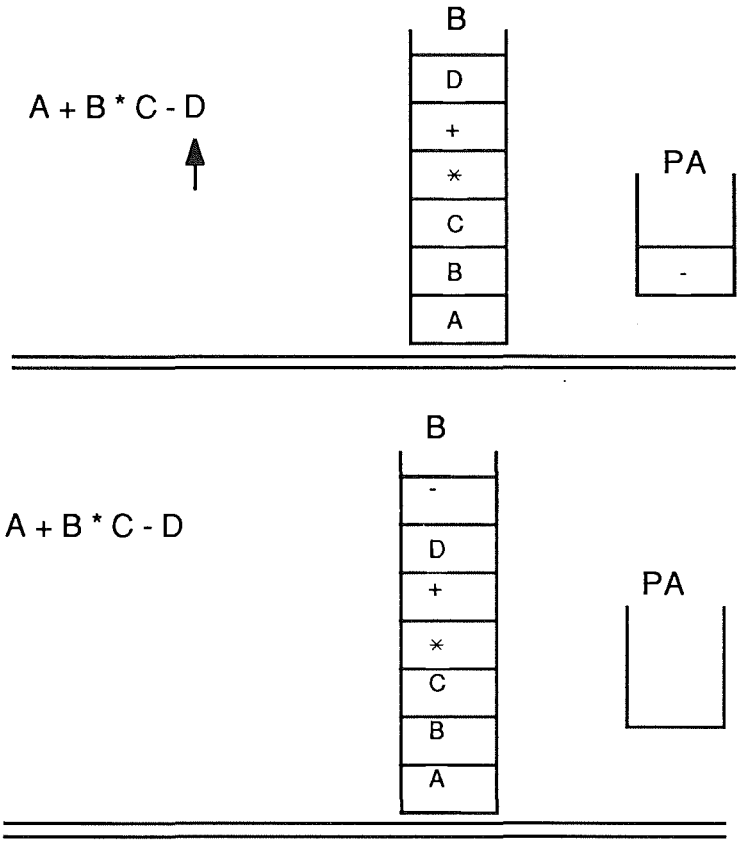
Grâce à la grammaire définie ci-dessus et à la matrice de priorité présentée au point II.2.2.1. nous pouvons réaliser une analyse de la syntaxe par précédence d'opérateurs. La matrice de priorité et le principe d'associativité à gauche définissent des relations de précédence entre opérateurs.

Expliquons brièvement le principe de l'algorithme. Supposons une expression générale E. Un pointeur se met au début de cette expression. Si le premier élément de l'expression est une grandeur, nous le plaçons dans la pile (appelée B). Par contre, s'il s'agit d'un opérateur (présent dans la matrice de priorité), on le place dans un pile temporaire PA. Chaque fois qu'on rencontre une grandeur, ont l'empile sur B. Mais, si le pointeur désigne un opérateur, on regarde la relation de précédence existant entre celui-ci (op2) et l'opérateur au-dessus de PA (op1). Si op1 est prioritaire par rapport à op2, on met op2 au dessus de B et op1 à la place de op2 sur PA. Si op2 est prioritaire par rapport à op1 ou op1 et op2 ont la même priorité, on empile op1 sur PA. A la fin de l'expression, si PA n'est pas vide, on dépile tous les éléments de PA et on les empile sur B. Lorsque l'expression est analysée, il suffit d'effectuer certaines opérations sur la pile B pour vérifier la syntaxe.

Analysons la syntaxe d'une règle très simple pour montrer la construction sans équivoque de la pile.

Exemple : On a l'expression : A + B * C - D. Pour chacune des étapes de l'analyse, nous donnons le pointeur de l'expression et le contenu des deux piles.





L'analyse lexicale lit les caractère formant une règle, les groupe en flot d'unités lexicales et crée la table des symboles. Dans la fonction, le module TRANSFORMEREGLE va effectuer cette analyse lexicale à la seule différence qu'il ne créera pas de table des symboles puisqu'elle existe déjà. En effet, avant l'analyse d'une règle, les grandeurs du modèle ont déjà été définies et stockées dans une base de données représentant le schéma Entité/Association (présenté au point II.2.7). Ce module aura comme fonction de vérifier l'appartenance des unités lexicales à la table des symboles existant déjà.

La table des symboles est stockée dans une base de données. Pour des raisons de performances et de facilités d'accès, le module CREATIONTABLE va créer une table des grandeurs et une table des dimensions mises en mémoire centrale et auxquelles l'accès est beaucoup plus rapide que s'il fallait accéder à la base de données sur un disque secondaire.

La phase d'analyse syntaxique est assurée par le module VERIFICATIONSYNTAXE. Nous avons choisi ce type d'analyse car elle est facile à implanter à cause de sa simplicité. De plus, elle s'adaptait parfaitement à la grammaire que nous avions choisie.

Pour conclure, l'analyse sémantique sera effectuée par le module VERIFICATIONREGLE qui met en pratique les mécanismes de vérification de cohérence sur les dimensions des grandeurs présentés aux points II.3.1.2., II.3.1.3. et II.3.1.4.

CONCLUSIONS

Le domaine de l'aide à la décision est à l'heure actuelle en pleine évolution. L'utilisateur dispose d'une panoplie d'outils puissants et ergonomiques qui lui permettent d'aborder des problèmes de plus en plus complexes. Face à cette complexité croissante, l'utilisateur final ne peut s'appuyer sur aucune aide méthodologique pour développer des applications de qualité. Cette absence méthodologique se fait cruellement ressentir dans un domaine où l'utilisateur-concepteur est le seul à intervenir à chaque étape du développement.

Pour combler cette carence, les premiers éléments d'un modèle de spécification de base de connaissances pour l'aide à la décision ont été proposés dans [HAINAUT,90]. A notre connaissance, mise à part cette proposition, peu de travaux ont été réalisés en la matière. Citons, cependant, les récents travaux de Karin Becker [BECKER,91]

D'abord, nous avons tenté de donner une définition formelle complète du modèle de spécification. L'analyse des concepts centraux de dimensions, grandeurs et règles semblent constituer un noyau stable dans ce processus de formalisation. Autour de cette partie centrale, d'autres concepts, tels que les contraintes d'intégrité, mériteraient une étude plus fouillée.

Ensuite, nous avons déduit de l'analyse des concepts un certain nombre de propriétés permettant de construire les graphes de dépendances entre les grandeurs d'un modèle. Ces graphes se sont révélés des outils précieux tout au long de notre travail de recherche. Une grande partie de notre étude de cohérence repose d'ailleurs sur ceux-ci.

Enfin, après une définition aussi complète que possible du modèle de spécification, nous nous sommes intéressés aux deux autres pôles de toute méthodologie : la démarche de conception et la mise en oeuvre. La démarche proposée est le résultat de notre courte expérience en développement d'applications d'aide à la décision. Elle est donc un prototype qui doit être mis à l'épreuve dans des situations réelles. Seule une confrontation avec les utilisateurs permettra de l'améliorer. Au niveau de la mise en oeuvre, au vu du travail de formalisation déjà accompli, on peut espérer à court terme la réalisation de générateurs de code facilitant l'implémentation de modèles. Quant à la construction d'outils d'aide à la conception, elle ne pourra s'envisager que dans un avenir plus lointain.

Notre travail ne constitue qu'une contribution partielle à l'élaboration d'une méthodologie complète. De nombreux points devront être étudiés : la conception de jeux de tests pour valider un modèle, la récursivité dans la définition de variables entité, les conflits entre contraintes d'intégrité du modèle et de la base de données, la relation entre variables simples et variables entité, l'élaboration d'une architecture logicielle pour la mise en oeuvre, l'utilisation du concept d'ensemble dérivé et d'héritage, ...

Pour terminer, nous aimerions souligner une perspective intéressante pour l'utilisation des types de modèles étudiés. A de nombreuses reprises, nous avons constaté une forte analogie entre la notion de base de données déductives et un système de gestion de bases de données classiques qui intégrerait la gestion de modèles. A la manière de la logique des prédicats du premier ordre dans les bases de données déductives, notre modèle de spécification pourra servir de langage de requête, de langage d'expression de contraintes d'intégrité, de règles de mise-à-jour et de règles d'inférence (qui étendent le schéma de la base de données) pour un SGBD classique.

Bibliographie

[AHO,86] Aho Alfred, Sethi Ravi, Ullman Jeffrey "COMPILATEURS : principes, techniques et outils" Interéditions, Paris, 1989

[BECKER,91], Becker Karin, François Bodart, "Generic and reusable specifications for decision support systems" , in Congrès INFORSID 91, actes du congrès, Paris, pp.137-159

[BODART,89] ,Bodart, Pigneur, "Conception assistée des applications informatiques - 1. Etude d'opportunité et analyse conceptuelle", Masson, Paris, 1989.

[CHEN,88], Chen Y.-S., "An Entity-Relationschip Approach To Decision Support And Expert Systems", in Decision Support Systems, vol.4, June1988.

[COLL,91], Coll R., Coll J.-H., Rein, "The effect of computerized decision aids on decision time and decision quality" in Information & Management, pp 75-81, vol.20, 1991.

[DATE,90], Date C.J., "An introduction to database systems", Volume 1, cinquième édition, Addison-Wesley Publishing Company, 1990.

[DITTRICH,86], Dittrich Klaus R., Kotz Angelika M., Mülle Jutta A., "An event/trigger mechanism to enforce complex consistency constraints in design databases", Sigmod Record, Volume 15, n°3, September 1986.

[ER,1988], Er M.C., "Decision support systems : a summary, problems, and futures trends" in Decision Support System, pp 355-363, vol.4, 1988.

[FLORY,85], Flory André, Bertin Marc, "Un outil d'aide à la conception d'un système de connaissances", Papier de recherche I 85/7, Gestion-Informatique-Economie, Institut d'administration des entreprises, 1985, Lyon III.

[GALLAIRE,84], Gallaire Hervé, Minker Jack, Nicolas Jean-marie, "Logic and databases : a deductive approach", Computing Survey, Volume 16, n°2, Juin 1984.

[HAINAUT,86], Hainaut J.-L., "Méthodes + Programmes : Conception assistée des applications informatiques 2. Conception de la base de données", Masson, Presses Universitaires de Namur, 1986.

[HAINAUT,86a], Hainaut J.-L, "Application de l'informatique à la résolution de problèmes", Notes de cours provisoires, FNDP, 1986.

[HAINAUT,88], Hainaut J.-L, "Introduction à la théorie relationnelle des bases de données (Notes provisoires)", FNDP, 1988.

[HAINAUT,89], Hainaut J.-L, "Bases de données et Bases de connaissances en gestion des organisations", notes du cours présenté à la cinquièmeEcole d'Automne de Bases de Données, AFCET, novembre 1989.

[HAINAUT,90],Hainaut J.-L., "Systèmes d'aide à la décision : une approche méthodologique intégrée pour l'utilisateur final" in Congrès INFORSID 90, actes du congrès Tome 2, Biarritz 90, pp. 7 à 34.

[HOLSAPPLE,87], Holsapple C.W., Whinston, "Guru : l'utilisation des systèmes experts dans l'entreprise", Les Editions d'Organisation, 1987.

[HOLSAPPLE,87a], Holsapple C.W., "Adapting demons to knowledge management environments" in Decision Support Systems, pp 289-298, vol.3, 1987.

[KEEN,78], Keen P.G.W. and Scott-Morton M.S.,"Decisions support systems : an organizational perspective", 1978.

[KEEN,87], Keen P.G.W., "Decision support systems : the next decade" in Decisions Support Systems, pp 253-265, vol.3, 1987.

[LAZIMY,87], Lazimy Rafael, "Knowledge representation and modeling support in knowledge-based systems" in Working paper 5-87-8, University of Wisconsin-Madison, May 1987.

[LAZIMY,89], Lazimy Rafael, "E2R model and object-oriented representation for data management, process modeling and decision support", in Proc. 8th Intern. Conf. on Entity-Relationship Approach, ER Institute/ACM/IEEE, October 1989.

[MULQUIN,89], Mulquin, "Infocentres, théorie et pratique", mémoire de fin d'études, FNDP, 1989.

[RIET,90], van de Riet R.P., "Introduction to de special issue on deductive and object-oriented databases", in Data & Knowledge Engineering, pp.255-261, North-Holland, 1990.

[**RONEN,89]**, Ronen Boaz, Palley Michael A., Lucas Henry C., "Spreadsheet analysis and design" in CACM, Vol.32, N°1, Jan.89.

[SPRAGUE,87], Sprague R.H., "DSS in context" in Decision Support Systems, pp 197-202, vol.3,1987.

[UFFREDI,74], de Uffredi Jean-Pierre, "Thèse : Compilation des expressions arithmétiques et optimisation", Université Claude Bernard de Lyon, n°369, 1974.

[ULLMAN,88], Ullman J., "Principles of databases and knowledge-base systems", Computer science press,1988.

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'Informatique

Année académique 1990 -1991

Contribution à une méthodologie de développement d'applications d'aide à la décision

ANNEXES

Mémoire de fin d'études présenté par

José FORTEMPS
Jean-Marc HICK

Pour l'obtention du grade de licencié et maître en informatique

Promoteur : J.- L. HAINAUT

Table des matières

ANNEXE I

ETUDES DE CAS

ANNEXE I : ETUDES DE CAS

I.1. Etude de cas n°1

I.1.1. Enoncé : étude de la rentabilité d'une revue

On considère des prêts dont bénéficient des abonnés dans une bibliothèque. Un abonné peut introduire plusieurs prêts, le même jour, concernant des exemplaires de numéros (de revues ou d'ouvrages).

On admet que la durée d'un prêt est fixée à l'avance et ne change pas d'une année à l'autre. La bibliothèque s'abonne en début d'année à chaque revue désirée et ce pour une durée d'un an. Un prêt est caractérisé par une date de début (jour de l'emprunt) et une date de fin (retour de l'emprunt).

Parmi les abonnés, certaines personnes peuvent obtenir une réduction de 50 pour cent sur le montant d'un prêt. Ces réductions concernent les étudiants, les pensionnés et les chômeurs.

Lorsqu'un abonné a ramené une revue en retard, il doit payer une amende en supplément par jour de retard. Cette amende journalière est remise à jour chaque année.

Pour chaque exemplaire, on va chaque année réajuster le montant du prêt le concernant selon son état matériel (pages détachées, état de la couverture, pages déchirées, ...). Un exemplaire endommagé aura un montant de prêt moins élevé.

On demande de confirmer par un modèle si l'abonnement à une revue est intéressant pour la bibliothèque. Cette analyse se fera sur une certaine période (à partir d'une année fixée jusqu'à l'année en cours juste avant le réabonnement).

I.1.2. Schéma de la base de données

On propose une base de données correspondant au schéma conceptuel de la figure 1.

Une entité **ABONNE** représente un abonné. Elle est caractérisée par le numéro de l'abonné (NOAB), le nom (NOM) et l'attribut RED qui indique si l'abonné bénéficie d'une réduction sur le montant des prêts. Elle est également caractérisée par les prêts dont l'abonné bénéficie (PRET via bénéficie).

Une entité **PRET** représente un prêt. Elle est caractérisée par le numéro de prêt (NOPRET), la date de début de l'emprunt (DATEDEB), la date de retour (DATERET) et l'exemplaire sur lequel porte le prêt (EXEMPLAIRE via concerne).

Une entité **EXEMPLAIRE** représente un exemplaire. Elle est caractérisée par le numéro de l'exemplaire (NOEX) et correspond à un numéro d'une revue (NUMERO via ne) ou à un volume d'un ouvrage (VOLUME via ve).

Une entité **NUMERO** représente un numéro de revue dont le numéro est (NO) et qui est caractérisée par la revue à laquelle elle appartient (REVUE via constitution).

Une entité **REVUE** représente une revue. Elle est caractérisée par le nom de la revue (TITRE).

Une entité **OUVRAGE** représente un ouvrage. Elle est caractérisée par un titre (TITREO) et un numéro d'éditeur (NOED).

Une entité **VOLUME** représente un volume. Elle est caractérisée par le numéro de volume (NOVOL) et par l'ouvrage auquel elle appartient (OUVRAGE via composition).

Une entité **AUTEUR** représente un auteur. Elle est caractérisée par un nom d'auteur (NOMAUT), par un numéro identifiant (NUMAUT) et par une suite d'ouvrages (OUVRAGE via écriture).

Une entité **ANNEE** représente une année d'exercice. Elle est caractérisée par une année (AN) et une amende (AMENDE).

Une entité **ANNEEXEM** représente le montant d'un prêt pour un exemplaire à une année donnée. Elle est caractérisée par une année (ANNEE via aae), un exemplaire (EXEMPLAIRE via ea) et un montant (MONTANT).

Une entité **ABONNEMENT** représente le prix d'abonnement pour une revue à une année donnée. Elle est caractérisée par un prix (PRIX), une année (ANNEE via aa) et une revue (REVUE via ar).

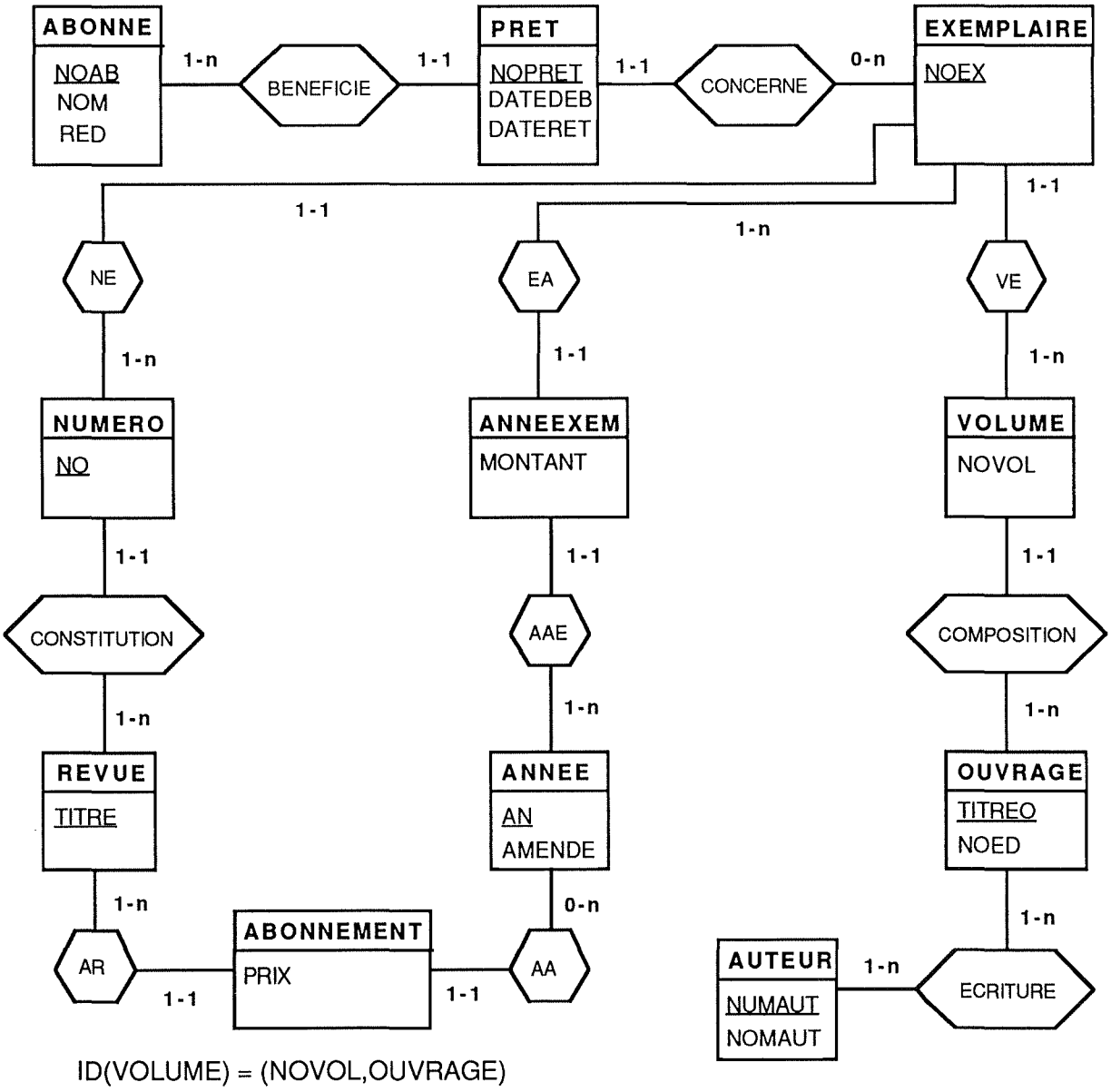


Figure 1 : Schéma E/A de la base de données d'une bibliothèque

I.1.3. Base de règles

Données

DUREE : entier ; durée maximale d'un prêt
A-D : entier ; année à partir de laquelle on veut calculer le coût d'une revue
NOMR : string ; nom de la revue analysée

Résultat

COUT-REV $_{REV}$: réel ; coût de la revue de nom NOMR depuis l'année A-D

Grandeurs internes

REV : REVUE ; revue de nom NOMR
ABO : ABONNEMENT ; abonnement concernant REV et dont l'année est supérieure ou égale à A-D
P : PRET ; prêt concernant REV après l'année A-D
A : ANNEE ; année de la date du prêt P
PA : ANNEEXEM ; montant du prêt P à la date du prêt
ABON : ABONNE ; abonné qui bénéficie du prêt P

COUT-PAIEMENT $_{REV}$: réel ; somme des coûts des abonnements à la revue REV depuis l'année A-D
RENTREE $_{REV}$: réel ; montant de tous les prêts (amendes incluses) depuis l'année A-D concernant la revue REV
RENTREE-PRET $_{P}$: réel ; montant du prêt P (amende incluse)
MONTANT-PRET $_{P}$: réel ; montant du prêt P
AMENDE-PRET $_{P}$: réel ; montant de l'amende éventuelle du prêt P
DEPASS-DATE $_{P}$: booléen ; vrai si l'exemplaire a été rapporté en retard
faux sinon
DUREE-DEPASS $_{P}$: entier ; jours de retard concernant le prêt P
REDUCTION $_{P}$: réel ; vaut 0.5 si l'abonné qui bénéficie du prêt P a droit à une réduction
vaut 1 sinon

Règles

Règles de domaine

REV = REVUE(:TITRE=NOMR)
ABO = ABONNEMENT((aa:ANNEE(:AN ≥ A-D)) et (ar:REV))
P = PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV))) et (:Fannée(DATEDEB) ≥ A-D))
A = ANNEE(:AN=Fannée(DATEDEB $_{P}$))
PA = ANNEEXEM((ea:EXEMPLAIRE(concerne:P)) et (aae:A))
ABON = ABONNE(bénéficie:P)

Remarque : Fannée est une fonction qui retire l'année d'une date.

Règles du problème

$$\text{COUT-REV}_{REV} = \text{COUT-PAIEMENT}_{REV} - \text{RENTREE}_{REV}$$

$$\text{COUT-PAIEMENT}_{REV} = \sum_{ABO} \text{PRIX}_{ABO}$$

$$\text{RENTREE}_{REV} = \sum_{P} \text{RENTREE-PRET}_{P}$$

$$\text{RENTREE-PRET}_{P} = \text{MONTANT-PRET}_{P} + \text{AMENDE-PRET}_{P}$$

$$\text{MONTANT-PRET}_{P} = \text{MONTANT}_{PA} * \text{REDUCTION}_{P}$$

$$\text{REDUCTION}_{P} = \begin{cases} 0.5 & \text{si RED}_{ABON} = \text{vrai} \\ 1 & \text{si RED}_{ABON} = \text{faux} \end{cases}$$

$$\text{AMENDE-PRET}_{P} = \begin{cases} 0 & \text{si DEPASS-DATE}_{P} = \text{faux} \\ \text{DUREE-DEPASS}_{P} * \text{AMENDE}_{A} & \text{si DEPASS-DATE}_{P} = \text{vrai} \end{cases}$$

$$\text{DEPASS-DATE}_{P} = \begin{cases} \text{vrai} & \text{si DATERET}_{P} - \text{DATEDEB}_{P} > \text{DUREE} \\ \text{faux} & \text{si DATERET}_{P} - \text{DATEDEB}_{P} \leq \text{DUREE} \end{cases}$$

$$\text{DUREE-DEPASS}_{P} = \text{DATERET}_{P} - \text{DATEDEB}_{P} - \text{DUREE}$$

I.2. Etude de cas n°2

I.2.1. Enoncé : établissement du quota de vente des représentants d'une région et du budget publicitaire alloué à celle-ci

Une société commerciale dispose d'un certain nombre de représentants. Ceux-ci sont répartis en région et s'occupent de clients. Ces représentants mettent en vente toute la gamme de produits disponibles pour la région.

Un client appartient à une région et est desservi par un et un seul représentant. Il passe des commandes au représentant. Chaque client commande les produits disponibles dans sa région.

Ces commandes portent sur des produits mis en vente dans la région. Chaque produit est caractérisé par un prix propre à la région où il est mis en vente.

On dispose également des quotas de vente estimés pour les nouveaux produits disponibles dans une région à partir de l'année en cours.

Au début de chaque année, la société désire fixer un quota de vente pour chacun de ses représentants et un budget publicitaire pour la région. Ces valeurs seront calculées sur une période d'analyse. La détermination de la publicité et des quotas était précédemment effectuée par des experts. C'est cette expertise qui a été traduite sous forme de règles.

I.2.2. Schéma de la base de donnée

On propose une base de données correspondant au schéma conceptuel de la figure 2.

Une entité **REPRESENTANT** représente un représentant de la société. Elle est caractérisée par un numéro (NOREP), un nom (NOM) et le quota fixé au début de l'année précédente (QUOTA).

Une entité **CLIENT** représente un client de la société. Elle est caractérisée par un numéro de client (NOCLI), un nom (NOM), un représentant qui en a la responsabilité (REPRESENTANT via rc) et un région où il habite (REGION via cr).

Une entité **REGION** représente une région où la société commercialise ses produits. Elle est caractérisée par un nom (NOM), un taux de croissance économique (CROISSANCE) et un taux de chômage (CHOMAGE).

Une entité **COMMANDE** représente une commande d'un client à la société. Elle est caractérisée par un numéro de commande (NOCOM) et le client qui l'a effectuée (CLIENT via cc).

Une entité **PRODUIT** représente un produit mis en vente par la société. Elle est caractérisée par un numéro de produit (NOPRO) et un libellé (LIBELLE).

Une entité **LIGNECOM** représente une ligne de commande. Elle est caractérisée par la quantité du produit commandé (QUANTITE), la commande à laquelle elle se rapporte (COMMANDE via lc) et le produit commandé (PRODUIT via pl).

Une entité **VENTEREG** représente une vente d'un produit dans une région. Elle est caractérisée par la région où la vente a lieu (REGION via rv), le produit vendu (PRODUIT via vp), le prix du produit dans la région (PRIX) et l'année du début de la vente (ANDEP).

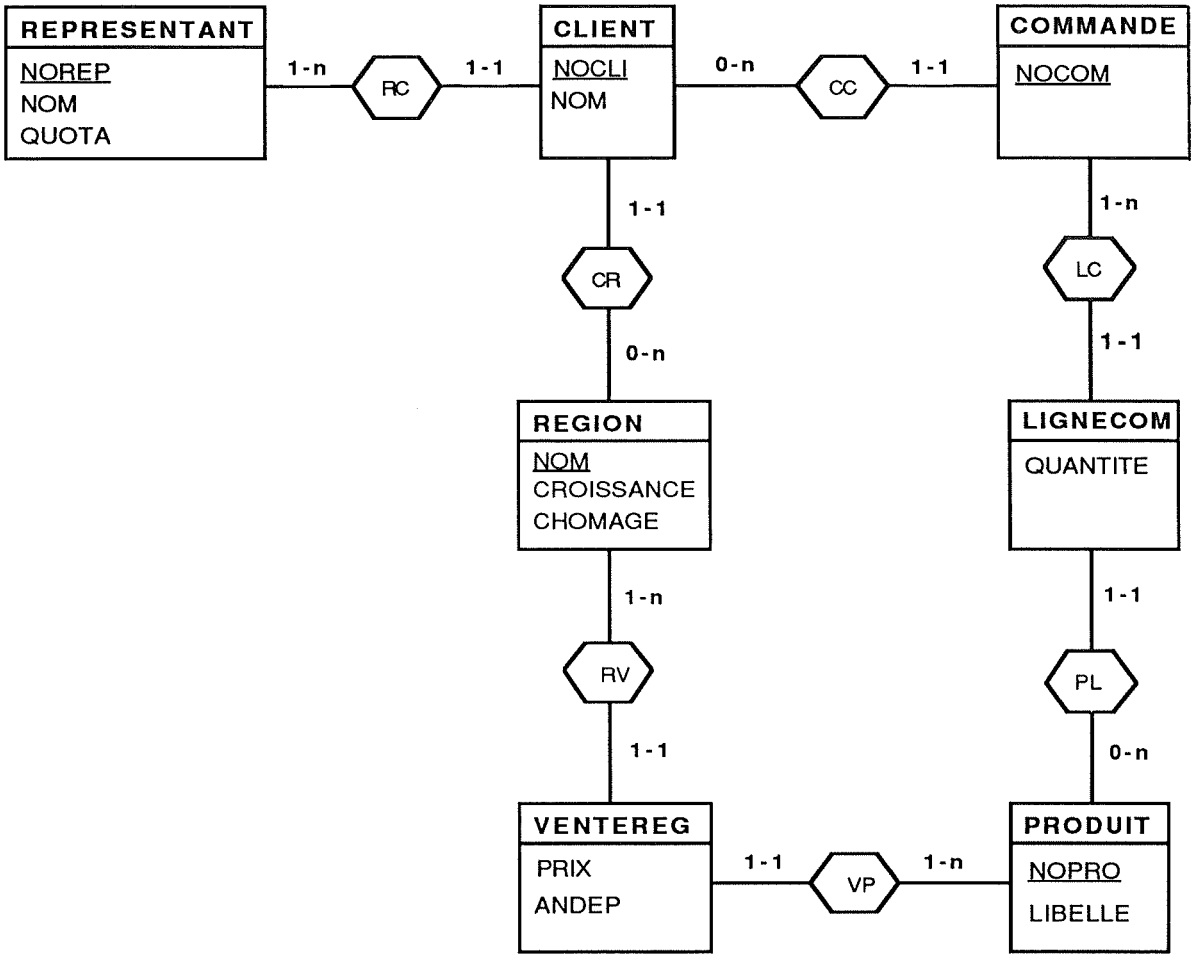


Figure 2 : Schéma E/A de la base de données des ventes d'une région

I.2.3. Base de règles

Données

NOMREG : string ; nom de la région analysée
AN : entier ; année d'établissement du nouveau quota et du budget
QUOTANOUVPRO $_{PRO}$: entier ; quota de vente du nouveau produit PRO

avec ACQUISITION(PRO)

Résultats

PUB $_{REG}$: réel ; budget publicitaire alloué à la région de nom NOMREG
NOUVQUOTA $_{REP}$: réel ; nouveau quota de vente pour le représentant REP appartenant à la région de nom NOMREG

Grandeurs internes

REG : REGION ; région de nom NOMREG
REP : REPRESENTANT ; représentant de la région REG
CLI : CLIENT ; client du représentant REP
PRO : PRODUIT ; produit mis en vente à partir de l'année AN
COM : COMMANDE ; commande faite par le client CLI
LC : LIGNECOM ; ligne de la commande COM
VR : VENTEREG ; prix du produit vendu dans la région REG

FACTEURECO $_{REG}$: réel ; facteur économique de la région REG
ECONOMIE $_{REG}$: (bonne,moyenne,mauvaise) ; état de l'économie de la région REG
FACTEURPUB $_{REG}$: réel ; facteur représentant l'impact publicitaire dans la région REG
BASECALCUL $_{REP}$: réel ; base de calcul du nouveau quota de REP en fonction des ventes de l'année (AN-1) et des nouveaux produits PRO
BASE1 $_{REP}$: réel ; base de calcul du nouveau quota de REP en cas d'augmentation des ventes de l'année (AN-1)
BASE2 $_{REP}$: réel ; base de calcul du nouveau quota de REP en cas de non augmentation significative des ventes de l'année (AN-1)
NBREP $_{REG}$: entier; nombre de représentants de la région REG
SUPNOUVPRO $_{REG}$: réel ; supplément de vente grâce aux nouveaux produits PRO pour la région REG
VENTES $_{REP}$: réel ; total des ventes du représentant REP
MONTANT-CLI $_{CLI}$: réel ; total des achats du client CLI
MONTANT-COM $_{COM}$: réel ; montant de la commande COM
MONTANT-LC $_{LC}$: réel ; montant de la ligne LC

Règles

Règles de domaine

REG = REGION(:NOM=NOMREG)
REP = REPRESENTANT(rc:CLIENT(cr:REG))
PRO = PRODUIT(vp:VENTEREG((:ANDEP=AN) et (rv:REG)))
CLI = CLIENT(rc:REP)
COM = COMMANDE(cc:CLI)
LC = LIGNECOM(lc:COM)
VR = VENTEREG(vp:PRODUIT(pl:LC))

Règles du problème

$$PUB_{REG} = FACTEURECO_{REG} * \sum_{REP} NOUVQUOTA_{REP}$$

FACTEURECO $_{REG}$ = CROISSANCE $_{REG}$ si ECONOMIE $_{REG}$ = bonne
CROISSANCE $_{REG}$ /3 si ECONOMIE $_{REG}$ = moyenne
CROISSANCE $_{REG}$ /5 si ECONOMIE $_{REG}$ = mauvaise

$ECONOMIE_{REG}$ = bonne si $CROISSANCE_{REG} \geq 0.04$ ou $CHOMAGE_{REG} < 0.076$
moyenne si $0.02 \leq CROISSANCE_{REG} < 0.04$ et $0.076 \leq CHOMAGE_{REG} < 0.082$
mauvaise si $CROISSANCE_{REG} < 0.02$ ou $CHOMAGE_{REG} \geq 0.082$

$NOUVQUOTA_{REP} = (1+(FACTEURPUB_{REG} + FACTEURECO_{REG})/2) * BASECALCUL_{REP}$

$FACTEURPUB_{REG}$ = $PUB_{REG}/100000$ si $PUB_{REG} > 7000$ et $ECONOMIE_{REG} <>$ mauvaise
-0.015 si $PUB_{REG} < 5000$ et $ECONOMIE_{REG} <>$ bonne
0 sinon

$BASECALCUL_{REP}$ = $BASE1_{REP}$ si $VENTES_{REP} > 1.15 * QUOTA_{REP}$
$BASE2_{REP}$ si $VENTES_{REP} \leq 1.15 * QUOTA_{REP}$

$BASE1_{REP} = QUOTA_{REP} + (VENTES_{REP} - 1.15 * QUOTA_{REP}) + SUPNOUVPRO_{REG}/NBREP_{REG}$

$BASE2_{REP} = QUOTA_{REP} + SUPNOUVPRO_{REG}/NBREP_{REG}$

$NBREP_{REG}$ = Taille(REP)

$SUPNOUVPRO_{REG} = \sum_{PRO} PRIX_{PRO} * QUOTANOUVPRO_{PRO}$

$VENTES_{REP} = \sum_{CLI} MONTANT\text{-}CLI_{CLI}$

$MONTANT\text{-}CLI_{CLI} = \sum_{COM} MONTANT\text{-}COM_{COM}$

$MONTANT\text{-}COM_{COM} = \sum_{LC} MONTANT\text{-}LC_{LC}$

$MONTANT\text{-}LC_{LC} = QUANTITE_{LC} * PRIX_{VR}$

I.3. Etude de cas n°3 [HAINAUT,90]

I.3.1. Enoncé : étude du coût d'une liaison aérienne

On propose de construire la base de connaissances d'un système d'aide à la décision qui permet d'étudier les coûts de vols aériens. Ce système doit aider le service de planification des vols tant en ce qui concerne les choix à court terme (tels que les charges utiles à transporter et les quantités de carburant à acheter à chaque escale) que les choix à moyen terme (relatifs, par exemple, au choix des escales d'un vol ou des appareils). Le domaine d'application se présente comme suit.

On considère des vols aériens dont chacun relie deux villes en passant par un certain nombre d'escales, qui sont des aéroports. Un même vol peut être effectué à des dates différentes par des appareils différents. Un appareil est caractérisé par la capacité de ses réservoirs (en kilos de carburant) ainsi que la consommation à vide, équipage compris (en kilos de carburant par Km).

On connaît aussi sa charge utile maximale et sa consommation supplémentaire par kilo de charge (en kilos de carburant par kilo de charge et par km). Il est à noter cependant que la consommation à vide n'inclut pas le transport du carburant lui-même. On admet que la charge utile (fret et passagers) est constante pour toute la durée du vol, mais qu'elle peut varier d'une date à l'autre. On connaît la longueur de chaque tronçon du vol, c'est-à-dire la distance entre deux villes ou escales consécutives de ce vol. On supposera que la consommation en vol est une fonction linéaire de la charge emportée (charge utile + carburant).

A chaque escale, l'appareil est ravitaillé en carburant. Celui-ci est acheté au tarif local (en dollars par kilo). Le tarif local dépend de la date.

Lorsque l'appareil atterrit, ainsi que dans l'aéroport de départ, ses réservoirs peuvent contenir une quantité résiduelle non consommée lors du parcours du tronçon précédent. Pour effectuer le tronçon suivant, il est généralement nécessaire d'ajouter au réservoir une quantité qui permet d'atteindre l'escale ou la ville suivante. Il est cependant possible d'emporter une quantité supérieure à ce qui est strictement nécessaire. Ce supplément peut être intéressant si le tarif local est particulièrement bas, et si le tronçon suivant n'est pas trop long. On fera l'hypothèse que la valeur financière d'une quantité résiduelle est à calculer au tarif de l'endroit où cette quantité est observée, donc à l'atterrissage (= valeur de revente). On notera que l'appareil est présumé avoir consommé la quantité résiduelle à l'aéroport de départ, et qu'il faut donc lui imputer, mais qu'il n'a pas consommé celle qui subsiste après l'atterrissage final, et qu'il ne faut donc pas lui imputer puisqu'elle n'aura pas servi au vol.

I.3.2. Schéma de la base de données

On propose une base de données correspondant au schéma conceptuel de la figure 3.

Une entité **APPAREIL** représente un modèle d'appareil. Elle est caractérisée par la dénomination du modèle (MODELE), la capacité maximale des réservoirs (CAP-RESERVOIR), la consommation en vol d'un appareil à vide, équipage compris, mais sans passagers, ni fret, ni carburant, exprimée en kilos de carburant par km (CONS-VIDE), la consommation en vol nécessaire au transport d'un kilo de charge sur un km (CONS-CHARGE), la charge maximale qu'un appareil peut emporter (CH-UT-MAX).

Une entité **VOL-TYPE** représente un type de vol caractérisé par le code identifiant du vol-type (ID-VOL), la suite des tronçons dont le type de vol est constitué (TRONCON-TYPE via comprend).

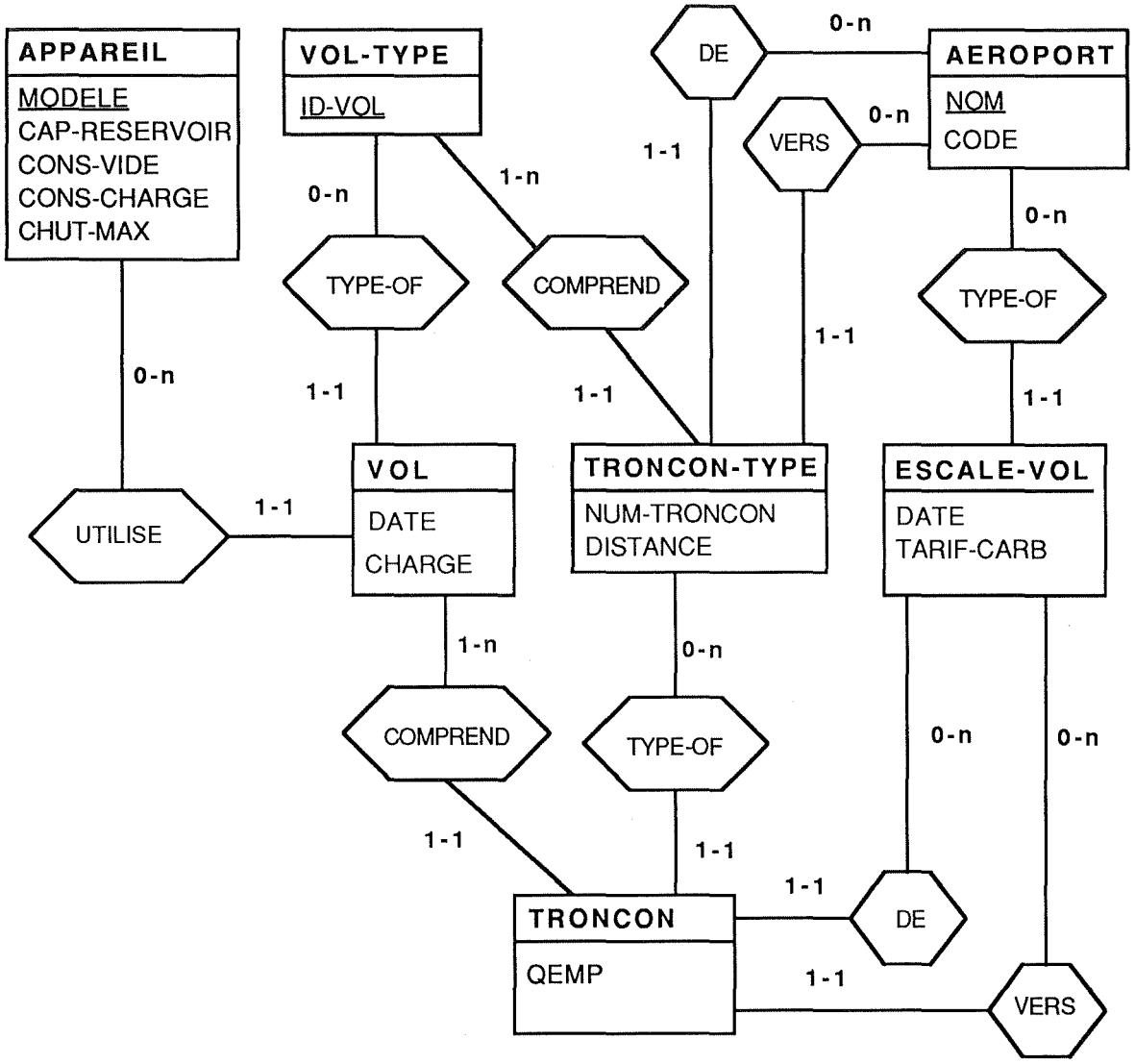
Une entité **AEROPORT** représente un aéroport. Elle est caractérisée par le nom en clair de l'aéroport (NOM), le nom de code de l'aéroport (CODE), les tronçons dont l'aéroport est le départ (TRONCON- TYPE via de), les tronçons dont l'aéroport est l'arrivée (TRONCON-TYPE via vers).

Une entité **TRONCON-TYPE** représente une section ininterrompue d'un vol-type. Elle est caractérisée par son type de vol (VOL-TYPE via comprend), le numéro de séquence du tronçon dans le type de vol (NUM-TRONCON), la longueur du tronçon en km (DISTANCE), l'aéroport de départ du tronçon (AEROPORT via de), l'aéroport d'arrivée du tronçon (AEROPORT via vers).

Une entité **VOL** représente un vol réel effectué à une date déterminée. Elle correspond à une entité VOL-TYPE qui en définit les propriétés stables. Elle est caractérisée par le type du vol (VOL-TYPE), le modèle de l'appareil effectuant le vol (APPAREIL via utilise), la date du vol (DATE), la charge utile transportée (CHARGE), les tronçons dont le vol est constitué (TRONCON via comprend).

Une entité **ESCALE-VOL** représente une escale constituée d'un aéroport à une date déterminée auquel tout appareil d'un vol peut atterrir ou décoller. Elle est caractérisée par l'aéroport (AEROPORT), la date à laquelle l'aéroport est considéré (DATE), le tarif du carburant dans cet aéroport à cette date, en $US, par kilo (TARIF-CARB), les tronçons dont l'escale est le départ (TRONCON via de), les tronçons dont l'escale est l'arrivée (TRONCON via vers).

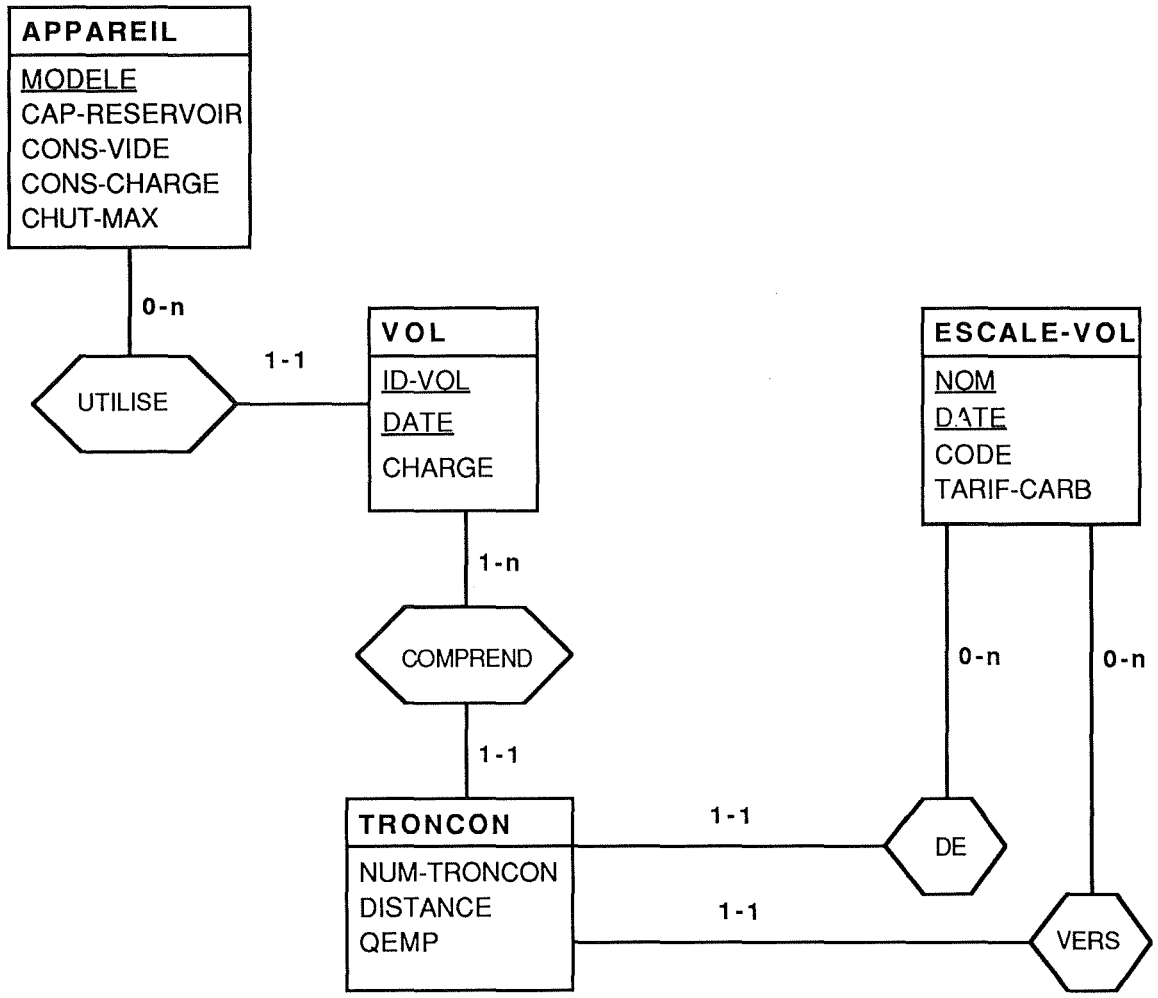
Une entité **TRONCON** représente une section effectué par un vol réel. Elle correspond à une entité TRONCON-TYPE qui en définit les propriétés stables. Elle est caractérisée par le type de tronçon (TRONCON-TYPE), le vol réel durant lequel le tronçon a été effectué (VOL via comprend), la quantité de carburant emportée (achetée) à l'escale de départ du tronçon (QEMP), l'escale de départ du tronçon (ESCALE-VOL via de), l'escale d'arrivée du tronçon (ESCALE-VOL via vers).



ID(TRONCON-TYPE) = (VOL-TYPE,NUM-TRONCON)
ID(VOL) = (VOL-TYPE,DATE)
ID(ESCALE-VOL) = (AEROPORT,DATE)
ID(TRONCON) = (VOL,TRONCON-TYPE)

Figure 3 : Schéma E/A de la base de données des vols aériens

Afin de simplifier l'élaboration du modèle de calcul du coût des vols, on dérivera du schéma de la figure 3 une vue obtenue par application des règles d'héritages aux types d'associations **type-de**, puis par abandon des types d'entités sources de cet héritage. Cette vue est représentée par la figure 4.



ID(TRONCON) = (VOL,NUM-TRONCON)

Figure 4 : Vue E/A de la base de données des vols aériens

I.3.3. La base de règles

On admet que la quantité résiduelle au départ d'un vol (grandeur QRES-INIT) est donnée par l'utilisateur du modèle. En outre, le modèle ne devant évaluer que le coût d'un seul vol, on a choisi de ne pas dimensionner COUT-VOL selon V.

Données

ID: entier ; numéro du vol
D : date ; date du vol
QRES-INIT $_V$: réel,en kg ; quantité résiduelle de carburant à l'aéroport de départ du vol

avec ACQUISITION(V)

Résultat

COUT-VOL $_V$: réel,en $; coût du vol numéro ID à la date D

Grandeurs internes

V : VOL ; vol dont on calcule le coût
APP : APPAREIL ; appareil effectuant le vol V
T : TRONCON ; tronçons du vol V
N : entier ; nombre de tronçons du vol V
I : entier ; numéro d'un tronçon d'un vol
T(I) : TRONCON ; tronçon numéro I du vol V
E : ESCALE-VOL ; escale de départ d'un tronçon du vol V
COUT-INIT $_V$: réel,en $; valeur du carburant résiduel à l'escale de départ du vol V
VALEUR-RES : réel,en $; valeur du carburant résiduel à l'escale d'arrivée du vol V du tronçon T
QINIT $_T$: réel,en kg ; quantité de carburant résiduel avant achat à l'escale de départ du tronçon T
QDEPART $_T$: réel,en kg ; quantité de carburant au décollage du tronçon T
QRES $_T$: réel,en kg ; quantité de carburant résiduel à l'arrivée du tronçon T
QCONS $_T$: réel,en kg ; quantité du carburant consommée durant le tronçon T
QCONS/km $_T$: réel,en kg ; quantité du carburant consommée par km durant le tronçon T
CHARGE-MOY $_T$: réel,en kg ; poids total moyen de l'appareil durant le tronçon T
QMOY $_T$: réel, en kg ; quantité moyenne de carburant durant le tronçon

Règles

Règles de domaine

V = VOL((:ID-VOL=ID) et (DATE=D))
APP = APPAREIL(utilise:V)
T = TRONCON(comprend:V)
N = taille(T)
T(I) = T(:NUM-TRONCON=I), (I = 1..N)
E = ESCALE-VOL(de:T)

Règles du problème

$$\text{COUT-VOL}_V = \text{COUT-INIT}_V + \sum_T \text{COUT-ESC}_T - \text{VALEUR-RES}$$

COUT-INIT $_V$ = QRES-INIT $_V$ * TARIF $_{E(1)}$
VALEUR-RES = QRES $_{T(N)}$ * TARIF $_{E(N)}$
COUT-ESC $_T$ = QEMP $_T$ * TARIF $_E$
QRES $_T$ = QINIT $_T$ + QEMP $_T$ - QCONS $_T$
QINIT $_{T(1)}$ = QRES-INIT $_V$
QINIT $_{T(I)}$ = QRES $_{T(I-1)}$, (I = 2..N)
QCONS $_T$ = QCONS/km $_T$ * DISTANCE $_T$
QCONS/km $_T$ = (CONS-VIDE $_{APP}$ + CONS-CHARGE $_{APP}$ * CHARGE-MOY $_T$)
CHARGE-MOY $_T$ = CHARGE $_V$ + QMOY $_T$
QMOY $_T$ = (QDEPART $_T$ + QRES $_T$) / 2
QDEPART $_T$ = QINIT $_T$ + QEMP $_T$

ANNEXE II

IMPLEMENTATION DE LA PREMIERE ETUDE DE CAS DANS GURU

ANNEXE II : Implémentation de la première étude de cas dans le logiciel intégré GURU

II.1. Première phase : élaboration de la maquette

Pour commencer, il faut choisir la position de chaque grandeur dans la feuille électronique. On regroupe les données en haut à gauche du tableau. La grandeur DUREE se voit assigner la cellule E2, A-D la cellule E3 et NOMR la cellule E4. Le résultat COUT-REV $_{REV}$ est placée à la cellule E5. On assigne aux grandeurs internes dimensionnées des colonnes, chaque ligne correspondant à une valeur de la dimension. Ainsi la grandeur AMENDE $_{A}$ aura ces valeurs sur la colonne W à partir de la cellule W3. Les grandeurs internes COUT-PAIEMENT $_{REV}$ et RENTREE $_{REV}$ sont le résultat de l'agrégation d'autres grandeurs. On leur assignera respectivement les cellules E11 et E12.

La figure 5 représente la maquette du tableau correspondant au modèle de l'étude de cas. Pour des raisons de présentation, la feuille a été scindée en deux.

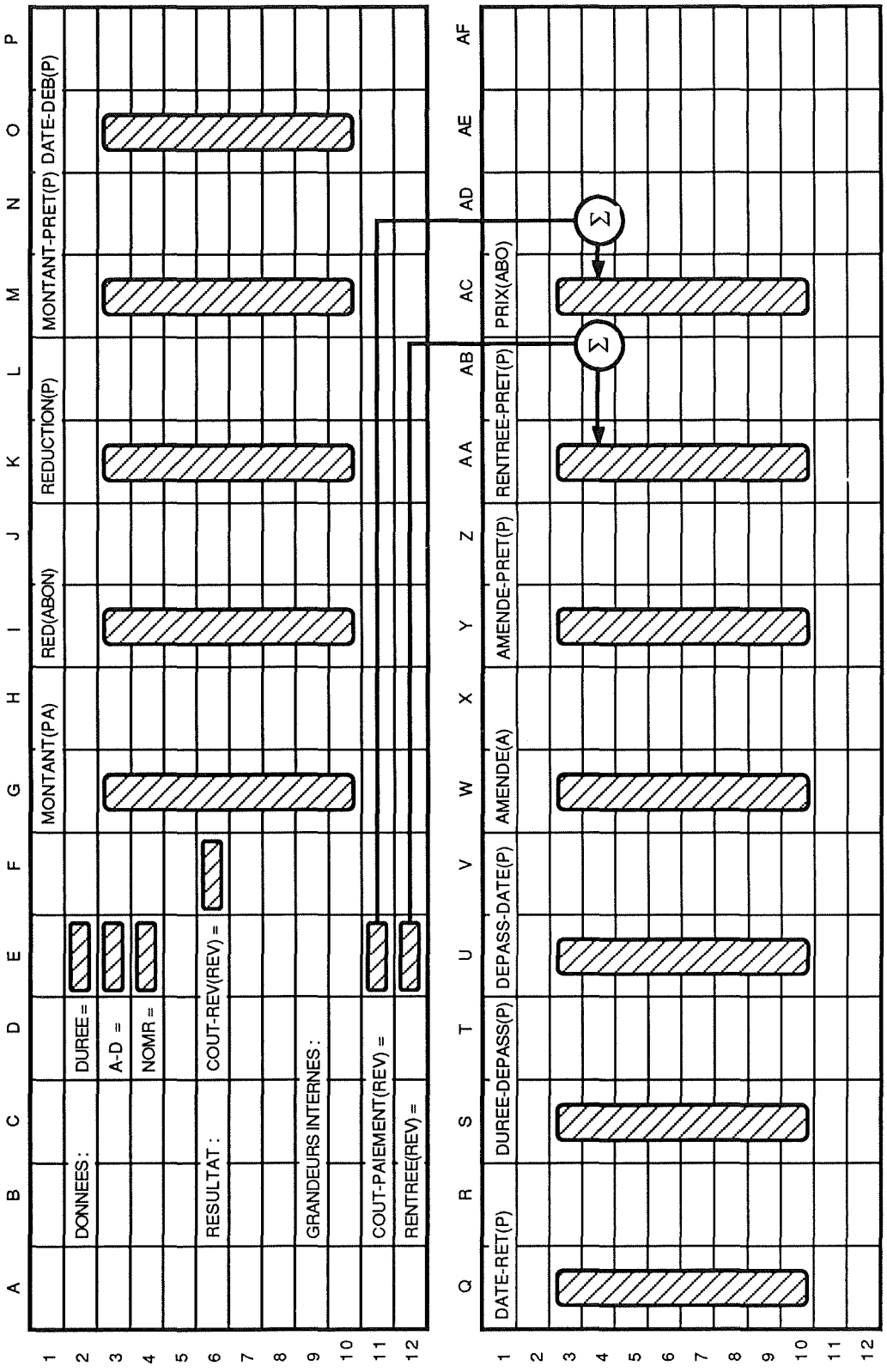


Figure 5 : Schéma du tableau de l'étude de cas n°1

II.2. Deuxième phase : Mise en place des données de la base de données

Cette phase consiste à construire des macro permettant d'importer des données de la base de données dans le tableau. Dans Guru, le gestionnaire de bases de données est du type de SQL (langage de requête relationnel).

Avant de donner la liste des fonctions (chacune correspondant à une données de la base de données à importer), nous allons détailler le programme qui importent les valeurs de PRIX $_{ABO}$ dans le tableau.

```
!PRIX ABO
!ABO=ABONNEMENT((aa:ANNEE(:AN≥A-D)) et (ar:REV))
      convert prix \
      from abonnement \
      for  titre=NOMR and \
         an>=A-D \
      to cell #AC2
      return
```

Le texte précédé d'un point d'exclamation est le commentaire du programme. La primitive **convert** permet d'importer l'attribut PRIX du type d'entité ABONNEMENT (**from**) en vérifiant la condition de sélection (**titre=NOMR and an≥A-D**). Le **convert** joue le même rôle que le **select** dans une requête SQL. Cette suite de valeurs est importée vers la cellule AC2 et est disposée en colonne.

Voici la liste des autres programmes utilisés pour importer les données de la base :

```
! MONTANT PA
! PA=ANNEEXEM((ea:EXEMPLAIRE(concerne:P)) et
!                    (aae:ANNEE(:AN=Fannée(DATEDEB P))))
      convert anneexem.montant \
      from pret, from anneexem \
      for noexemp=anneexem.noex and \
         tonum(substr(datedeb,7,4))>=AD and \
         tonum(substr(datedeb,7,4))=anneexem.an \
      from exemplaire \
      for noexemp=exemplaire.noex, \
      from numero \
      for exemplaire.no=numero.no and \
         numero.titre=NOMR \
      to cell #G3
      return

!RED ABON
!ABON=ABONNE(beneficie:P)
      convert abonne.red \
      from pret, from abonne \
      for noab=abonne.noab and \
         tonum(substr(datedeb,7,4))>=AD \
      from exemplaire \
      for exemplaire.noex=noexemp, \
      from numero \
      for exemplaire.no=numero.no and \
```

```
        numero.titre=NOMR \
    to cell #I3
    return

!DATERET P
!P=PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV)))
!           et (:DATEDEB >= A-D))
    convert dateret from pret \
    for tonum(substr(datedeb,7,4))>=A-D \
    from exemplaire \
    for noexemp=exemplaire.noex, \
    from numero \
    for numero.no=exemplaire.no and \
        numero.titre =NOMR \
    to cell #Q3

!DATEDEB P
!P=PRET((concerne:EXEMPLAIRE(ne:NUMERO(constitution:REV)))
!           et (:DATEDEB >= A-D))
    convert datedeb from pret \
    for tonum(substr(datedeb,7,4))>=AD \
    from exemplaire \
    for noexemp=exemplaire.noex, \
    from numero \
    for numero.no=exemplaire.no and \
        numero.titre =NOMR \
    to cell #O3
    return

!AMENDE A
!A=ANNEE(:AN=Fannée(DATEDEB P))
!
    convert annee.amende \
    from pret, from annee \
    for annee.an=tonum(substr(datedeb,7,4)) and \
        tonum(substr(datedeb,7,4))>= AD \
    from exemplaire \
    for noexemp=exemplaire.noex, \
    from numero \
    for exemplaire.no=numero.no and \
        numero.titre=NOMR \
    to cell #W3
    return
```

II.3. Troisième phase : Traduction des règles

La deuxième phase consiste à traduire les règles du problèmes. La syntaxe du tableur utilisé en Guru permet un certain nombre d'opérations simples (somme, division, ...). Toutefois, pour les règles plus complexes, nous avons construit des macro. Avant d'en donner la liste, nous allons détailler la fonction calculant la grandeur REDUCTION P.

```
!REDUCTION P = 0.5 si RED ABON=vrai
!                  1   si RED ABON=faux
    let li = 3
    while ((#(li,9)=vrai) or (#(li,9)=faux)) do
        let #(li,11) = 1;
        if #(li,9) = VRAI
          let #(li,11) = 0.5;
        endif;
        let li = li + 1;
    endwhile
```

Li est un compteur initialisé à 3. Tant que la cellule (correspondant à la colonne 9 et à la ligne Li) contient la valeur VRAI ou la valeur FAUX, on assigne la valeur O,5 à la cellule correspondant de la colonne 11 (REDUCTION P) si la cellule de la colonne 9 est égale à VRAI. On lui assigne la valeur 1 sinon. La macro se construit de manière presque analogue à une programme Pascal itératif.

```
!DEPASS-DATE P = DATERET P - DATEDEB P > DUREE
    let li=3
    while #(li,17) <> "" do
        let ddebut = #(li,15)
        let dfin = #(li,17)
        let #(li,21) = FAUX
        if tojul(dfin)-tojul(ddebut) > DUREE
          let #(li,21) = VRAI
        endif
        let li=li+1
    endwhile

!DUREE-DEPASS P = DATERET P - DATEDEB P - DUREE
    let li=3
    while #(li,17) <> "" do
        let ddebut = #(li,15)
        let dfin = #(li,17)
        let #(li,19)=0
        if #(li,21)=VRAI
          let #(li,19)=tojul(dfin)-tojul(ddebut)-DUREE
        endif
        let li=li+1
    endwhile

!AMENDE-PRET P = 0          si DEPASS-DATE P = faux
!                  DUREE-DEPASS P * AMENDE A
!                       si DEPASS-DATE P =vrai
    let li = 3
```

```
        while #(li,23) > 0 do
            let #(li,25) = #(li,19) *  #(li,23)
            let li = li + 1
        endwhile

!MONTANT-PRET  P = MONTANT PA * REDUCTION P
!RENTREE-PRETP =  MONTANT-PRETP +AMENDE-PRETP
        let li = 3
        while #(li,23) > 0 do
            let #(li,13) = #(li,11) * #(li,7)
            let #(li,27) = #(li,13) + #(li,25)
            let li = li + 1
        endwhile
        return
```

ANNEXE III

FONCTIONS IMPLEMENTEES

ANNEXE III : FONCTIONS IMPLEMENTEES

III.1. Analyse des règles de domaine

III.1.1. Analyse syntaxique des règles de domaine : Module VERIFSYNRD

```
(*-------------------------------------------------------------*)

Function lettreident (a : char) :boolean;

var x : byte ;

begin
  x:=ord(a);
  if (((x>64)and(x<91)) or ((x>96)and(x<123)) or (x=95))
     then lettreident:=true
     else lettreident:=false;
end;

(*-------------------------------------------------------------*)

Procedure miseablancarbre (A : arbre);

begin
  fillchar(A^.contenu.op,2,' ');
  fillchar(A^.contenu.typ,3,' ');
  fillchar(A^.contenu.fx,10,' ');
  fillchar(A^.contenu.t1,30,' ');
  fillchar(A^.contenu.t2,30,' ');
  fillchar(A^.contenu.ind,5,' ');
  A^.contenu.op:='';
  A^.contenu.typ:='';
  A^.contenu.fx:='';
  A^.contenu.t1:='';
  A^.contenu.t2:='';
  A^.contenu.ind:='';
end;

(*-------------------------------------------------------------*)

Procedure detruirearbre (A : arbre);

begin
  if A <> nil then begin
                    detruirearbre(A^.sad);
                    detruirearbre(A^.sag);
                   dispose(A);
                  end;
end;
```

```
(*-------------------------------------------------------------*)

function veriffonction(e:tabchar30) : boolean;

begin
  veriffonction:=true;
end;

(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*------------        Analyse de la Rd        ------------*)
(*-------------------------------------------------------------*)

Procedure AnalSynRD (var A : arbre ; var Rd : tabchar240 ;
                     var erreur : integer);

var saut   : SAUTBUF ;

(*-------------------------------------------------------------*)

Procedure AnalLex (deb : byte ; lc : tabchar5 ; var lexeme : tabchar30 ;
                   var fin : byte ; var trouve : boolean);

var j,k : byte ;

begin
  j:=deb;
  k:=1;
  trouve:=false;
  lexeme[0]:=chr(0);fillchar(lexeme,sizeof(lexeme),' ');
  While (j<=length(Rd)) And (erreur=-1) And (trouve=false) Do
  begin
    if Pos(Rd[j],lc) <> 0
       then trouve:=true
       else begin
            if lettreident(Rd[j])
              then lexeme[k]:=Rd[j]
              else erreur:=0
          end;
    inc(j);inc(k);
  end;
  if erreur=-1
     then begin
          if j > length(Rd)
            then lexeme[0]:=chr(k-1)
            else lexeme[0]:=chr(k-2);
          if lexeme[0]=chr(0) then erreur:=2;
        end;
  fin:=j;
end;

(*-------------------------------------------------------------*)
```

```
Procedure AnalDebutRd (var debut : byte ; var lexeme : tabchar30);

var j     : byte ;
   trouve : boolean ;

begin
  AnalLex (1,'(',lexeme,j,trouve);
  if erreur=-1
    then debut:=j-1;
end;

(*-------------------------------------------------------------*)

Procedure AnalCsel(Ar:arbre ; i:byte); forward;

(*-------------------------------------------------------------*)

Procedure Analfin ;

var i,comp : byte ;

begin
  i:=1;
  comp:=0;
  While (i<=length(Rd)) Do
  begin
    if Rd[i]='('
      then inc(comp)
      else if Rd[i]=')'
          then dec(comp);
    inc(i)
  end;
  if comp > 0 then LONGSAUT(saut,48)
            else if comp < 0 then LONGSAUT(saut,49);
end;

(*-------------------------------------------------------------*)

Procedure InitanalCsel ;

var entite : tabchar30;
   deb    : byte;

begin
  AnalDebutRd(deb,entite);
  if erreur=-1
    then begin
          new(A);miseablancarbre(A);
          A^.sad:=nil;
          A^.contenu.typ:='Cas';
          A^.contenu.t2:=entite;
          if length(Rd)=deb
```

```
            then A^.sag:=nil
            else begin
                 erreur:=ENGSAUT(saut);
                 if erreur=NOSAUT
                   then begin
                          new(A^.sag);miseablancarbre(A^.sag);
                          AnalCsel(A^.sag,deb+1);
                         Analfin;
                        end
              end
        end
    else A:=nil
end;


(*-------------------------------------------------------------------------*)
(*----- Procédure de recherche d'opérateur logique ----------*)
(*-------------------------------------------------------------------------*)

Procedure trouveoplog (var A : arbre ; var p : integer ;
                       i : byte);

var j,comp : byte;
   oplog  : string[2];

begin
 j:=i;
 comp:=0;
 p:=0;
 while (p=0) and (j<=length(Rd)) do
 begin
  if Rd[j]='('
    then inc(comp)
    else if Rd[j]=')'
          then dec(comp);
  if comp=0 then p:=j;
  inc(j);
 end;
 if p=0 then erreur:=1
       else begin
            oplog:=Rd[p+1]+Rd[p+2];
            if (oplog <>'ou') and (oplog <>'et')
              then erreur:=2
           end;
 if erreur = -1 then begin
                      A^.contenu.op:=oplog;
                      A^.contenu.typ:='opl'
                     end
             else begin
                      A^.sad:=nil;
                      A^.sag:=nil;
                     end;
end;
```

```
(*-------------------------------------------------------------*)

Procedure AnalCapdebut(A : arbre ;i : byte ; var j : byte);

var trouve  : boolean ;
   lexeme  : tabchar30 ;
   oper    : string[2] ;

begin
  AnalLex(i+1,'<>= ',lexeme,j,trouve);
  case Rd[j-1] of
       '>' : begin
             if (Rd[j]= '=') or (Rd[j]= '<')
                then begin
                     oper:=Rd[j-1]+Rd[j];
                     j:=j+1;
                    end
                else oper:=Rd[j-1];
             end;
       '<' : begin
             if (Rd[j]= '=') or (Rd[j]= '>')
                then begin
                     oper:=Rd[j-1]+Rd[j];
                     j:=j+1;
                    end
                else oper:=Rd[j-1];
             end;
       '=' : begin
             if (Rd[j]= '<') or (Rd[j]= '>') or (Rd[j]='=')
               then begin
                    oper:=Rd[j-1]+Rd[j];
                    j:=j+1;
                   end
               else oper:=Rd[j-1];
             end;
       ' ' : begin
             if copy (Rd,j,3) ='IN '
               then begin
                    oper:='IN';
                    j:=j+3;
                   end
               else if (copy(Rd,j,6)='NOTIN ')
                       or (copy(Rd,j,6)='NOT== ')
                    then begin
                         oper:=Rd[j]+Rd[j+3];
                         j:=j+6;
                        end
                    else erreur:=10;
             end;
      end;
  if erreur=-1
    then if trouve=false
```

```
          then erreur:=1
          else begin
               A^.contenu.op:=oper;
               A^.contenu.t1:=lexeme;
               A^.contenu.typ:='Cap';
             end;
end;


(*-------------------------------------------------------------*)

Procedure AnalCapCsel (var A:arbre ; j : byte);

var k    : byte;
   trouve : boolean;
   lexeme : tabchar30;

begin
  AnalLex (j,'()',lexeme,j,trouve);
  if erreur=-1
    then if trouve=false
          then erreur:=1
          else if Rd[j-1]=')'
               then begin
                     new(A^.sad); miseablancarbre(A^.sad);
                    A^.sad^.sad:=nil;
                    A^.sad^.sag:=nil;
                    A^.sad^.contenu.t2:=lexeme;
                  end
               else begin
                     new(A^.sad); miseablancarbre(A^.sad);
                    A^.sad^.sad:=nil;
                    A^.sad^.contenu.typ:='Cas';
                    A^.sad^.contenu.t2:=lexeme;
                     new(A^.sad^.sag);miseablancarbre(A^.sad^.sag);
                    AnalCsel(A^.sad^.sag,j);
                  end
end;

(*-------------------------------------------------------------*)

procedure AnalCapRd (A : arbre ; var j : byte);

var lexeme : tabchar30 ;
   trouve  : boolean;

begin
  AnalLex(j,'(',lexeme,j,trouve);
  if erreur=-1
    then if trouve=false
          then erreur:=1
          else if Rd[j]=':'
               then begin
```

```
                    A^.contenu.t2:=lexeme;
                    AnalCapCsel(A,j+1)
                  end
               else erreur:=4;
end;

(*-------------------------------------------------------------*)

Procedure AnalCapVal (A : arbre ; j : byte );

var lexeme  : tabchar30;
   trouve   : boolean;

begin
  AnalLex (j+1,"",lexeme,j,trouve);
  if (erreur=-1)
     then if trouve=false
          then erreur:=1
          else if Rd[j]=')'
               then A^.contenu.t2:=""+lexeme+""
               else erreur:=6;
end;

(*-------------------------------------------------------------*)

Procedure AnalCapGrdind (A : arbre ; j : byte);

var trouve   : boolean;
    lexeme  : tabchar30;

begin
  AnalLex(j,')',lexeme,j,trouve);
  if (erreur=-1)
     then if trouve=false
          then erreur:=1
          else A^.contenu.ind:=lexeme
end;

(*-------------------------------------------------------------*)

procedure AnalCapfonction (A : arbre ; var j : byte);

var trouve : boolean;
    lexeme : tabchar30;

begin
  if Rd[j]=""
     then AnalCapVal(A,j)
     else begin
   AnalLex(j,'()',lexeme,j,trouve);
  if (erreur=-1)
     then if trouve=false
          then erreur:=1
```

```
          else begin
                A^.contenu.t2:=lexeme;
                if Rd[j-1]='('
                   then AnalCapGrdind(A,j)
               end
       end
end;

(*-------------------------------------------------------------*)

procedure AnalCapGrd (A : arbre ; var j : byte);

var lexeme  : tabchar30;
   trouve   : boolean;

begin
  AnalLex(j,'()',lexeme,j,trouve);
  if (erreur=-1)
     then if trouve=false
          then erreur:=1
          else if Rd[j-1]=')'
               then A^.contenu.t2:=lexeme
               else if veriffonction(lexeme)=true
                    then begin
                         A^.contenu.fx:=lexeme;
                         AnalCapfonction(A,j);
                        end
                    else begin
                         A^.contenu.t2:=lexeme;
                         AnalCapGrdind(A,j);
                        end;
end;

(*-------------------------------------------------------------*)

Procedure AnalCap (var A : arbre ; i : byte);

var j : byte ;

begin
  AnalCapdebut(A,i,j);
  if erreur = -1
     then if (A^.contenu.op[1]='N') or (A^.contenu.op[1]='I') or
(A^.contenu.op='==')
          then AnalCapRd(A,j)
          else if Rd[j]='"' then AnalCapVal(A,j)
                            else AnalCapGrd(A,j)
     else begin
          A^.sag:=nil;
          A^.sad:=nil;
         end;
end;
```

```
(*-------------------------------------------------------------*)

Procedure AnalCas (var A : arbre ; i : byte ; var p : boolean ;
                   var op : integer);

var trouve : boolean ;
   lexeme : tabchar30 ;
   j     : byte ;

begin
  AnalLex (i,':',lexeme,j,trouve);
  if erreur=-1
    then if trouve=false
           then erreur:=1
           else begin
                  A^.contenu.t1:=lexeme;
                  AnalLex(j,'()',lexeme,j,trouve);
                  if erreur=-1
                    then if trouve=false
                           then erreur:=1
                           else begin
                                  A^.contenu.t2:=lexeme;
                                  p:=(Rd[j-1]='(');
                                  op:=j-1;
                                end
                    else begin
                           A^.sad:=nil;
                           A^.sag:=nil
                         end
                end
     else begin
            A^.sad:=nil;
            A^.sag:=nil;
          end;
end;

(*-------------------------------------------------------------------------*)
(*-----  Procédure récursive d'analyse de la règle ------------*)
(*-------------------------------------------------------------------------*)

Procedure AnalCsel (Ar : arbre ; i : byte);

var p : boolean;
   op : integer;

begin
  case Rd[i] of
       '(' : begin
               trouveoplog(Ar,op,i);
               if erreur <> -1
                 then LONGSAUT(saut,erreur);
               new(Ar^.sag); miseablancarbre(Ar^.sag);Ar^.sad:=nil;
               AnalCsel(Ar^.sag,i+1);
```

```
            new(Ar^.sad); miseablancarbre(Ar^.sad);
            AnalCsel(Ar^.sad,op+4)
          end;
      ':' : begin
            Ar^.sad:=nil;Ar^.sag:=nil;
            AnalCap(Ar,i);
            if erreur <> -1
              then LONGSAUT(saut,erreur);
          end;
      else begin
           if lettreident(Rd[i])
             then begin
                  AnalCas(Ar,i,p,op);
                  if erreur <> -1
                    then LONGSAUT(saut,erreur);
                  if p=true
                    then begin
                         Ar^.contenu.typ:='Cas';
                         Ar^.sad:=nil;
                          new(Ar^.sag); miseablancarbre(Ar^.sag);
                          AnalCsel(Ar^.sag,op+1)
                        end
                    else begin
                         Ar^.sag:=nil;
                         Ar^.sad:=nil;
                          Ar^.contenu.typ:='Cas';
                        end
                end
           end
      end
end;

(*-------------------------------------------------------------*)

begin
  erreur:=-1;
  InitanalCsel;
  if erreur <> -1
    then begin
         writeln('Erreur n° ',erreur);
         detruirearbre(A);
         A:=nil
       end;
end;

(*-------------------------------------------------------------*)
```

III.1.2. Analyse sémantique des règles de domaine : Module VERIFSEMRD

```
(*-------------------------------------------------------------------------*)
(*---------------- ANALYSE SEMANTIQUE ----------------------*)
(*-------------------------------------------------------------------------*)

Procedure AnalSem (Ar : arbre ; var erreur : integer);

var saut : SAUTBUF;

Procedure Semantique (A : arbre ; Prec : arbre) ; forward;

Procedure InitSemantique ;

begin
  if Verifent(Ar^.contenu.t2)=false
    then erreur:=50;
  If A^.sag <> nil
    then begin
          erreur:=ENGSAUT(saut);
          if erreur=NOSAUT
            then Semantique(Ar^.sag,Ar)
        end;
end;

(*-----------------------------------------------------------*)

Procedure SemCapEns (A: arbre ; Prec : arbre );

begin
  writeln('SemCapEns');
  if verifcomptype (A^.contenu.t1,A^.contenu.t2)=false
    then erreur:=63
    else AnalSem (A^.sad,erreur);
end;

(*-----------------------------------------------------------*)

Procedure SemCapVal (A : arbre ; Prec : arbre );

begin
  writeln('SemCapVal');
  if A^.contenu.t2[1]=""
    then begin
          if veriftypeval (A^.contenu.t1,A^.contenu.t2)=false
            then erreur:=60;
          if A^.contenu.fx <> ''
            then if verifcompfx (A^.contenu.t2,A^.contenu.fx)=false
                  then erreur:=64
        end
    else if A^.contenu.ind ='
```

```
          then begin
               if verifgrdsimple (A^.contenu.t2)=false
                 then erreur:=52;
               if verifcomptype (A^.contenu.t1,A^.contenu.t2)=false
                 then erreur:=62;
               if A^.contenu.fx <> ''
                 then if verifcompfx (A^.contenu.t2,A^.contenu.fx)=false
                      then erreur:=63
             end
          else begin
               if verifgrdind(A^.contenu.t2,A^.contenu.ind)=false
                 then erreur:=54;
               if verifcomptype (A^.contenu.t1,A^.contenu.t2)=false
                 then erreur:=62;
               if A^.contenu.fx <> ''
                 then if verifcompfx (A^.contenu.t2,A^.contenu.fx)=false
                      then erreur:=63
             end;
end;

(*-------------------------------------------------------------*)

Procedure SemCap (A : arbre ; Prec : arbre);

var t : tabchar5 ;

begin
  writeln('SEMCAP  A:', A^.contenu.t2,'/ Prec : ',Prec^.contenu.t2);
  if verifattrreel (A^.contenu.t1,Prec^.contenu.t2)=false
    then if verifattrvir (A^.contenu.t1,Prec^.contenu.t2)=false
          then erreur:=51
          else if (A^.contenu.op[1]='N') or (A^.contenu.op[1]='I')
               or (A^.contenu.op ='==')
               then SemCapEns(A,Prec)
               else SemCapVal(A,Prec)
    else if (A^.contenu.op[1]='N') or (A^.contenu.op[1]='I')
          or (A^.contenu.op ='==')
          then SemCapEns(A,Prec)
          else SemCapVal(A,Prec);

end;

(*-------------------------------------------------------------*)

Procedure SemCas (A : arbre ; Prec : arbre);

var entite : tabchar30;

begin
  writeln('SEMCAS  A:', A^.contenu.t2,'/ Prec : ',Prec^.contenu.t2);
  if verifent(A^.contenu.t2) = false
    then if verifvarent (A^.contenu.t2) = false
          then erreur:=7
```

```
          else begin
               Trouveent (A^.contenu.t2,entite);
               if verifas (A^.contenu.t1,entite,Prec^.contenu.t2) = false
                 then erreur:=6;
             end
    else if verifas (A^.contenu.t1,A^.contenu.t2,Prec^.contenu.t2)=false
          then erreur:=6;
end;

(*-------------------------------------------------------------*)

Procedure AvancerPrec (Adep : arbre ; A : arbre ; var Prec : arbre) ;

begin
 if Adep <> nil
 then begin
       if Adep^.sag = A
         then Prec:=Adep
         else if Adep^.sad = A
               then Prec:=Adep
               else begin
                     AvancerPrec (Adep^.sag,A,Prec);
                     AvancerPrec (Adep^.sad,A,Prec);
                   end;
     end;
end;

(*-------------------------------------------------------------*)

Procedure Semantique (A : arbre ; Prec : arbre) ;

begin
 if A <> nil
   then begin
         case A^.contenu.typ[3] of
          'l' : begin
                 Semantique (A^.sag,Prec);
                 Semantique (A^.sad,Prec);
               end;
          's' : begin
                 SemCas (A,Prec);
                 if erreur <> -1
                   then LONGSAUT (saut,erreur);
                 If A^.sag <> nil
                   then begin
                         AvancerPrec (Ar,A^.sag,Prec);
                         Semantique (A^.sag,Prec);
                        end;
               end;
          'p' : begin
                 SemCap (A,Prec);
                 if erreur <> -1
                   then LONGSAUT (saut,erreur);
```

```
            end;
          end;
        end;
end;

(*-------------------------------------------------------------*)

begin
 erreur:=-1;
 Initsemantique;
end;

(*-------------------------------------------------------------*)
```

III.1.3. Création GDV : Module GDV

```
(*-------------------------------------------------------------------------------------*)
(*-------DECOMPOSITION DE LA CONDITION DE SELECTION ----------*)
(*-------------------------------------------------------------------------------------*)

Procedure copielien (L1,L2,P : arbre ; var prec : arbre);

begin
 L2^:=L1^;
 prec:=L2;
 if (L1^.sag=nil) or (L1^.sag=P)
   then L2^.sag:=nil
   else begin
          new(L2^.sag);
          copielien(L1^.sag,L2^.sag,P,prec)
        end;
 if (L1^.sad=nil) or (L1^.sad=P)
   then L2^.sad:=nil
   else begin
          new(L2^.sad);
          copielien(L1^.sad,L2^.sad,P,prec)
        end
```

```
end;

(*-------------------------------------------------------------*)

Procedure Detruirelien (L1,P : arbre);

begin
 if (L1 <> nil) and (L1 <> P)
   then begin
        detruirelien (L1^.sag,P);
        dispose(L1);
       end;
end;

(*-------------------------------------------------------------*)

Procedure Initlien (A : arbre ; P  : arbre ; gauche : boolean);

var L1,L2      : arbre ;
   Prec1,Prec2 : arbre ;
   Temp        : arbre ;

begin
 if gauche=true
   then begin
        if A^.sag <> P
          then begin
               new(L1);new(L2);
               Prec1:=nil;Prec2:=nil;
               copielien (A^.sag,L1,P,Prec1);
               copielien (A^.sag,L2,P,Prec2);
               (*detruirelien(A^.sag,P);*)
              A^.sag:=P;
              Temp:=P^.sag;
               P^.sag:=L1;Prec1^.sag:=Temp;
              Temp:=P^.sad;
               P^.sad:=L2; Prec2^.sag:=Temp;
             end;
      end
   else begin
        if A^.sad <> P
          then begin
               new(L1);new(L2);
               Prec1:=nil;Prec2:=nil;
               copielien (A^.sad,L1,P,Prec1);
               copielien (A^.sad,L2,P,Prec2);
               (*detruirelien(A^.sad,P);*)
              A^.sad:=P;
              Temp:=P^.sag;
               P^.sag:=L1;Prec1^.sag:=Temp;
              Temp:=P^.sad;
               P^.sad:=L2; Prec2^.sag:=Temp;
             end;
```

```
        end;
end;

(*-------------------------------------------------------------*)

Procedure decomposer (A : arbre) ;

var P         : arbre ;
   saut       : SAUTBUF ;
   coderetour : integer ;
   decomp     : byte   ;

(*-------------------------------------------------------------*)

Procedure repereroplog (A : arbre) ;

begin
 if A=nil then P:=nil
        else if A^.contenu.typ='opl'
             then begin
                  P:=A;
                  coderetour:=2;
                  LONGSAUT(saut,coderetour);
                end
             else begin
                  repereroplog(A^.sag);
                  repereroplog(A^.sad)
                end
end;

(*-------------------------------------------------------------*)

begin
 if A <> nil
   then begin
        decomp:=0;
        coderetour:=ENGSAUT(saut);
        if coderetour = NOSAUT
          then repereroplog (A^.sag)
          else if P <> nil
               then begin
                    Initlien(A,P,true);
                    decomp:=1;
                  end;
        coderetour:=ENGSAUT(saut);
        if coderetour = NOSAUT
          then repereroplog (A^.sad)
          else if P <> nil
               then begin
                    Initlien (A,P,false);
                    decomp:=decomp+2;
                  end;
        if decomp=1
```

```
            then Decomposer (A^.sag)
            else if decomp=2 then Decomposer (A^.sad)
                        else if decomp=3
                             then begin
                                    Decomposer(A^.sag);
                                    Decomposer(A^.sad);
                                  end;
    end;
end;

(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*---------------- Copie arbre --------------------------------*)
(*-------------------------------------------------------------*)

Procedure copiearbre (A1,A2 : arbre);

begin
  A2^:=A1^;
  if A1^.sag=nil then A2^.sag:=nil
              else begin
                     new(A2^.sag);
                     copiearbre(A1^.sag,A2^.sag)
                   end;
  if A1^.sad=nil then A2^.sad:=nil
              else begin
                     new(A2^.sad);
                     copiearbre(A1^.sad,A2^.sad)
                   end
end;

(*-------------------------------------------------------------*)

Procedure copie (A : arbre ; gauche : boolean);

var L1,L2       : arbre;
    Prec1,Prec2 : arbre;
    SA,SABIS    : arbre;
    PSAG,PSAD   : arbre;
    I : char;

begin
  if gauche=true
     then begin
          SA:=A^.sad;
          new(SAbis);
          copiearbre(SA,SABIS);
          PSAD:=A^.sag^.sad;
          PSAG:=A^.sag^.sag;
          A^.contenu.op:='ou'; A^.sag^.contenu.op:='et';
          new(A^.sad);miseablancarbre(A^.sad);
          A^.sad^.contenu.op:='et'; A^.sad^.contenu.typ:='opl';
```

```
          A^.sag^.sag:=PSAG;
          A^.sad^.sag:=PSAD;
          A^.sag^.sad:=SA;
          A^.sad^.sad:=SAbis;
        end
    else begin
          SA:=A^.sag;
          new(SAbis);
          copiearbre(SA,SABIS);
          PSAD:=A^.sad^.sad;
          PSAG:=A^.sad^.sag;
          A^.contenu.op:='ou'; A^.sad^.contenu.op:='et';
          new(A^.sag); miseablancarbre(A^.sag);
          A^.sag^.contenu.op:='et'; A^.sag^.contenu.typ:='opl';
          A^.sag^.sad:=PSAG;
          A^.sad^.sad:=PSAD;
          A^.sag^.sag:=SA;
          A^.sad^.sag:=SAbis;
        end
end;

(*-------------------------------------------------------------------*)
(*------------ Distribution des opérateurs ------------------*)
(*-------------------------------------------------------------------*)

procedure Distrib (A : arbre );

var distribution : byte ;

begin
  if A <> nil
    then if A^.contenu.op <> 'et'
          then begin
               Distrib(A^.sag);
               Distrib(A^.sad);
             end
          else begin
               distribution:=0;
               if A^.sag^.contenu.op='ou'
                 then begin
                      copie(A,true);
                      Distribution:=1;
                    end
                 else distribution:=2;
               if A^.sad^.contenu.op='ou'
                 then begin
                      copie(A,false);
                      Distribution:=1;
                    end
                 else distribution:=distribution+3;

               if (distribution=1)  or (distribution=4)
                 then distrib(A)
```

```
                else begin
                     distrib(A^.sag);
                     distrib(A^.sad);
                   end;
            end;
end;

(*-------------------------------------------------------------*)

Function Count (A:arbre):byte;

begin
  if A=nil then count:=0
        else count:=1+count(A^.sag)+count(A^.sad);
end;

(*-------------------------------------------------------------*)

procedure distributiontotale (A : arbre);

var n1,n2 : byte;

begin
  n1:=count(A);
  distrib(A);
  n2:=count(A);
  while (n2 <> n1) do
  begin
    n1:=n2;
    distrib(A);
    n2:=count(A);
  end;
end;

(*-------------------------------------------------------------*)
(*-----------Anal arbre ET-----------------------------------*)
(*-------------------------------------------------------------*)

Function identifiant (Ent,attr : tabchar30) : byte ;

begin
  identifiant:=2;
end;

(*-------------------------------------------------------------*)

Procedure Placerfinliste (nom1,nom2 : tabchar30 ; i,j : byte);

var l,p : ptliste ;

begin
  if tab[i,j]=nil
    then begin
```

```
          new(tab[i,j]);
          tab[i,j]^.suiv:=nil;
          tab[i,j]^.attr:=nom1;
          tab[i,j]^.ent:=nom2;
        end
    else begin
          l:=tab[i,j]^.suiv;
          p:=tab[i,j];
          while (l <> nil) do
          begin
           p:=l;
           l:=l^.suiv;
          end;
          new(p^.suiv);
          p^.suiv^.suiv:=nil;
          p^.suiv^.attr:=nom1;
          p^.suiv^.ent:=nom2;
        end;
end;

(*-------------------------------------------------------------*)

Procedure Regle1 (A : arbre ; i : byte);

begin
 if verifvarent (A^.contenu.t2)
    then placerfinliste(",A^.contenu.t2,i,1)
end;

(*-------------------------------------------------------------*)

Procedure Regle2 (A,Prec,Init : arbre ; i : byte);

var x : byte ;

begin
 if A^.contenu.ind <> "
    then placerfinliste(",A^.contenu.ind,i,1);

 if Prec=Init
    then begin
          if identifiant (Prec^.contenu.t2,A^.contenu.t1) <> 0
            then placerfinliste (A^.contenu.t1,",i,2);
        end
    else begin
          x:=identifiant (Prec^.contenu.t2,A^.contenu.t1);
          if x=2
            then Placerfinliste (A^.contenu.t1,Prec^.contenu.t2,
                                 i,3)
            else if x=1
                 then Placerfinliste (Prec^.contenu.t2,",i,3);
        end;
end;
```

```
(*-------------------------------------------------------------*)

Procedure Cassimple (A,Prec,Init : arbre ; i : byte);

begin
 if A^.sag <> nil
    then  Cassimple (A^.sag,A,Init,i)
    else if A^.contenu.typ='Cas'
         then Regle1 (A,i)
         else Regle2 (A,Prec,Init,i);
end;

(*-------------------------------------------------------------*)

Procedure Creeret (A,Prec,Init : arbre ; i : byte);

begin
 if A <> nil
    then begin
         if A^.contenu.op='et'
           then begin
                creeret(A^.sag,Prec,Init,i);
                creeret(A^.sad,Prec,Init,i);
              end
           else Cassimple(A,Prec,Init,i);
       end;
end;

(*-------------------------------------------------------------*)
Procedure AnalarbreET (A,Prec,Init : arbre ; var i : byte);

var x : char ;

begin
 i:=i+1;
 tab[i,1]:=nil;
 tab[i,2]:=nil;
 tab[i,3]:=nil;
 Creeret(A,Prec,Init,i);
end;

(*-------------------------------------------------------------*)

Procedure ArbreET (A,Prec,Init : arbre ; var i : byte) ;

begin
 if A <> nil
    then if A^.contenu.op = 'ou'
         then begin
              if A^.sag^.contenu.op='ou'
                then ArbreET (A^.sag,Prec,Init,i)
                else AnalarbreET (A^.sag,Prec,Init,i);
```

```
                if A^.sad^.contenu.op='ou'
                  then ArbreET (A^.sad,Prec,Init,i)
                  else AnalarbreET (A^.sad,Prec,Init,i);
              end
            else AnalarbreET (A,Prec,Init,i);
end;

(*-------------------------------------------------------------*)

Procedure liretab (t : tabet ; lg : byte);

var i,j : byte ;
   l : ptliste;

begin
 i:=1;
 while (i<=lg) do
 begin
   writeln('arbre numéro : ',i);
   j:=1;
   while (j<=3) do
   begin
     l:=tab[i,j];
     write('liste : ',j);
     while (l<>nil) do
     begin
       write('|',l^.attr, '/',l^.ent,'  ');
       l:=l^.suiv;
     end;
     inc(j);
     writeln;
   end;
   inc(i);
   writeln;
 end;
end;

(*-------------------------------------------------------------*)
```

III.2. Analyse des règles du problème

III.2.1. Création de la table des symboles : Module CREERTABLE

```
{-------------------------------------------------}
{--- Création de la table des symboles ---}
{-------------------------------------------------}

{$I floyd.pas}

Procedure CREERTABLE;

var i,j,k : integer;
   model : TMODELE;
   gra   : TGRANDEUR;
   gra1  : TGRANDEUR;
   gra2  : TGRANDEUR;
   grad  : TGD;
   grand : TGND;
   dim1  : TDIMENSION;
   Ve    : TVARENT;
   Vl    : TVARLIB;
   ind   : TINDICAGE;
   dep   : TDEPENDANCE;
   db    : DDB;

{-----------------------------------------------------------------}

Procedure Initialisation;

begin
 i:=1;
 while i<=20 do
 begin
  j:=1;
  while j<=20 do
  begin
   if i=j then m[i,j]:=1
         else m[i,j]:=1024;
   j:=j+1;
  end;
  i:=i+1;
 end;
 i:=1;
 while i<=20 do
 begin
  fillchar(gdv[i],15,' ');gdv[i]:='';
  fillchar(dim[i],15,' ');dim[i]:='';
  i:=i+1;
 end;
 i:=1;
 while i<=100 do
```

```
  begin
    fillchar(grdi[i].libelle,15,' ');grdi[i].libelle:='';
   j:=1;
   while j<=5 do
   begin
    grdi[i].dimension[j]:=0;
    j:=j+1;
   end;
   i:=i+1;
  end;
end;

{-----------------------------------------------------------------}

Procedure Creertablegrandeur;

begin
 i:=1;j:=1;k:=1;
 dbfpath(gra,model,APPARTENANCE);
 while dbfound do
 begin
  dbfpath(grand,gra,ISGND);
  if dbfound
    then begin
         grdi[i].libelle:=gra.NOM_G;
        i:=i+1;
       end
    else begin
         dbfpath(grad,gra,ISGD);
         if dbfound
           then begin
                grdi[i].libelle:=gra.NOM_G;
               i:=i+1;
              end
           else begin
                dbfpath(dim1,gra,ISD);
                dbfpath(Ve,dim1,ISVE);
                if dbfound
                  then begin
                       gdv[j]:=gra.NOM_G;
                       j:=j+1;
                      end
                  else begin
                       dim[k]:=gra.NOM_G;
                       k:=k+1;
                      end;
             end;
        end;
   dbnpath(gra,model,APPARTENANCE);
 end;
 lgvl:=k-1;lgve:=j-1;lggr:=i-1;
end;
```

```
{-------------------------------------------------------------------}

Procedure Creerdimgrandeur;

var trouvergr,trouverve,trouvervl : boolean;

begin
  dbfpath(gra,model,APPARTENANCE);
  while dbfound do
  begin
    dbfpath(grad,gra,ISGD);
    if dbfound
      then begin
           i:=1;
           trouvergr:=false;
            while (i<=lggr) and (trouvergr=false) do
           begin
             if grdi[i].libelle=gra.NOM_G
               then begin
                    trouvergr:=true;
                    dbfpath(ind,grad,GDI);
                   k:=1;
                    while dbfound do
                   begin
                     dbfpath(dim1,ind,-DI);
                     dbfpath(gra2,dim1,-ISD);
                     dbfpath(ve,dim1,ISVE);
                     if dbfound
                       then begin
                            trouverve:=false;
                           j:=1;
                            while (j<=lgve) and
                                 (trouverve=false) do
                           begin
                             if gdv[j]=gra2.NOM_G
                               then begin
                                 grdi[i].dimension[k]:=j;
                                   k:=k+1;
                                    trouverve:=true;
                                 end;
                             j:=j+1;
                           end;
                         end;
                    dbfpath(vl,dim1,ISVL);
                    if dbfound
                      then begin
                           j:=1;trouvervl:=false;
                           while (j<=lgvl) and
                                (trouvervl=false) do
                           begin
                             if dim[j]=gra2.NOM_G
                               then begin
                                 grdi[i].dimension[k]:=100+j;
```

```
                        k:=k+1; trouvervl:=true;
                          end;
                       j:=j+1;
                     end;
                   end;
                dbnpath(ind,grad,GDI);
              end;
            end
          else i:=i+1;
        end;
      end;
  dbnpath(gra,model,APPARTENANCE);
 end;
end;

{-----------------------------------------------------------------}

Procedure Creermatricegdv;

var trouvergra1,trouvergra2 : boolean;
  k,i,j : integer;

begin
 dbfpath(gra1,model,APPARTENANCE);
 while dbfound do
 begin
  dbfpath(dim1,gra1,ISD);
  if dbfound
    then begin
         dbfpath(dep,gra1,grdep);
         while dbfound do
         begin
           dbfpath(gra2,dep,-depgr);
           dbfpath(dim1,gra2,ISD);
           if dbfound
             then begin
                  k:=1;trouvergra1:=false;trouvergra2:=false;
                  while (k<=lgve) and ((trouvergra1=false)or
                      (trouvergra2=false)) do
                 begin
                   if ((gdv[k]=gra1.NOM_G) and
                     (trouvergra1=false))
                     then begin
                          i:=k;trouvergra1:=true;
                        end;
                   if ((gdv[k]=gra2.NOM_G) and
                     (trouvergra2=false))
                     then begin
                          j:=k;trouvergra2:=true;
                        end;
                  inc(k);
                 end;
                 m[i,j]:=dep.RELATION;
```

```
                end;
              dbnpath(dep,gra1,GRDEP);
            end;
          end;
    dbnpath(gra1,model,APPARTENANCE);
  end;
end;

{-----------------------------------------------------------------}

begin
  dbopen('MODSPECI',db);
  model.NOM_M:=nommodele;
  dbid(MODELE,model);
  initialisation;
  creertablegrandeur;
  creerdimgrandeur;
  creermatricegdv;
  dbclose(db);
  floyd(m,lgve,resul);
end;

{-----------------------------------------------------------------}
{---------- Recherche du chemin le plus court - --------}
{-----------------------------------------------------------------}

Procedure FLOYD (var m : matint20;lg : integer;var res : matint22);

var a,p : matint20;
    z,i,j,k : integer;

{-----------------------------------------------------------------}

Procedure initialisation;

begin
  i:=1;
  while i<=lg do
  begin
    j:=1;
    while j<=lg do
    begin
      a[i,j]:=m[i,j];
      if i=j then p[i,j]:=1
            else if m[i,j]=1024 then p[i,j]:=0
                                else p[i,j]:=m[i,j];
      inc(j);
    end;
    inc(i);
  end;
end;

{-----------------------------------------------------------------}
```

```
Procedure calculchemin1;

begin
 k:=1;
 while k<=lg do
 begin
  i:=1;
  while i<=lg do
  begin
   if a[i,k]<>1024
     then begin
          j:=1;
          while j<=lg do
          begin
           z:=a[i,k]+a[k,j];
           if z<a[i,j]
              then begin
                   if z<>0
                      then begin
                           a[i,j]:=z;
                           p[i,j]:=p[i,k]*p[k,j];
                      end;
                end;
           inc(j);
          end;
         end;
   inc(i);
  end;
  inc(k);
 end;
end;

{-----------------------------------------------------------------}

Procedure calculchemin2;

begin
 k:=1;
 while k<=lg do
 begin
  i:=1;
  while i<=lg do
  begin
   if a[i,k]<>1024
     then begin
          j:=1;
          while j<=lg do
          begin
           z:=a[i,k]+a[k,j];
           if z<a[i,j]
              then begin
                   if z<>0
```

```
                    then begin
                        if p[i,k]*p[k,j]<>1
                          then begin
                              a[i,j]:=z;
                               p[i,j]:=p[i,k]*p[k,j];
                             end;                                    end;
                end;
            inc(j);
           end;
         end;
    inc(i);
   end;
   inc(k);
  end;
end;

{-----------------------------------------------------------------}

begin
  initialisation;
  calculchemin1;
  for i:=1 to lg do
  begin
    for j:=1 to lg do
    begin
      res[i,j,1]:=p[i,j];
    end;
  end;
  initialisation;
  calculchemin2;
  for i:=1 to lg do
  begin
    for j:=1 to lg do
    begin
      res[i,j,2]:=p[i,j];
    end;
  end;
end;

{-----------------------------------------------------------------}
```

III.2.2. Analyse lexicale des règles du problème : Module TRANSFORMEREGLE

```
{-------------------------------------------------------------------}
{------------ Transformation d'une règle ----------------}
{-------------------------------------------------------------------}

Procedure TRANSFORMEREGLE (var entree : tabchar240);

var i,k : integer;
    varent,parentouv,varlib,grandeur : boolean;
    numind,compteur,autre : integer ;

{-------------------------------------------------------------------}

Procedure Trouversidimension ;

var s : tabchar15;
    p,debut,fin,place : integer;
    att : TATTRIBUTE;
    db  : DDB;
    t   : tabchar32;
    ok  : boolean;

begin
  fillchar(s,15,' ');s:='';
  debut:=i;
  autre:=0;
  place:=1;
  ok:=false;
  while ((entree[i]>=#65) and (entree[i]<=#90))or(entree[i]=#95)do
  begin
    autre:=autre+(ord(entree[i])*place);
    inc(place);
    inc(i);
  end;
  fin:=i-1;
  autre:=autre+1000;
  s:=copy(entree,debut,fin-debut+1);
  p:=1;
  while (p<=lggr)and(grandeur=false) do
  begin
    if grdi[p].libelle=s then begin
                        grandeur:=true;
                        numind:=p;
                      end
                    else grandeur:=false;
    inc(p);
  end;
  if grandeur=false
     then begin
          p:=1;
          while (p<=lgve)and(varent=false) do
```

```
        begin
          if gdv[p]=s then begin
                          varent:=true;
                          numind:=p;
                        end
                     else varent:=false;
          inc(p);
        end;
      end;
if (varent=false)and(grandeur=false)
  then begin
        p:=1;
         while (p<=lgvl) and (varlib=false) do
        begin
          if dim[p]=s then begin
                          varlib:=true;
                          numind:=p;
                        end
                     else varlib:=false;
          inc(p);
        end;
      end;
  if (varent=false)and(grandeur=false)and(varlib=false)
  then begin
        fillchar(t,32,' ');t:='';
       t:=s;
        dbopen('BIBLIO',db);
        dbfirst(ATTRIBUTE,att);
         while (ok=false)and(dbfound) do
        begin
          if t=att.NAME
            then ok:=true;
          dbnext(ATTRIBUTE,att);
        end;
        dbclose(db);
        if ok=false
          then begin
               if s='ET'
                 then begin
                      ok:=true;
                      autre:=1;
                     end;
               if s='OU'
                 then begin
                      ok:=true;
                      autre:=2;
                     end;
               if s='VRAI'
                 then begin
                      ok:=true;
                      autre:=3;
                     end;
               if s='FAUX'
```

```
                then begin
                     ok:=true;
                     autre:=4;
                    end;
              end;
          if ok=false
            then begin
                 err:=(t)+(' n"existe pas');
                 messageerreur(err);
              end;
        end;

end;

{-----------------------------------------------------------------}

begin
  i:=1;compteur:=0;
  while i<=length(entree) do
  begin
    entree[i]:=upcase(entree[i]);inc(i)
  end;
  i:=1;k:=1;
  varent:=false;varlib:=false;parentouv:=false;grandeur:=false;
  while i<=length(entree) do
  begin
    case entree[i] of
      'A'..'Z' : begin
                 trouversidimension;
                 if (varent=true)
                   then sortie[k]:=numind+300
                   else if (varlib=true)
                        then sortie[k]:=numind+400
                        else begin
                             if (parentouv=true)
                               then begin
                                    sortie[k]:=40;inc(k);
                                   end;
                             if grandeur=true
                               then begin
                                    sortie[k]:=numind+500;
                                    grandeur:=false;
                                   end
                               else sortie[k]:=autre
                            end;
                inc(k);
                parentouv:=false;
               end;

      '(': begin
           compteur:=compteur+1;
           if (varent=true) or (varlib=true)
             then i:=i+3
```

```
            else begin
                  if parentouv=true
                    then begin
                          sortie[k]:=40;inc(k);
                         end;
                  parentouv:=true;
                  inc(i);
                end;
          end;

      ')': begin
           dec(compteur);
           if (varent=false) and (varlib=false)
             then begin
                  sortie[k]:=41;
                  inc(k);
                end;
           varent:=false;varlib:=false;
           inc(i);
          end;

      ',' : begin
            inc(i);
            varent:=false;varlib:=false;
           end;

      '<','>' : begin
                if entree[i+1]='='
                  then begin
                       if (entree[i]='<')
                         then sortie[k]:=243
                         else sortie[k]:=242;
                       inc(i);
                      end
                  else sortie[k]:=ord(entree[i]);
                inc(i);inc(k);
               end;

      ' ' : inc(i);

      else begin
           if parentouv=true
             then begin
                  sortie[k]:=40;inc(k);
                end;
           sortie[k]:=ord(entree[i]);
           inc(k);inc(i);
           parentouv:=false;
          end;
    end;
  end;
  sortie[k]:=1000;
end;
```

III.2.3. Analyse syntaxique des règles du problème : Module VERIFICATIONSYNTAXE

```
{-------------------------------------------------------------------}
{------- Vérification de la syntaxe d'une règle ---------}
{-------------------------------------------------------------------}

Procedure VERIFICATIONSYNTAXE;

var b,pa : tabint100;
    c : matint10;
    i,j,lg,p,v,w,compteur,val : integer;
    reviser,erreur,valeur : boolean;

{-------------------------------------------------------------------}

procedure Inittablepriorites;

begin
  (*-------------------- + ------------------------------*)
  c[1,1]:=0;c[1,2]:=1;c[1,3]:=1;c[1,4]:=1;c[1,5]:=1;c[1,6]:=1;
  c[1,7]:=2;c[1,8]:=1;c[1,9]:=1;c[1,10]:=1;
  (*-------------------- - ------------------------------*)
  c[2,1]:=2;c[2,2]:=2;c[2,3]:=1;c[2,4]:=1;c[2,5]:=1;c[2,6]:=1;
  c[2,7]:=2;c[2,8]:=1;c[2,9]:=1;c[2,10]:=1;
  (*-------------------- * ------------------------------*)
  c[3,1]:=2;c[3,2]:=2;c[3,3]:=0;c[3,4]:=1;c[3,5]:=1;c[3,6]:=1;
  c[3,7]:=2;c[3,8]:=2;c[3,9]:=2;c[3,10]:=2;
  (*-------------------- / ------------------------------*)
  c[4,1]:=2;c[4,2]:=2;c[4,3]:=2;c[4,4]:=1;c[4,5]:=1;c[4,6]:=1;
  c[4,7]:=2;c[4,8]:=2;c[4,9]:=2;c[4,10]:=2;
  (*-------------------- ^ ------------------------------*)
  c[5,1]:=2;c[5,2]:=2;c[5,3]:=2;c[5,4]:=2;c[5,5]:=2;c[5,6]:=1;
  c[5,7]:=2;c[5,8]:=2;c[5,9]:=2;c[5,10]:=2;
  (*-------------------- ( ------------------------------*)
  c[6,1]:=1;c[6,2]:=1;c[6,3]:=1;c[6,4]:=1;c[6,5]:=1;c[6,6]:=1;
  c[6,7]:=3;c[6,8]:=2;c[6,9]:=2;c[6,10]:=2;
  (*-------------------- ) ------------------------------*)
  c[7,1]:=4;c[7,2]:=4;c[7,3]:=4;c[7,4]:=4;c[7,5]:=4;c[7,6]:=4;
  c[7,7]:=4;c[7,8]:=4;c[7,9]:=4;c[7,10]:=4;
  (*-------------------- Σ ------------------------------*)
  c[8,1]:=2;c[8,2]:=2;c[8,3]:=1;c[8,4]:=1;c[8,5]:=1;c[8,6]:=1;
  c[8,7]:=1;c[8,8]:=4;c[8,9]:=4;c[8,10]:=4;
  (*-------------------- π ------------------------------*)
  c[9,1]:=2;c[9,2]:=2;c[9,3]:=1;c[9,4]:=1;c[9,5]:=1;c[9,6]:=1;
  c[9,7]:=1;c[9,8]:=4;c[9,9]:=4;c[9,10]:=4;
  (*-------------------- √ ------------------------------*)
  c[10,1]:=2;c[10,2]:=2;c[10,3]:=1;c[10,4]:=1;c[10,5]:=1;
  c[10,6]:=1;c[10,7]:=1;c[10,8]:=4;c[10,9]:=4;c[10,10]:=4
end;

{-------------------------------------------------------------------}
```

```
procedure Conversion (var v,w : integer);

begin
  if pa[i-1]=43 then v:=1 ;if pa[i-1]=45 then v:=2;
  if pa[i-1]=42 then v:=3 ;if pa[i-1]=47 then v:=4;
  if pa[i-1]=94 then v:=5 ;if pa[i-1]=40 then v:=6;
  if pa[i-1]=41 then v:=7 ;if pa[i-1]=228 then v:=8;
  if pa[i-1]=227 then v:=9;if pa[i-1]=251 then v:=10;
  if sortie[p]=43 then w:=1 ;if sortie[p]=45 then w:=2;
  if sortie[p]=42 then w:=3 ;if sortie[p]=47 then w:=4;
  if sortie[p]=94 then w:=5 ;if sortie[p]=40 then w:=6;
  if sortie[p]=41 then w:=7 ;if sortie[p]=228 then w:=8;
  if sortie[p]=227 then w:=9;if sortie[p]=251 then w:=10;
end;

{-----------------------------------------------------------------}

begin
  inittablepriorites;
  for i:=1 to 100 do begin
                       b[i]:=1000;
                     end;
  p:=1;erreur:=false;
  while sortie[p]<>61 do inc(p);
  inc(p);
  j:=1;i:=1;
  while (sortie[p]<>1000) and (erreur=false) do
  begin
    val:=0;valeur:=false;
    while (sortie[p]>=48) and (sortie[p]<=57) do
    begin
      val:=(val*10)+(sortie[p]-48);
      inc(p);valeur:=true;
    end;
    if valeur=true then begin
                          b[j]:=val+1001;
                          inc(j);
                        end;
    if ((sortie[p]>=300) and (sortie[p]<>1000))
      then begin
             b[j]:=sortie[p];
             inc(j);inc(p);
           end;
    if (sortie[p]<300)
      then begin
             reviser:=true;
             while reviser=true do
             if i=1
               then begin
                      pa[i]:=sortie[p];
                      inc(i);
                      if (sortie[p]=228)or(sortie[p]=227)
                        then p:=p+2
```

```
                    else inc(p);
                  reviser:=false;
                end
              else begin
                   conversion(v,w);
                   if (c[v,w]=1)or(c[v,w]=0)
                     then begin
                          pa[i]:=sortie[p];
                          inc(i);
                           if (sortie[p]=228)or(sortie[p]=227)
                            then p:=p+2
                            else inc(p);
                          reviser:=false;
                        end;
                   if c[v,w]=3
                     then begin
                          inc(p);i:=i-1;
                          reviser:=false;
                        end;
                   if c[v,w]=2
                     then begin
                          b[j]:=pa[i-1];
                          i:=i-1;inc(j);
                          reviser:=true;
                        end;
                   if c[v,w]=4
                     then begin
                          erreur:=true;
                          reviser:=false;
                        end;
                end;
        end;
end;
while i>1 do
begin
 b[j]:=pa[i-1];
 i:=i-1;
 inc(j);
end;
i:=1;compteur:=0;
while (b[i]<>1000)and(erreur=false) do
begin
 if (b[i]>=1)and(b[i]<=299)
   then dec(compteur)
   else if ((b[i]>=500)and(b[i]<>1000)) then inc(compteur);
 if compteur<0 then erreur:=true;
 inc(i);
end;
i:=1;
while b[i]<>1000 do
begin
 write(b[i]);write('|');
 inc(i);
```

```
  end;
  fillchar(err,70,' ');err:='';
  err:='Erreur de syntaxe dans la règle.';
  if erreur=true then messageerreur(err);
end;

{-----------------------------------------------------------------}
```

III.2.4. Analyse sémantique des règles du problème : Module VERIFICATIONREGLE

```
{-----------------------------------------------------------------}
{--------- Vérification au niveau des dimensions ------}
{-----------------------------------------------------------------}

Procedure VERIFICATIONREGLE;

var fa,present,trouverve,trouvervs,indnivsup : boolean;
    compteur,i : integer;
    indice,Ve : integer;
    cont : char;
    err : tabchar70;

{$I transfre.pas}
{$I creerpil.pas}

{-----------------------------------------------------------------}

Procedure Comparer (var a : tabint5 ; num : integer);

var p,som1,som2 : integer;

begin
  p:=1;som1:=0;som2:=0;
  while p<=5 do
  begin
    if a[p]=0 then som1:=som1+1
           else som1:=som1+(a[p]*10);
    if grdi[num].dimension[p]=0
      then som2:=som2+1
      else som2:=som2+(grdi[num].dimension[p]*10);
    inc(p);
  end;
  if som1<>som2
     then begin
          err:=('Erreur dans les dimensions de ')
              +copy(grdi[num].libelle,1,length(grdi[num].libelle));
          messageerreur(err);
        end;
end;
```

```
{-------------------------------------------------------------------}

procedure VerificationAppBD;

var i,j,num : integer;
   a : tabint5;

begin
 i:=1;
 while sortie[i]<1000 do
 begin
   if (sortie[i]>=501)and(sortie[i]<=600)
     then begin
          num:=sortie[i]-500;
          j:=1;
          a[1]:=0;a[2]:=0;a[3]:=0;a[4]:=0;a[5]:=0;
          inc(i);
          while (sortie[i]>=301)and(sortie[i]<=500) do
          begin
            if (sortie[i]>=301)and(sortie[i]<=400)
              then a[j]:=sortie[i]-300
              else a[j]:=sortie[i]-300;
            inc(j);inc(i);
          end;
          comparer(a,num);
        end
     else inc(i);
 end;
end;

{-------------------------------------------------------------------}

Procedure TrouverVeEntete;

var p : integer;

begin
 p:=1;trouverve:=false;
 while (sortie[p]<>61)and(trouverve=false) do
 begin
   if (sortie[p]>=301)and(sortie[p]<=400)
     then begin
          trouverve:=true;
          Ve:=sortie[p];
        end;
  inc(p);
 end;
end;

{-------------------------------------------------------------------}

Procedure TrouverVsEntete;
```

```
var p : integer;

begin
  p:=1;trouvervs:=false;
  while (sortie[p]<>61)and(trouvervs=false) do
  begin
    if (sortie[p]>=401)and(sortie[p]<=500)
      then trouvervs:=true;
   inc(p);
  end;
end;

{-----------------------------------------------------------------}

Procedure VerificationIndiceNivSup (var num2 : integer) ;

var p : integer;

begin
  indnivsup:=true;
  p:=1;
  while (p<=lgve)and(indnivsup=true) do
  begin
    if num2<>p then if (resul[p,num2,1]=0)and(resul[p,num2,2]=0)
                    then indnivsup:=true
                    else indnivsup:=false;
   inc(p)
  end;
end;

{-----------------------------------------------------------------}

Procedure VerificationDimEntete;

var num1,num2 : integer;

begin
  if (sortie[i]>=301)and(sortie[i]<=400)
    then if trouverve=true
           then begin
                  num1:=Ve-300;
                  num2:=sortie[i]-300;
                  if (resul[num1,num2,1]>=2) and
                    (resul[num1,num2,2]>=2)
                   then begin
err:=('Il y a une erreur avec la Ve (')
    +copy(gdv[num2],1,length(gdv[num2]))+(') et la Ve (')
    +copy(gdv[num1],1,length(gdv[num1]))+(') dimension à gauche.');
                        messageerreur(err);
                      end;
             end
           else begin
err:=('La Ve (')+copy(gdv[sortie[i]-300],1,length(gdv[sortie[i]-300]))
```

```
    +(') devrait être dimension de la grandeur expliquée.');
              messageerreur(err);
            end;
   if (sortie[i]>=401)and(sortie[i]<=500)
     then begin
          TrouverVsEntete;
          if trouvervs=false
            then begin
err:=('La Vs (')+copy(dim[sortie[i]-400],1,length(dim[sortie[i]-400]))
    +(') devrait être dimension de la grandeur expliquée.');
                messageerreur(err);
              end;
        end;
end;

{-----------------------------------------------------------------}

Procedure VerificationFaEntete;

var p,num1,num2 : integer;

begin
  if (indice>=301)and(indice<=400)
    then if trouverve=true
         then begin
              num1:=Ve-300;
              num2:=indice-300;
              if (resul[num1,num2,1]<>2) or
                (resul[num1,num2,2]<>2)
                then begin
err:=('Pas de chemin N entre la Ve de gauche et celle de la FA (')
    +copy(gdv[num2],1,length(gdv[num2]))+(').');
                   messageerreur(err);
                 end;
            end
          else begin
              num2:=indice-300;
              VerificationIndiceNivSup(num2);
              if indnivsup=false
                then begin
err:=('Aucune Ve dans l''entête ! Impossible d''aggréger sur ')
    +copy(gdv[num2],1,length(gdv[num2]));
                   messageerreur(err);
                 end;
                {Pas de problème avec la Ve de niveau 1}
            end
    else begin
         p:=1;
         while sortie[p]<>61 do
         begin
           if (sortie[p]>=401)and(sortie[p]<=500)
             then if indice=sortie[p]
                  then begin
```

```
err:=('Impossible d"aggréger ! La Vs (')
    +copy(dim[indice-400],1,length(dim[indice-400]))+
    (') est dimension à gauche.');
                        messageerreur(err);
                      end;
          inc(p);
         end;
        end;
end;

{-----------------------------------------------------------------}

Procedure VerificationIndiceFa;

var num1,num2 : integer;

begin
  if (sortie[i]>=401)and(sortie[i]<=500)
     then if (indice>=401)and(indice<=500)
          then if indice=sortie[i]
                 then present:=true;

  if (sortie[i]>=301)and(sortie[i]<=400)
     then if (indice>=301)and(indice<=400)
          then begin
               num1:=sortie[i]-300;
               num2:=indice-300;
               if resul[num2,num1,1]=1
                 then present:=true
                 else if resul[num2,num1,1]=0
                        then {VerificationDimEntete}
                        else begin
err:=('Les aggrégations composées sont interdites !');
                               messageerreur(err);
                             end;
             end;
end;

{-----------------------------------------------------------------}

begin
  transformeregle(entree);
  write('SORTIE : ');
  fillchar(err,70,' ');err:='';
  i:=1;
  while (sortie[i]<>1000) do begin write(sortie[i]);write('|');inc(i) end;
  verificationsyntaxe;
  verificationappbd;
  TrouverVeEntete;
  fa:=false;
  present:=false;
  compteur:=0;
  indice:=0;
```

```
i:=1;
while sortie[i]<>1000 do
begin
 case sortie[i] of
    0 : begin end;

    228 : begin
          fa:=true;
          if (sortie[i+1]<=300)or(sortie[i+1]>=500)
            then begin
                 err:=('Une aggrégation n"a pas d"indice.');
                 messageerreur(err);
                end;
        end;

    227 : begin
          fa:=true;
          if (sortie[i+1]<=300)or(sortie[i+1]>=500)
            then begin
                 err:=('Une aggrégation n"a pas d"indice.');
                 messageerreur(err);
                end;
        end;

    40 : if fa=true then inc(compteur);

    41 : begin
          if compteur>0 then dec(compteur);
          if (compteur=0)and(fa=true)
            then begin
                 fa:=false;
                 if present=false
                   then begin
                          if (indice>=301)and(indice<=400)
then err:=('L"indice de l"aggrégation (')
        +copy(gdv[indice-300],1,length(gdv[indice-300]))
        +(') ne correspond à aucune dimension.')
else err:=('L"indice de l"aggrégation (')
        +copy(dim[indice-400],1,length(dim[indice-400]))
        +(') ne correspond à aucune dimension.');
                          messageerreur(err);
                        end;
                 present:=false
                end
          end;
  end;
  if (sortie[i]>=301)and(sortie[i]<=500)
    then begin
         if fa=true
           then if compteur=0
                then begin
                     indice:=sortie[i];
                     VerificationFaEntete;
```

```
                end
            else begin
                VerificationIndiceFa;
                end
        else VerificationDimEntete;
    end;
  inc(i);
 end;
end;
```