Contributions à la définition d'un nouveau langage d'exploitation des bases de données relationnelles

par

Zouhir Abouaddaoui

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, juillet 2012

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-88873-5

Our file Notre référence
ISBN: 978-0-494-88873-5

Le 12 juillet 2012

le jury a accepté le mémoire de Monsieur Zouhir Abouaddaoui dans sa version finale.

Membres du jury

Professeur Luc Lavoie
Directeur de recherche
Département d'informatique

Professeur Bessam Abdulrazak
Membre
Département d'informatique

Professeur Marc Frappier
Président rapporteur
Département d'informatique

Sommaire

Le but du projet DOMINUS est de définir un modèle de SGBD adapté au développement de services de dépôt de données autonomes capables de maintenir un haut standard d'intégrité et de fiabilité dans un contexte applicatif contemporain. Le présent mémoire, réalisé dans le cadre du projet DOMINUS, vise à contribuer à la définition d'un premier langage conforme à ce modèle, Discipulus, et à mettre en œuvre un premier traducteur expérimental de ce langage.

Le modèle DOMINUS demeure basé sur le modèle relationnel de E. F. Codd tout d'abord parce qu'il est simple, facile à appréhender, et repose sur de solides bases théoriques qui permettent notamment de définir de façon formelle les langages de manipulation associés et, en second lieu, parce qu'il est éprouvé, comme le démontrent plus de trente années de prédominance ininterrompue.

L'évolution de la gestion d'information a vu apparaître de nouvelles applications (systèmes de gestion intégrée, traitement d'images, vidéo...) nécessitant l'utilisation de bases de données complexes de plus en plus importantes. Ces nouvelles applications ont mis en évidence les insuffisances majeures des systèmes relationnels existants fondés sur le langage SQL :

- L'inadéquation du modèle relationnel à représenter directement des données complexes, comme des dossiers médicaux structurés, des images radiographiques ou des textes annotés.
- Les performances insuffisantes dans la manipulation de ces mêmes données.

Ces lacunes ont conduit certains à vouloir remplacer le modèle relationnel par le modèle orienté objet. En effet, la notion d'objet (plus exactement de classe) permet de modéliser des éléments complexes et composites du monde réel. En 1990 sont apparus les premiers systèmes de gestion de bases de données à objets, mais, vu les performances et la maturité

des systèmes de bases de données relationnelles, les systèmes à objets n'ont pas pris une place significative au sein des organisations.

La voie explorée ici est plutôt celle de l'intégration du modèle objet au modèle relationnel, ce dernier demeurant prééminent. L'adoption des deux structures (la relation et la classe) semble donc nécessaire afin de répondre aux besoins et aux exigences des applications complexes tout en gardant la simplicité et la cohésion conceptuelle nécessaire à la vérification et à la validation.

Le modèle DOMINUS est donc inspiré des travaux fondamentaux de E. F. Codd et de ses continuateurs, dont C. J. Date et H. Darwen [S1] ainsi que des modèles algorithmiques et de typage de B. Meyer [L13]. Au final, le langage Discipulus retient plusieurs acquis du langage SQL, s'inspire également de langage Tutorial D et emprunte la structure générale et plusieurs mécanismes syntaxiques du langage Eiffel [L13]. Notre proposition comporte également de nombreuses différences sensibles tant sur le fond que sur la forme [L1,L7]. Ces apports sont présentés au fil du mémoire.

Le langage Discipulus a été conçu dans le but de permettre l'expression rigoureuse de modèles complexes (intégration complète des classes, des tuples et des relations dans un seul système de typage homogène et cohérent) tout en favorisant la réutilisation (l'utilisation d'un système de paquetage destiné à développer des modules cohérents tout en permettant leur réutilisation simple pour le développement d'autres systèmes), l'évolutivité (l'adoption de l'héritage multiple permet d'éviter la redondance de code et facilite l'extensibilité du logiciel et, par conséquent, l'évolutivité sans compromettre son intégrité et sa fiabilité) et la fiabilité (incorporation des principes de programmation par contrat et leur extension aux opérateurs relationnels, traitement cohérent de l'annulabilité).

Remerciements

En préambule à ce mémoire, je souhaitais adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire.

Je tiens à remercier sincèrement Monsieur Luc Lavoie, qui, en tant que directeur de recherche, s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce mémoire, ainsi pour l'inspiration, l'aide et le temps qu'il a bien voulu me consacrer et sans qui ce mémoire n'aurait jamais vu le jour.

Mes remerciements s'adressent également à mes amies Aurélie Ottavi et Christina Khnaisser qui m'ont toujours soutenu et encouragé au cours de cette période et avec qui j'ai eu le plaisir de partager cette première expérience de la recherche.

Je n'oublie pas mes parents et mes sœurs pour leur contribution, leur patience et qui malgré l'océan qui nous sépare ont su me soutenir, m'encourager et surtout être toujours très présents pour moi.

Enfin, j'adresse mes plus sincères remerciements à tous mes proches et amis, qui m'ont toujours soutenu et encouragé au cours de la réalisation de ce mémoire.

Merci à tous et à toutes.

Table des matières

Liste des abréviations

ANTLR	ANother Tool for Language Recognition
BD	Base de données
Cocktail	COmpiler Compiler ToolkIt KArLsruhe
DB	Systèmes de gestion de base de données propriétaire d'IBM,
DOMINUS	Le modèle de SGBDR et le projet de sa mise en œuvre
Dominus-1	Notre première proposition de gestion des annulables
Dominus-2	Notre deuxième proposition de gestion des annulables
Discipulus	Langage des bases de données proposé dans le modèle DOMINUS
ISO	International Organization for Standardization)
JavaCC	Java Compiler Compiler
LL(k)	Left to right scanning, Leftmost derivation, with *k* anticipation
LPG	Licence Publique Générale
MBDR	Modèle de bases de données relationnel
MOO	Modèle Orienté Objet
OO	Orienté Objet
POO	Programmation Orienté Objet
PL/SQL	Procedural Language / Structured Query Language
SGBD	Système de Gestion de Bases de Données
SGBDR	Système de Gestion de Bases de Données Relationnel
SQL	Structured Query Language
XML	Extensible Markup Language

Liste des tableaux

Liste des figures

Introduction

Contexte

Le modèle relationnel a été proposé par E. F. Codd [L2] en 1969. Les premiers systèmes commerciaux basés sur le modèle relationnel sont apparus à la fin des années 70 et au début des années 80. Cela fait donc plus de 30 ans que l'industrie utilise et améliore ces systèmes qui sont maintenant largement à maturité comme Oracle, DB2, PostgreSQL et SQL Server.

Des recherches montrent que les SGBDR actuels présentent des faiblesses dans la manipulation des données telle qu'elle peut être exprimée à l'aide du langage SQL que ceux-ci utilisent [A7, A11, A14, A23]. Les lacunes de SQL introduisent une complexité inutile et des incohérences dans le modèle relationnel des bases de données à tel point qu'il fait l'objet de nombreuses critiques [L1].

Objectifs

L'objectif de notre recherche est de définir un nouveau langage de manipulation et d'exploitation de base des données relationnelles fondées sur le modèle relationnel et qui évite les erreurs de SQL tout en se basant sur un système de typage fort et extensible afin de répondre aux exigences des nouvelles applications. L'objectif du mémoire est d'apporter une contribution significative à cette recherche notamment en regard du traitement des attributs nuls, des doublons et d'un système de typage complet et cohérent.

Méthodologie

Dans notre travail, nous commençons par l'analyse du modèle relationnel qui constitue la base de notre recherche. Nous analysons ensuite plusieurs propositions visant à corriger les lacunes de SQL plus particulièrement en ce qui concerne les attributs nuls, les doublons, l'ordre des tuples dans une relation et l'ordre des attributs dans un tuple. Finalement, nous dégageons les conclusions sur lesquelles nous avons élaboré notre proposition.

Nous définissons aussi un ensemble de critères permettant de faciliter la comparaison de plusieurs systèmes de typage susceptibles de remplacer celui de SQL et sur lequel notre modèle de typage sera construit.

Finalement, pour évaluer concrètement notre proposition, le langage Discipulus intégrant les principaux aspects de notre proposition est défini sur la base du langage Eiffel. Un prototype de traducteur du langage Discipulus a été réalisé et expérimenté.

Contributions

Notre première contribution porte sur la définition du langage Discipulus basé sur un système de typage fort, flexible et conforme à l'approche de L. Cardelli et P. Wegner [A16]. Le langage intègre aussi des mécanismes puissants inspirés du modèle d'Eiffel développé par B. Meyer [L13] dont l'héritage multiple et la programmation par contrat.

Notre deuxième contribution est un traducteur expérimental du langage Discipulus en SQL pour pouvoir tester les solutions proposées et comparer l'expressivité de notre langage à celle de SQL. Les premiers résultats de notre expérimentation tendent à montrer que le langage est versatile, concis, d'une grande puissance d'expression et qu'il semble facile d'écrire et de comprendre des programmes rédigés en Discipulus.

Perspectives

Certains points restent à développer pour compléter la description du modèle et la définition du langage, cependant les résultats préliminaires obtenus montrent la pertinence de tester le langage plus à fond à l'aide du banc d'essai développé par Aurélie Ottavi [L12].

Structure du mémoire

La structure du mémoire est la suivante :

- Le chapitre 1 est consacré à la définition du modèle relationnel et du modèle SQL. Nous présentons aussi l'historique des propositions menées pour modéliser l'absence d'une valeur, les doublons, l'ordre des attributs et les tuples dans une relation.

- Le chapitre 2 est consacré à la problématique et l'approche retenue pour la solution proposée. Nous y présentons les différents aspects de l'approche retenue pour définir un modèle relationnel-objet intégré.
- Le chapitre 3 est consacré à une brève présentation du langage Discipulus, défini dans un rapport de recherche distinct. Nous nous limitons à la description des éléments les plus significatifs.
- Le chapitre 4 est consacré à la présentation de la mise en œuvre du traducteur, de son architecture et des outils utilisés.
- Le chapitre 5 présente les résultats obtenus à l'aide du traducteur et du SGBD Oracle 10g Enterprise Edition.
- L'annexe A présente les 14 cas d'annulabilité recensés par ANSI/SPARC.
- L'annexe B présente le comportement des quelques requêtes SQL portant sur tables comportant des attributs annulés.
- L'annexe C présente le comportement de 12 requêtes SQL dans des situations comportant des doublons.
- L'annexe D présente des exemples montrant quelques incohérences diverses en SQL.
- L'annexe E présente quelques modèles d'annulabilité.
- L'annexe F présente la structure de la table de symboles du traducteur expérimental de Discipulus.
- L'annexe G présente la couverture des tests pour les règles de validité.
- L'annexe H présente le sommaire de vérification de l'exécution de la traduction des trois exemples réalisés dans le cadre des essais de système (Distribution, Ferrailleur et Patinage).
- L'annexe I présente le mécanisme de traitement des exceptions de Discipulus.
- L'annexe J présente des exemples illustrant la traduction des éléments du langage algorithmique.
- L'annexe K présente l'implémentation complète en Discipulus de l'exemple Distribution ainsi que le résultat de la traduction en SQL.

Chapitre 1

État de l'art

1.1 Modèle relationnel

Avant de commencer la présentation du modèle relationnel, il faut comprendre que celui-ci n'est pas statique : il a subi des modifications au cours des années et continue à en subir. Les sections qui suivent reflètent en grande partie certaines des idées des auteurs du troisième manifeste [L10, A11, S3] C. J. Date et H. Darwen et ne sauraient être considérées comme le point de vue définitif sur le modèle relationnel.

Le modèle relationnel a été proposé par E. F. Codd en 1969 [L2]. C'est l'une des premières tentatives de formalisation de la notion de la base de données. Il est considéré comme le fondement essentiel de la modélisation et de l'exploitation des bases de données depuis plus de 30 ans. Le modèle relationnel présente de nombreux avantages :

- Il se base sur la théorie mathématique des ensembles grâce à l'algèbre relationnelle.
- Il permet un haut degré d'indépendance entre le modèle de données et la représentation interne des données.
- Il est un bon support à la minimisation de la redondance des données.
- Il vise à traiter les données avec cohérence et intégrité.

La présentation du modèle relationnel est divisée en trois parties : la structure, la manipulation et l'intégrité.

- Structure : les objets fondamentaux du modèle relationnel : les relations, les tuples, les attributs et les domaines.

- Manipulation : la façon dont les données sont accessibles et manipulées à l'aide de l'algèbre relationnelle.
- Intégrité : les mécanismes permettant de garantir la cohérence et l'intégrité des données.

1.1.1 Les concepts structurels

Les objets de base manipulés par le modèle relationnel sont les domaines, les relations, les attributs et les tuples. La figure suivante représente la terminologie structurelle du modèle relationnel :

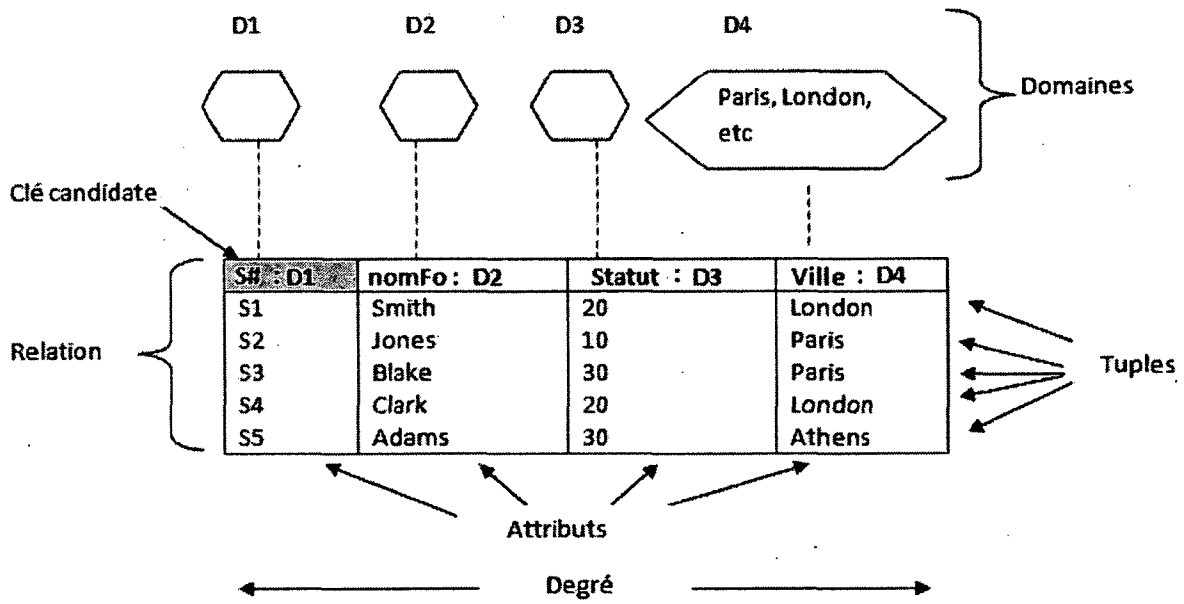

Figure 1-1 : Terminologie structurelle [L2]

Domaine

Selon C. J. Date, un domaine est un ensemble fini de valeurs. Par exemple, le domaine INTEGER est un sous-ensemble fini des entiers mathématiques. La notion de type comprend un domaine et un ensemble d'opérateurs pouvant être appliqués aux valeurs d'un domaine. Par exemple, un type INTEGER pourrait associer au domaine INTEGER les opérateurs de comparaison et de calcul arithmétique (addition, soustraction, multiplication, division).

Il faut faire la distinction entre le domaine et la représentation physique de ses valeurs dans le système. Par exemple, les numéros des fournisseurs pourraient être physiquement représentés sous forme de chaîne de caractères, mais cela ne signifie pas que les opérations applicables aux chaînes de caractère le soient aux numéros de fournisseurs. Il faut donc définir des opérateurs appropriés pour un type en fonction de la signification ou de la sémantique de celui-ci.

Selon C. J. Date [L2] une relation est une structure constituée d'un en-tête et d'un corps lui-même constitué d'un ensemble de tuples ayant tous le même en-tête. Par définition, un ensemble n'est pas ordonné, il en et donc de même pour la relation.

Tuple

Soit D_i (i=1,2,3..., n) une collection de n domaines non nécessairement distincts et soit A_i (i=1,2,3..., n) un ensemble de n attributs[1] distincts, un tuple T est constitué des deux parties suivantes :

- En-tête : un ensemble de n paires $(A_i : D_i)$ où l'attribut A_i désigne une représentation d'une valeur du domaine D_i.
- Corps : un ensemble de triplets de la forme $<A_i,D_i,v_i>$, où :
  - A_i : le nom d'un attribut de T qui doit être distinct des autres attributs.
  - D_i : le nom du domaine associé à l'attribuer A_i.
  - v_i : une représentation d'une valeur de domaine D_i, associé a l'attribut A_i du tuple T.

Les représentations des valeurs des attributs sont atomiques (axiome d'atomicité, parfois désigné abusivement comme « première forme normale » (1NF) dans la littérature).

---

[1] Rigourseusement, il faut plutôt parler de noms d'attributs. Pour alléger le texte, nous utiliserons cependant le mot attribut, là où le contexte le permet.

Relation

Soit D_i (i=1,2,3..., n) une collection de n domaines non nécessairement distincts et A_i (i=1,2,3..., n) un ensemble de n attributs distincts, une relation R est constituée des deux parties suivantes :

- En-tête : un ensemble de n paires (A_i : D_i) où l'attribut A_i désigne une représentation d'une valeur du domaine D_i.
- Corps : un ensemble de tuples, chacun ayant le même en-tête que la relation R.

On remarque que la définition de l'en-tête d'un tuple est la même que celle d'une relation.

Une relation possède donc les propriétés suivantes :

- Une relation comporte zéro ou plusieurs tuples.
- Il n'y a pas des tuples en double.
- Il n'y a pas d'ordre pour les tuples.
- Il n'y a pas d'ordre pour les attributs (pas plus que pour les tuples).

Schéma de relation

Le schéma d'une relation est une définition de la relation au sein d'une base de données. Un schéma de relation contient le nom de la relation, la liste de ses attributs, le domaine de chaque attribut ainsi que l'ensemble des contraintes d'intégrité associées à la relation.

La définition des contraintes d'intégrité est présentée à la section 1.1.5.

1.1.2 L'algèbre relationnelle

L'algèbre relationnelle est constituée d'un ensemble d'opérateurs internes sur les relations. On y retrouve les opérateurs ensemblistes classiques (union, différence, intersection et produit), ainsi que des opérateurs spécifiques tels que la projection et la sélection qui déterminent respectivement une relation définie sur sous-ensemble d'attributs, un sous-ensemble de tuples d'une relation. La figure suivante présente graphiquement quelques-uns des opérateurs les plus fréquents.

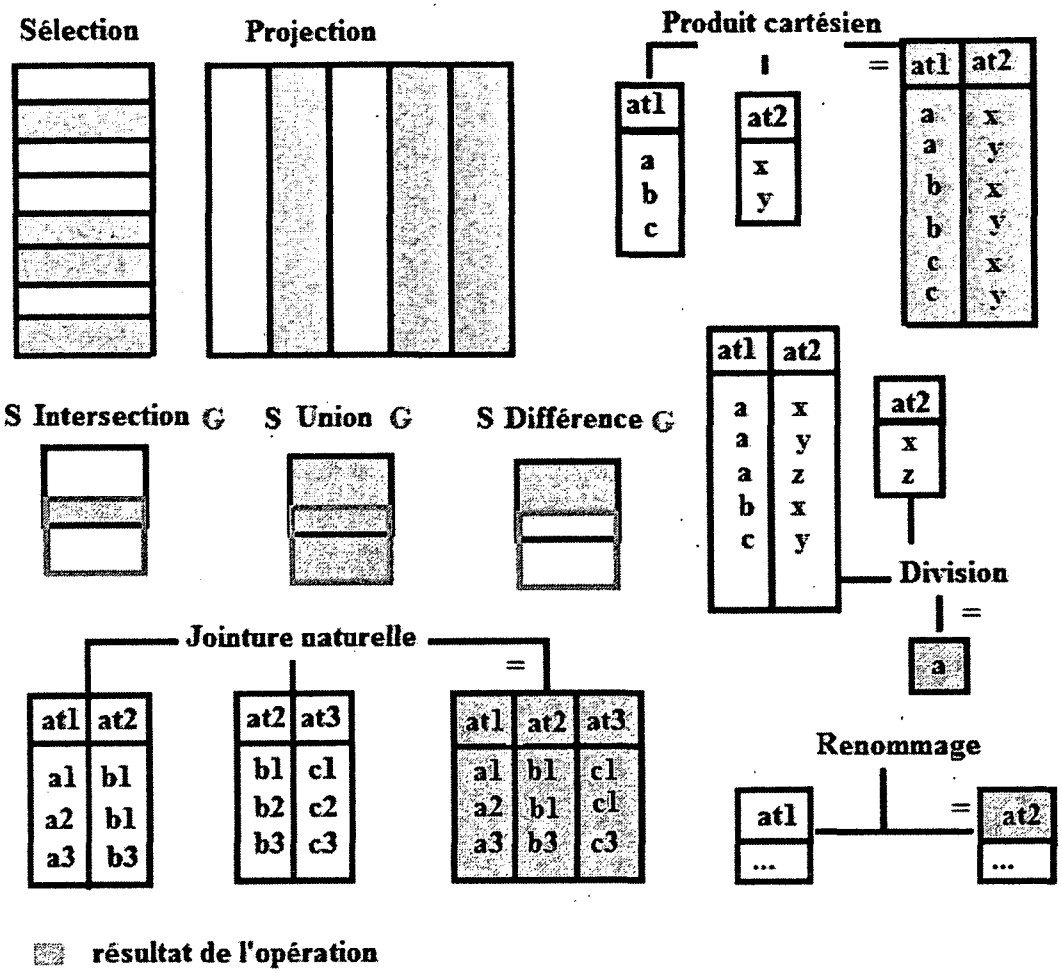

Figure 1-2 : Synthèse graphique des opérateurs de l'algèbre relationnelle inspirée de [L2]

Note : Dans le cas des opérations d'union, intersection et de différence, les deux relations S et G ont le même schéma.

Les opérations de l'algèbre relationnelle sont définies ainsi :

- **Sélection :** à partir d'une relation S et d'une expression booléenne E sur les attributs de S, l'opération de sélection permet d'obtenir la relation comprenant les seuls tuples de S pour lesquelles l'expression E est vraie.

- **Projection :** à partir d'une relation S et d'un sous-ensemble de ses attributs, la projection permet d'obtenir la relation à partir de la relation S en ne gardant que le sous-ensemble des attributs spécifiés dans la projection.
- **Opérateurs ensemblistes :** ce sont les opérateurs de la théorie des ensembles, définis sur des relations de mêmes schémas :
  - **Intersection :** à partir des relations S et G, l'intersection permet d'obtenir une relation B qui contient les tuples présents à la fois dans S et dans G.
  - **Union :** à partir des relations S et G, l'union permet d'obtenir une relation C qui contient les tuples présents dans S ou dans G.
  - **Différence :** à partir des relations S et G, la différence permet d'obtenir une relation B qui contient les tuples présents dans S mais pas dans G.
- **Produit cartésien :** à partir des relations S et G, le produit cartésien permet d'obtenir la relation composée de tous les tuples résultant de la concaténation de chaque tuple de la relation S avec chaque tuple de G. Les deux relations n'ont pas nécessairement le même schéma.
- **Jointure naturelle :** la jointure naturelle de deux relations S et G est obtenue par la projection sur les attributs distincts de la sélection des tuples du produit cartésien SxG pour lesquels les attributs communs sont égaux.
- **Division cartésienne :** à partir de deux relations S et G, la division S/G génère une relation regroupant exclusivement toutes les parties d'occurrence de la relation S qui sont associées à toutes les occurrences de la relation G.
- **Renommage :** c'est une opération qui permet de résoudre des problèmes de compatibilité entre les noms des attributs d'une opération binaire. Cette opération exige que le nouveau nom n'existe pas déjà dans la relation pour les attributs.

Parce que l'ordre des attributs n'est pas signifiant dans le modèle relationnel, les attributs doivent être référencés par leur nom et non par leur position. En général, un nom d'un attribut ne doit pas être unique au sein d'une base de données entière, mais seulement dans la relation à laquelle il appartient.

À la figure 2, nous avons défini les opérations habituelles du modèle relationnel. Ces opérations ne constituent ni la base de l'algèbre définie à l'origine par E. F. Codd, ni toutes les opérations usuelles. En fait, il existe plusieurs bases possibles et même plusieurs algèbres possibles. Voir entre autres A* proposé par C. J. Date [A28].

1.1.3 Les contraintes d'intégrité

Un des objectifs essentiels d'une base de données est d'assurer la cohérence des données, c'est-à-dire les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base.

Du point de vue du modèle relationnel, les contraintes d'intégrité sont des propriétés sur les données généralement issues de la pragmatique (du domaine d'application). La conjonction de toutes ces propriétés exprimées sous forme de contraintes booléennes forme la contrainte globale d'intégrité de la base de données. Il appartient au SGBD de s'assurer que toute modification maintient la contrainte globale.

Si on considère une base de données comme un objet appartenant à une classe, chaque contrainte d'intégrité définit un invariant (une assertion) qui doit être exprimé grâce au mécanisme de définition de classe. Cet invariant est vérifié automatiquement à chaque mise à jour d'un objet (à l'issue de chaque transaction sur la base de données).

Les types des contraintes d'intégrité

Il existe plusieurs types (ou spécialisation) de contraintes d'intégrité [L1], nous les définissons ci-dessous.

- Contrainte de domaine : c'est le domaine de valeur associé à chaque attribut d'une relation. Exemple : si le domaine d'un attribut est un intervalle d'entiers, la valeur de l'attribut doit être comprise dans les limites de cet intervalle.
- Dépendance fonctionnelle : une dépendance fonctionnelle est une association plusieurs vers un entre deux ensembles d'attributs à l'intérieur d'une relation. Pour exprimer cette contrainte, on a traditionnellement recours au concept de clé candidate : tout sous-ensemble minimal d'attributs d'une relation déterminant toujours uniquement les

autres attributs. Le concept peut aisément être étendu entre les relations d'une même base de données grâce au concept de clé étrangère. Une clé étrangère d'une relation R par rapport à une relation S et un sous-ensemble des attributs de R formant une clé candidate de S et dont toute valeur au sein de R doit être représentée au sein de la projection de S sur l'ensemble des attributs de clé.

- Contrainte générale : ce type de contrainte représente la notion générale d'une contrainte d'intégrité qui est essentiellement une expression booléenne qui doit être maintenue vraie. Toutes les autres contraintes sont des cas particuliers de celle-ci.

Bien que d'autres types de contraintes aient été définies (dépendances multivaluées, dépendances de jointure...), elles n'ont pas engendré de mécanismes particuliers dans les langages de bases données. Sans perte de généralité, nous avons décidé de ne pas les traiter, puisqu'elles sont de toute façon exprimables à l'aide des contraintes générales.

1.2 Le modèle SQL

Le modèle relationnel a été proposé par E. F. Codd en 1969 [L2]. Suite à cette proposition, de nombreux langages basés sur le modèle relationnel sont apparus :

- IBM Sequel (Structured English Query Language) [A30];
- IBM Sequel/2 [A31];
- IBM System/R [A32];
- IBM DB2 [L15].

Ce sont ces langages qui ont donné naissance au standard SQL, normalisé en 1986 par l'ANSI [L3] puis, dans les années subséquentes, par l'ISO. Le tableau 1 présente le parcours de l'effort de normalisation.

Tableau 1-1 : Révision de la norme internationale SQL [S1]

Année	Nom	Appellation	Commentaires
1989	ISO/CEI 9075:1989	SQL-1 ou SQL-89 ou SQL-1989	Édité par l'ANSI puis adopté par l'ISO en 1987.
1992	ISO/CEI 9075:1992	SQL-92 ou SQL-92 ou SQL-1992	Les modifications apportées à la syntaxe de jointure, en particulier l'ajout de la clause OUTER JOIN.
1999	ISO/CEI 9075:1999	SQL-99 ou SQL-1999	Expressions rationnelles, requêtes récursives, déclencheurs, types non scalaires et quelques fonctions orientées objet.
2003	ISO/CEI 9075:2003	SQL-2003	Introduction de fonctions pour la manipulation XML, « window functions », ordres standardisés et colonnes avec valeurs autoproduites (y compris colonnes d'identité).
2008	ISO/CEI 9075:2008	SQL-2008	Ajout de quelques fonctions de fenêtrage (ntile, lead, lag, first value, last value, nth value), limitation du nombre de ligne (OFFSET / FETCH), amélioration mineure sur les types distincts, curseurs et mécanismes d'auto incréments.
2011	ISO/IEC 9075:2011	SQL-2011	Révision sur les parties suivantes de la norme : SQL/Framework, SQL/Foundation, SQL/PSM, SQL/Schemata, et SQL / XML.

Une constante se dégage toutefois, tout au long de cette évolution. Les SGBD des principaux éditeurs ne mettent jamais en œuvre la totalité de la norme. Chaque éditeur ajoute à son produit des fonctionnalités absentes de la norme, et parfois en contradiction avec celle-ci.

1.2.1 Les concepts structurels

SQL est le langage de SGBD le plus utilisé et il est basé sur la notion de collection, tandis que le modèle relationnel utilise la notion de l'ensemble.

Dans le modèle relationnel proposé par E. F. Codd, l'algèbre relationnelle est fondée sur la théorie des ensembles alors que les SGBDR qui utilisent SQL considèrent une table comme une collection (*bag* en anglais) et non comme un ensemble, ce qui permet explicitement la présence des doublons.

Dans une relation, les attributs ne sont pas ordonnés. Au contraire, en SQL, l'ordre des colonnes dans une table fait partie de la définition de celle-ci. Entre autres conséquences, le produit cartésien n'est ni commutatif, ni associatif[2].

Par ailleurs, SQL ne fait pas la distinction entre une relation et une variable de relation.

Nous présentons ci-après les concepts structurels du modèle SQL.

Table

On trouve deux types de table :

- Tables : les tables sont des structures qui existent dans la base de données qu'on peut manipuler à travers des requêtes.
- Vues : les vues sont des expressions permettant le calcul d'une variable de relation; les tables « induites » ne sont pas stockées, mais construites à la demande.

Les relations au niveau du modèle relationnel sont stockées sous forme de tables composées de lignes et de colonnes. Chaque table possède un en-tête et un corps. L'en-tête de la table est une liste ordonnée de noms de colonnes. Le corps de la table est une collection ordonnée de lignes. Les lignes partagent le même en-tête que leur table.

Colonne

Une colonne en SQL est l'élément vertical dans une table représentant un ensemble de valeurs. La colonne est identifiée par un nom unique[3], les données contenues dans une colonne doivent être toutes d'un même type de données.

---

[2] La notion de l'égalité entre tables n'est pas définie en SQL et pour cause. Plusieurs cas problématiques doivent être considérés, par exemple : deux tables avec le même nombre de colonnes et les mêmes noms, mais dont l'ordre des colonnes est différent, deux tables de même structure avec les mêmes données, mais dont les lignes sont ordonnées différemment, etc.

[3] C'est du moins ce que stipule la norme. Par contre, il existe de nombreuses situations où il est possible d'ajouter une colonne sans en donner le nom. Ce n'est là qu'une des nombreuses incohérences de SQL. [A27].

Ligne

Élément horizontal dans une table représentant une énumération des valeurs des différentes colonnes.

Type de données

Toutes les colonnes dans une table en SQL doivent avoir un type de données, les types de données primitifs les plus fréquents sont :

- chaîne de caractères (fixe ou variable);
- chaîne de bits (fixe ou variable);
- booléen;
- nombre (avec ou sans partie fractionnaire);
- date.

Il existe toutefois une grande disparité dans la dénomination de ces types et même l'offre effective des différents éditeurs.

1.2.2 Les opérations relationnelles

Le langage SQL permet d'exprimer simplement les opérations relationnelles de base. La principale commande du langage de manipulation de données est la commande SELECT. Le bloc SQL se compose de la clause SELECT, suivie des attributs associés à la table résultat, de la clause FROM, de la liste des tables utilisées pour calculer le résultat et, finalement, de la clause WHERE suivie d'une qualification.

Les opérations relationnelles sont définies ainsi :

- **Projection** : une projection est une instruction permettant de sélectionner un ensemble de colonnes dans une table, l'obtention de colonnes déterminées s'effectue simplement par les deux clauses SELECT et FROM[4].
- **Restriction** : une restriction consiste à sélectionner les lignes satisfaisant à une condition logique effectuée sur leurs attributs. La restriction s'exprime à l'aide de la clause WHERE suivie d'une condition logique exprimée à l'aide des opérateurs logiques AND et OR.
- **Jointures** : une jointure permet de répondre aux questions mettant en œuvre, dans le résultat ou dans les assertions, des informations issues de plusieurs tables. La présence des NULL a nécessité l'introduction de plusieurs variétés de jointures, puisque l'égalité entre un attribut valué et un attribut annulé est, pour le moins, problématique. Une valeur NULL ne correspond pas à aucune autre valeur NULL, si les colonnes des tables jointes contiennent des valeurs NULL, les résultats de la jointure ne contiendront pas ces valeurs NULL sauf pour le cas des jointures externes. (Annexe B). On distingue donc trois types :
  - **Jointures internes** : ce type de jointure utilise certains opérateurs de comparaison (=, <>). Parmi ces jointures on trouve les équi-jointures et les jointures naturelles.
  - **Jointures externes** : les jointures externes peuvent être des jointures gauches, droites ou complètes. Toutes les lignes sont extraites de la table de gauche référencée par une jointure externe gauche, et de la table de droite référencée par une jointure externe droite. Toutes les lignes des deux tables sont retournées dans une jointure externe complète.
  - **Les jointures croisées** : Les jointures croisées renvoient toutes les lignes de la table de gauche, combinées à toutes les lignes de la table de droite. Ces jointures sont également appelées produits cartésiens.

---

[4] La clause FROM spécifie une ou plusieurs tables pour le SELECT et elle contribue à l'expression de la projection, du produit cartésien et des jointures. En fait, la syntaxe du SELECT ... FROM est plus proche du calcul relationnel que de l'algèbre relationnelle.

- **Opérations ensemblistes** : Les opérations ensemblistes en SQL, sont celles définies dans l'algèbre relationnelle :
  - UNION
  - INTERSECT
  - MINUS

Puisque les tables sont des collections, il y possibilité d'avoir des doubles. La plupart du temps, leur élimination n'est pas automatique, elle doit être commandée grâce à l'option DISTINCT. Par contre, dans certaines situations, ils sont éliminés automatiquement, par exemple lors de l'utilisation de UNION, seules les valeurs distinctes sont sélectionnées, contrairement UNION ALL sélectionne toutes les lignes dans les résultats, y compris les doublons.

Finalement, il est possible d'ordonner les lignes du résultat d'un SELECT grâce à la clause ORDER BY. La clause ORDER BY est suivie des mots clés ASC ou DESC, qui précisent respectivement si le tri se fait de manière croissante (par défaut) ou décroissante.

1.2.3 Les contraintes

SQL comprend un mécanisme de contrôle de validité des données interdisant l'insertion de données violant les contraintes ce qui, par conséquent, permet d'assurer (dans une certaine mesure) la cohérence des données de la BD. Les types de contraintes disponibles sont les suivants :

- NOT NULL : l'attribut est non annulable.
- UNIQUE et PRIMARY KEY : unicité d'un groupe d'attributs (clé candidate). Une table peut avoir plusieurs clés candidates (groupes d'attributs uniques), un seul de ceux-ci est désigné comme étant la clé primaire (PRIMARY KEY). Un des objectifs des SGBD est de séparer la vue logique (les données telles que perçues par les utilisateurs) et la vue physique (les données telles qu'organisées dans le support de stockage), la distinction d'une clé primaire ne respecte pas cet objectif vu que cette distinction n'a d'effet qu'en regard des choix de représentation interne (dont les index). Les distinctions entre UNIQUE et PRIMARY KEY sont :

- La contrainte NOT NULL est comprise dans la contrainte PRIMARY KEY, c'est-à-dire l'ensemble des attributs formant la clé primaire ne peut pas avoir d'attributs annulables, alors qu'un ensemble d'attributs avec la contrainte UNIQUE peut comporter des attributs annulables.
- Il ne peut y avoir qu'une seule clé (donc forcément primaire) dans une table, puisqu'il n'y a pas d'autres moyens d'en déclarer. Cependant, plusieurs ensembles d'attributs peuvent être déclarés uniques (mais ne sont pas pour autant des clés... même si on les considère le plus souvent comme telles).

- FOREIGN KEY (clé étrangère) : une clé étrangère (d'une table A) est un groupe de colonnes correspondant à celui d'une clé candidate (d'une table B); une contrainte référentielle implicite en découle, c'est-à-dire que la valeur de la clé étrangère d'une ligne de A doit être égale à la valeur d'une clé candidate de B.
- TYPE et DOMAIN (contrainte de domaine) : restreint les valeurs possibles d'une colonne à un ensemble de valeurs prédéfinies.
- CONSTRAINT (contrainte générale) : exprime une contrainte sous la forme d'une expression booléenne générale.

1.3 La correspondance avec le modèle relationnel

Dans cette section nous tenterons d'établir jusqu'à quel point le langage SQL satisfait les règles du modèle relationnel de E. F. Codd. Nous nous concentrerons sur les éléments suivants :

- la correspondance entre les éléments structurels,
- l'utilisation des doublons,
- l'ordre des attributs dans les tuples,
- l'ordre des tuples dans les relations,
- l'expression des contraintes.

1.3.1 Correspondance structurelle

E. F. Codd a dénoncé très tôt les lacunes et les incohérences introduites par l'utilisation de SQL [L9]. Les différences sont illustrées jusque dans les noms qu'on a donnés aux différents objets ou concepts du modèle. Ces différences sont résumées au tableau suivant.

Tableau 1-2 : Correspondance structurelle entre SQL et le modèle relationnel

SQL	**Modèle relationnel**
Table (collection)	Relation (ensemble)
Type de données	Domaine
Colonne	Attribut
Ligne	Tuple
Clé candidate (dont une primaire)	Clé candidate
Clé étrangère	Clé étrangère

1.3.2 L'utilisation des doublons

En général, la modélisation du monde réel à l'aide d'une base de données prend en compte les deux critères suivants :

- Exhaustivité : la base contient toutes les informations requises pour le service qu'on en attend.
- Unicité : la même information n'est présente qu'une seule fois (pas de doublons[5]).

La théorie des dépendances fonctionnelles et les formes normales qui en découlent ont notamment pour but de réduire, voire d'éliminer les doublons. En SQL, l'identifiant représenté par la clé primaire donne à chaque tuple un caractère unique. Malheureusement,

---

[5] Cela n'empêche pas qu'il puisse y avoir duplication de données, mais que toute modification d'une information entraîne que toutes les données (même dédoublées) qui la représente sont modifiées en conséquence.

cette propriété n'est pas nécessairement maintenue par les requêtes, notamment en raison de du traitement des requêtes de type SELECT-FROM-WHERE :

- calculer le produit cartésien des tables de la clause FROM;
- garder les tuples qui respectent la condition dans clause WHERE;
- sélectionner les attributs mentionnés dans la clause SELECT.

Cette dernière projection peut induire plusieurs tuples identiques dans la collection finale. Pour éliminer les doublons des résultats, SQL propose la clause DISTINCT qui permet d'avoir une seule copie de chaque tuple sans avoir de doublons, mais le problème c'est que SELECT DISTINCT peut prendre plus de temps que SELECT ALL même si le DISTINCT est effectivement sans utilité (par exemple, lorsque le traitement n'a pas induit, dans un cas précis, de doublons). Pour cette raison, C. J. Date recommande d'utiliser la clause DISTINCT seulement là où il faut, mais ce n'est pas toujours évident de le savoir [L2].

Par contre, du point de vue du modèle relationnel, la nécessité de l'existence d'une clé ne relève absolument pas de la normalisation, c'est plutôt une conséquence de la propriété des relations selon laquelle celles-ci ne peuvent pas contenir de l'information en double : une relation est un ensemble, or un ensemble ne contient pas d'éléments en double.

Nous présentons ci-dessous deux approches différentes à ce problème : celle de C. J. Date et H. Darwen et celle de J. D. Ullman

Critique de C. J. Date et H. Darwen

Supposons qu'il y a un sens attaché aux doublons, quel que soit ce sens il n'est pas très explicite et par conséquent on ne respecte pas l'un des objectifs du modèle relationnel qui impose que la signification des données doive être aussi évidente et explicite que possible. Si l'utilisateur prend en considération des doublons, il doit être extrêmement prudent dans la formulation de la requête de manière à obtenir exactement le résultat souhaité [L2], en particulier le « bon » nombre de doublons, puisque celui-ci a un sens. L'annexe C présente douze différentes formulations produisent neuf différents résultats différents par rapport à leur degré de duplication. Ceci pose le problème d'arriver à faire la différence entre les

« vrais » doublons et les doublons dus à une erreur l'entrée des données. Cet argument est majeur et rend incontournable la distinction des doublons entre eux, c'est-à-dire ce ne sont pas des doublons.

Ullman

Dans le modèle relationnel proposé par E. F. Codd l'algèbre relationnelle manipule des ensembles de tuples pour exprimer certaines requêtes sur les relations. Les SGBD existants implémentent le modèle relationnel et considèrent une relation comme une collection et non comme un ensemble, ce qui permet explicitement la présence des doublons.

J. D. Ullman [L7] propose une algèbre sur les collections par extension de l'algèbre relationnelle. Selon J. D. Ullman, en pratique, la manipulation d'une collection serait plus efficiente que celle d'un ensemble. Selon lui, l'utilisation des instructions pour éliminer les doublons (par exemple DISTINCT dans SQL) peut être dangereuse et coûteuse, le temps consommé pour éliminer les doublons peut être plus grand que celui de l'exécution de la requête elle-même. Donc il faudrait conserver les doublons si on veut que les requêtes s'exécutent rapidement.

J. D. Ullman [L4] donne l'exemple de l'union de deux relations. Dans le cas de collections, l'opération se résume à une recopie des deux relations, d'où une complexité dans $O(n)$. Dans le cas relationnel, on doit appliquer un tri préalablement à la suppression des doublons, d'où une complexité dans $O(n \log(n))$.

Réflexion

On remarque cependant que si la relation est munie d'une clé, l'opération de suppression doit être faite d'une manière ou d'une autre, ce qui en pratique réduirait considérablement l'avantage des collections, surtout si les requêtes ultérieures reposent sur l'unicité des tuples ou si elles induisent des jointures (auquel cas la cardinalité des relations résultantes sera indûment élevée). On remarque également que l'utilisation d'index par adressage dispersé permettrait de ramener la complexité de l'opération dans $O(n)$ en pratique. Seule une expérimentation poussée avec des engins de qualité comparable permettrait alors de

départager les deux représentations en regard de séquences de transactions représentatives de scénarios d'utilisation réalistes. Nous n'avons pas retracé de publications traitant spécifiquement de telles expérimentations bien que les récentes techniques de représentation horizontale sont très susceptibles de tirer parti de cette approche [A29].

1.3.3 L'ordre des attributs dans les tuples

Dans le modèle relationnel, les attributs d'une relation (d'un tuple) sont non ordonnés. Par contre, l'ordre des colonnes dans une table SQL est défini et signifiant.

Si l'ordre des attributs du tuple est important, les utilisateurs peuvent avoir de la difficulté à se souvenir de l'ordre exact des colonnes ce qui peut être une source d'erreurs de manipulation et d'interprétation. C. J. Date montre avec un exemple que si une insertion est réalisée en se basant sur l'ordre des colonnes, les résultats obtenus peuvent être incorrects si l'ordre initial des colonnes a été modifié et que l'ordre des colonnes de la requête d'insertion ne reflète pas cette modification.

Supposons la table suivante qui représente une boutique avec l'heure d'ouverture et l'heure de fermeture :

ID	Nom	Date	H_ouverture	H_fermeture

Après une modification des noms des attributs, la table devient :

ID	Nom	Date	H_fermeture	H_ouverture

Avec SQL, lors d'une insertion après cette modification, il est possible de faire l'erreur de prendre en compte l'ordre des attributs avant la modification.

ID	Nom	Date	H_fermeture	H_ouverture
1	HM	2010-12-09	9h30	17h30

Cette insertion entraîne une incohérence dans la base de données (la boutique ouvre à 17h30 et ferme à 9h30 le même jour) en plus lors de l'exécution, aucune erreur ne sera signalée puisque les deux attributs ont le même type. De plus, cela limite inutilement la liberté de mise en œuvre, cela pourrait avoir une incidence dans les représentations verticales.

Selon C. J. Date l'ordonnancement des attributs dans SQL est une violation du modèle relationnel puisqu'une relation est un ensemble. Par exemple, pour obtenir des données et les insérer dans une table, SQL utilise SELECT / FETCH INTO: v1,: v2, ... et INSERT INTO, le mappage des colonnes source vers les colonnes cibles est par numéro de colonne. Donc définir l'ordre signifie qu'il faut le définir pour chaque opérateur de la requête, y compris pour les opérations UNION et JOIN qui deviennent non commutatives, dans ce cas il faut mémoriser l'ordre des opérandes et cela rend la maintenance des bases de données très complexe. L'approche correcte consiste à associer les colonnes aux variables par leur nom, non par ordre.

Prenons par exemple deux tables :

- T1 avec les attributs At1, At2 ;
- T2 avec les attributs At3, At4.

Les deux insertions suivantes sont différentes :

```
INSERT INTO T3 (At1, At2, At3, At4)
SELECT *
FROM T1 JOIN T2 ON T1.At1 = T2.At3
```

```
INSERT INTO T3 (At1, At2, At3, At4)
SELECT *
FROM T2 JOIN T1 ON T1.At1 = T2.At3
```

Le résultat est différent vu que T1 JOIN T2 est différente de T2 JOIN T1, plus particulièrement dans l'ordre des attributs.

1.3.4 L'ordre des tuples dans les relations

Une relation est définie comme un ensemble de tuples. Mathématiquement, les éléments d'un ensemble n'ont pas d'ordre donc, les tuples dans une relation n'ont pas un ordre particulier. Toutefois, dans un fichier, les dossiers sont physiquement stockés sur le disque (ou en mémoire), donc il y a toujours un ordre entre les enregistrements. De même, lorsque nous présentons un rapport sous forme de tableau, les lignes sont affichées dans un certain ordre et celui-ci peut être non seulement pratique (pour le repérage), mais signifiant (priorisation).

Selon C. J. Date [L2] l'opérateur ORDER BY est un opérateur qui ne fait pas partie de l'algèbre relationnelle, parce qu'il produit des résultats qui ne correspondent pas à la définition du modèle relationnel. Cela pourrait engendrer de la confusion chez l'utilisation,

c'est-à-dire que la même table est représentée de plusieurs façons notamment selon les critères de tri associés.

Par contre, J. D. Ullman [L7] propose une extension de l'algèbre relationnelle pour traiter les collections ordonnées. L'extension est naturelle pour les opérateurs de sélection et de projection qui préservent l'ordre « naturellement ». Par contre, l'ordre des tuples n'est pas préservé pour les autres opérateurs tels que la jointure ou l'union.

L'insertion pose toutefois un problème particulier puisqu'il n'est pas possible, en général, de déduire la place du tuple dans la table. Le principal avantage de cette propriété est que les lignes d'une relation peuvent être récupérées dans un ordre différent.

Pour conclure, l'ordre des tuples ne fait pas partie de la définition d'une relation, car une relation est une représentation au niveau logique ou abstrait. Par conséquent, de nombreux ordres logiques peuvent être spécifiés sur une relation. Mais un ordre particulier peut être spécifié sur les tuples d'une relation lors de récupération des résultats. Cet ordre peut être déterminé par un ordre de tri ou d'insertion.

1.3.5 Les contraintes d'intégrité

En SQL, on considère qu'une contrainte est satisfaite si elle n'est pas fausse, c'est donc qu'elle peut être vraie... ou inconnue. Par contre, dans une clause WHERE, une expression booléenne de valeur inconnue est assimilée à faux, il en découle donc de nombreuses incohérences, car des formulations équivalentes en logique classique ou en algèbre relationnelle ne le sont pas en SQL (voir Annexe D). L'impact de l'annulabilité sur les contraintes d'intégrité est également important, il est analysé à la section 1.5 Modélisation de l'annulabilité.

Dans le modèle relationnel, par définition dans une relation R, l'ensemble des attributs de R possède la propriété d'unicité, c'est-à-dire qu'il n'existe pas de doublons. Par contre, du point de vue du modèle SQL, la nécessité de l'existence d'une clé relève nécessairement de la normalisation. C'est-à-dire que les tables en SQL peuvent avoir des doublons.

Comme dans le modèle relationnel, une clé étrangère en SQL est un ensemble d'attributs d'une relation dont les valeurs doivent s'unifier avec les valeurs d'une clé candidate.

Une contrainte de domaine SQL représente une forme généralisée des contraintes d'attributs et elle s'applique à chaque colonne définie sur le domaine en question. SQL permet aussi de définir une contrainte de domaine sous forme d'expression booléenne. Par exemple, les valeurs associées à un domaine peuvent dépendre des valeurs apparaissant dans l'état courant d'une certaine TABLE T [L1].

1.4 Système de typage

Nous nous proposons maintenant d'étudier les systèmes de typage sous deux aspects : leurs caractéristiques principales et leur modèle sous-jacent. Nous traiterons d'abord des caractéristiques principales suivantes :

- rigueur : fort ou faible,
- évaluation : statique ou dynamique,
- définition : explicite ou implicite.

Nous analyserons ensuite le polymorphisme en utilisant la théorie de la hiérarchisation des systèmes de typage fondés sur leurs opérations caractéristiques proposée par L. Cardelli et P. Wegner [A16]. Nous ne traiterons pas des autres aspects de cette hiérarchisation, déjà analysée par S. Medini [L15] dans le cadre son essai réalisé au sein du groupe de recherche Μῆτις.

1.4.1 Typage fort et typage faible

Les définitions associées à ces termes varient d'un auteur à l'autre [A16]. Dans le cadre de notre mémoire, nous définissons un système de typage fort comme étant un système qui garantit l'intégrité des données en ne permettant d'affecter à une variable que l'une des valeurs de son domaine définition et l'utilisation des seules opérations définies sur les domaines de ses arguments. Un système de typage est faible s'il n'offre pas une telle garantie.

En général, on atteint l'objectif du typage fort en associant (effectivement ou implicitement) l'identification du domaine à toutes les valeurs. La conversion implicite est utilisée dans la majorité des langages faiblement typés alors que la conversion explicite est utilisée par la majorité des langages fortement typés.

1.4.2 Typage statique et typage dynamique

Dans un langage statiquement typé, le code source passe par une étape de vérification de typage. Le grand avantage de typage statique c'est qu'il permet de détecter les erreurs avant l'exécution. Il présente aussi un avantage au niveau de la performance, en fait une fois la vérification de typage terminée, il permet au compilateur de se débarrasser des informations de typages accumulées, contrairement au typage dynamique qui conserve les informations sur le type de chaque variable tout au long de l'exécution.

1.4.3 Typage explicite et typage implicite

Un système de typage explicite exige la déclaration de type des données, la déclaration de types est exprimée explicitement par l'utilisateur contrairement au typage implicite, le compilateur détermine le type des données par le contexte (comme en ML [L17], par exemple). On peut demander explicitement une conversion d'un opérande dans un type désiré.

Les conversions implicites peuvent avoir lieu dans le calcul d'une expression quand on passe directement un argument à une fonction ou lors du retour d'une valeur par une fonction. Ces conversions implicites facilitent la tâche du programmeur. Elles peuvent cependant avoir un impact sur le typage explicite. Par exemple, si une valeur virgule flottante est convertie vers une valeur entière, la partie fractionnaire est incertaine. La valeur de `int x = 1.9`, est-elle 1 ou 2? Dans un contexte de typage explicite, il est donc préférable d'éviter des conversions implicites au profit des conversions explicites dont on peut documenter et contrôler l'effet. Le mécanisme de conversion doit aussi être examiné en regard du polymorphisme, nous y reviendrons donc à la section 1.4.4 Polymorphisme.

L'exigence du typage et de la conversion explicite permet donc de diagnostiquer des erreurs qui autrement seraient passées inaperçues (lorsque le programmeur croit se souvenir du type d'une variable, mais qu'il se trompe).

1.4.4 Polymorphisme

En général, le polymorphisme est un concept très important dans la majorité des systèmes de typage. Selon la classification de L. Cardelli et P. Wegner [A16], il existe deux principales catégories de polymorphisme : polymorphisme universel et polymorphisme ad hoc.

Polymorphisme ad hoc

On distingue deux sous-types du polymorphisme ad hoc :

- Surcharge : La surcharge permet de désigner plusieurs fonctions à l'aide d'un même nom dans la mesure où elles peuvent être distinguées sur la base de leur signature.
- Conversion : mécanisme de mise en correspondance d'une valeur d'un type avec une valeur d'un autre type ; la conversion peut-être totale ou partielle, implicite ou explicite, statique ou dynamique.

Polymorphisme universel

Les deux types suivants sont qualifiés d'universels parce qu'ils sont construits à l'aide de règles et d'opérateurs généraux applicables à tous les types.

- Polymorphisme paramétrique (paramétrisation) : Le polymorphisme paramétrique permet d'appliquer des opérations à des arguments d'un type passé en paramètre (le plus souvent contraint par dérivation sur un type de base représentant les propriétés communes attendues de tout type effectivement passé en paramètre). Par exemple, en Java les types `List` ou `Vector` sont des types génériques ou paramétrés, les opérations de tri, d'ajout ou de suppression peuvent être appliquées indépendamment du type de l'élément de la structure. *Parametric polymorphism is obtained when a function works uniformly on a range of types [...] [being] achieved by type parameters. (Cardelli-Wegner, 1985)* [A16]

- Polymorphisme d'inclusion (spécialisation) : Dans le polymorphisme d'inclusion les valeurs ont plusieurs types c'est-à-dire on peut les utiliser comme des paramètres à des opérations demandant un de ces types. Généralement, la notion du polymorphisme d'inclusion est associée à celle de sous-typage, toutes les valeurs du sous-type sont aussi des valeurs du super-type. *Subtyping : [values] of a subtype can be uniformly manipulated as if belonging to their supertypes. (Cardelli-Wegner, 1985)* [A16]

1.4.5 Mécanismes syntaxiques

Afin de faciliter la comparaison des systèmes de typage, nous présentons auparavant deux mécanismes syntaxiques fréquemment utilisés dans la mise en œuvre du polymorphisme : l'héritage et l'interface.

Héritage

Le système doit permettre de définir une hiérarchie de classes ou de types. Il s'agit certainement là d'un des concepts centraux de la technologie objet. L'héritage est un mécanisme syntaxique permettant d'exprimer le polymorphisme d'inclusion (sous-typage), une classe qui hérite d'une autre classe en devient une spécialisation. Cela signifie en général que la structure de la classe contient plus d'informations ou des informations plus spécifiques, et que certaines opérations associées en tirent parti. Nous distinguons deux types :

- héritage simple,
- héritage multiple.

Plusieurs langages de programmation orientés objets, tels que Eiffel et C++, incorporent l'héritage multiple qui permet à une classe fille d'hériter des propriétés et des fonctionnalités de plus qu'une classe mère, contrairement à l'héritage simple qui ne permet à une classe d'hériter que d'une seule classe mère.

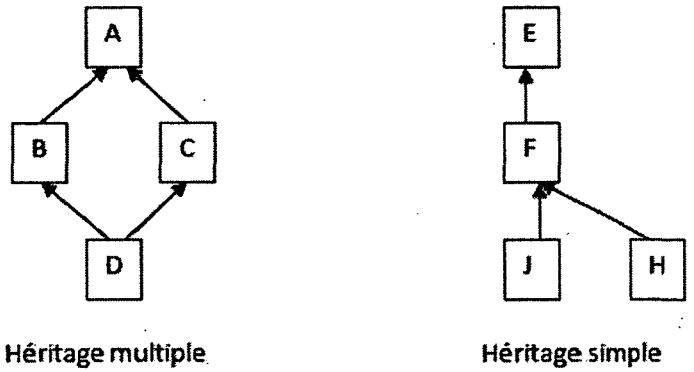

Figure 1-3 : Héritage multiple et simple

L'héritage multiple peut poser certains problèmes au niveau des classes filles, prenant l'exemple ci-dessus la classe D hérite des deux classes B et C, de même les deux classes héritent de la classe A; le problème que pose l'héritage multiple provient des propriétés qui sont présentes plusieurs fois dans les deux classes B et C, attendu qu'elles héritent d'une seule classe A.

Eiffel [A16] implémente de l'héritage multiple, l'approche d'Eiffel intègre des mécanismes qui permettent de gérer les conflits entre les propriétés héritées des parents. Ces mécanismes offrent la possibilité de renommer, de sélectionner, de redéfinir, ou d'indiquer qu'une propriété ne peut pas être redéfinie.

Prenons l'exemple ci-dessus ou x et y sont des propriétés de la classe A, la solution en Eiffel et la suivante :

```
class D inherit
B rename x as x1, y as y1 end
C rename x as x2, y as y2 end
feature...
```

Dans cet exemple, B et C ont des caractéristiques nommées x et y. La classe D ne serait pas valide sans renommer les propriétés x et y. La possibilité de renommer sert également à fournir des noms plus significatifs pour les descendants. Eiffel permet aussi d'autres adaptations telles que la redéfinition des propriétés pour les rendre plus spécifiques à la classe

fille, interdire la redéfinition de certaines propriétés des classes mères par les classes filles ou sélectionner seulement les propriétés qui peuvent être héritées.

La solution est générale, traite tous les cas, mais semble lourde à certains. De plus, la solution d'Eiffel garantit statiquement que tout conflit issu de l'héritage multiple est traité adéquatement dans la définition d'une classe au moment de la compilation, sans impact au moment de l'exécution.

On notera finalement qu'il est toujours possible de se limiter à l'héritage simple au sein de l'héritage multiple alors qu'on ne peut simuler l'héritage multiple par l'héritage simple. L'héritage multiple est strictement « plus puissant ».

Prenons l'exemple ci-dessous :

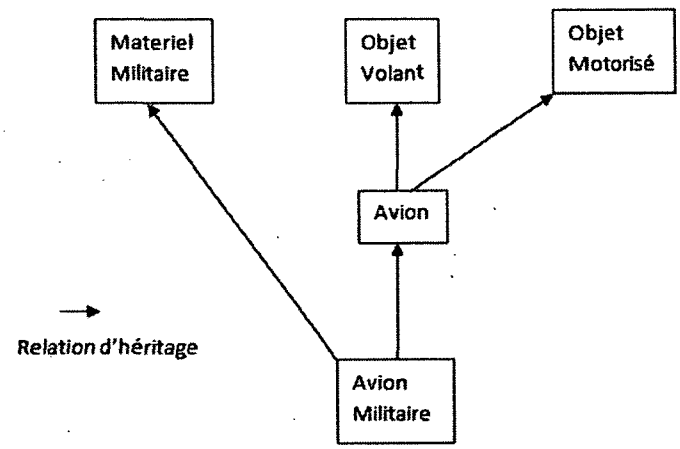

Figure 1-4 : Exemple de l'héritage multiple

Un avion militaire a les caractéristiques des avions (en général), mais aussi les caractéristiques du matériel militaire, dans ce sens la classe Avion_militaire hérite de la classe Avion et de la classe Matériel_militaire, de même pour un Avion il a les caractéristiques d'un Objet_volant et d'un Objet_motorisé. L'utilisation des interfaces et très lourde, les classes Avion et Avion_militaire ne peuvent pas hériter de deux classes et il faut réécrire l'ensemble des caractéristiques des deux autres classes. Finalement, l'interface ne

permet pas de vérifier la compatibilité des assertions entre les classes héritées, un inconvénient majeur en regard de la vérification, de la validation et de la fiabilité.

Interface

L'héritage est un mécanisme puissant, mais considéré complexe à manipuler par certains. Son grand avantage consiste dans le fait qu'il permet de réutiliser des structures qui existent déjà tout en s'assurant de respecter leur définition par la prise en compte automatisée des assertions (antécédents, conséquents et invariants). Ce type de réutilisation peut souvent être atteint grâce à un mécanisme plus simple, l'interface. Celle-ci procède par factorisation limitée à un seul niveau, ce qui le soustrait à la nécessité de définir les assertions de dérivation (ou de spécialisation). Supposons les deux classes D et C de la figure 3, l'implémentation de la classe D peut être faite soit en utilisant la factorisation avec la classe C ou soit en utilisant l'héritage. La factorisation permet d'utiliser les propriétés de l'objet C. Par contre, avec l'héritage, un objet de D « devient » un objet de C (polymorphisme d'inclusion) et, si on ne peut pas restreint l'accès aux propriétés de C lors de la définition de D, il devient possible d'utiliser n'importe quelles propriétés de la classe C sur un objet de la classe D.

La notion de l'héritage est une relation de partage de toutes les propriétés d'une classe. En l'absence de propriétés communes et d'assertions, il peut être avantageux d'utiliser la notion d'interface.

Java introduit le mécanisme d'interface pour décrire un type avec l'ensemble des fonctions que les objets de ce type doivent faire. Une interface est une classe abstraite sans attribut sans code. Elle représente un canevas syntaxique, sans plus. L'utilisation du mécanisme d'interface couplé à celui de l'héritage simple a été retenue en Java. La flexibilité des interfaces permet au système de typage de représenter des situations plus complexes que l'héritage simple et tout évitant la complexité de l'héritage multiple lorsque ce dernier n'est utilisé que pour offrir une abstraction syntaxique différenciée à une classe.

Supposons que nous ayons besoin de décrire un ensemble de fonctions sur des objets dont la propriété principale est d'être un moyen de transport. Considérons une voiture, un train, un avion comme des moyens de transport sans pour autant les hiérarchiser en classe les uns avec

les autres. Nous avons simplement besoin d'un type (moyen de transport) muni de propriétés, que chaque moyen de transport aura à charge de mettre oeuvre indépendamment. La mise en commun se limite donc au partage d'une syntaxe commune et non d'une sémantique commune (ou du moins sans que cette dernière puisse être vérifiée et validée).

Lorsqu'une interface est utilisée pour mettre en oeuvre une classe, on peut définir des variables de type interface, par exemple :

```
Public interface I{ ...}
I i; // i est une référence à un objet d'une classe implémentant l'interface I
```

Toutefois, on ne pourra pas affecter à i une variable de type I parce qu'on ne peut pas instancier une interface. Par contre, on pourra affecter à i n'importe quelle variable référencée à un objet d'une classe implémentant l'interface I par exemple :

```
Public class A implements I{ ... }
I i = new A(...);
```

De cette façon, on pourra manipuler des objets de toutes les classes implémentant I, non nécessairement liées par héritage. Dans ce sens, l'interface est un mécanisme syntaxique d'abstraction et de renommage. À ce titre, plusieurs considèrent l'interface comme un mécanisme de polymorphisme ad hoc, voire, selon certains auteurs, comme un simple mécanisme syntaxique.

1.4.6 Étude comparative

Pour conclure, nous avons choisi cinq langages pour illustrer les variations possibles sur les thématiques présentées précédemment :

- Java : un langage très répandu utilisant les mécanismes d'héritage simple et d'interface.
- Ada95 : le premier langage orienté objet normalisé et largement dans l'industrie.

- Eiffel : un langage normalisé qui implémente des mécanismes efficaces pour la gestion de l'héritage multiple.
- SQL[6] : le langage relationnel le plus utilisé; nous prenons en compte les versions 1992 (sans extensions objet) et 2003 (avec extensions objet) de la norme ISO 9075.
- Tutorial D : Tutoriel D est un langage qui implémente un ensemble d'exigences spécifiques définies dans le troisième manifeste par C. J. Date et H. Darwen.

Java

Java est un langage de programmation orienté objet développé par Sun Microsystems en 1995. L'objectif principal du langage Java était de faciliter le développement d'applications fonctionnant sur des plateformes différentes. L'utilisation d'une application Java nécessite l'installation préalable d'un environnement applicatif appelé JRE (Java Runtime Environement). La portabilité du langage Java résulte du fait que le code source est compilé pour produire un pseudo-code (ou byte-code) interprétable par la machine virtuelle Java.

Java est un langage fortement typé et possède deux sortes de type de données, les types primitifs et les types référencés représentés par les classes; ces types sont gérés par référence (pointeur). La gestion de la mémoire est assurée par le compilateur pour réduire de manière importante les erreurs et éviter toute utilisation nuisible de la mémoire.

---

[6] Depuis la norme SQL:1999, SQL comprend le sous-langage algorithmique PSM (Persistent Stored Module) appelé PL/SQL sous Oracle.

Tableau 1-3 : Système de typage du langage Java

<table>
<tr><td>Surcharge</td><td>Java permet la surcharge des méthodes on donnant le même nom a plusieurs méthodes avec des paramètres différents ce qui permet d'écrire des méthodes sémantiquement similaires sur des arguments différents en type, par exemple :
<pre>void calcul()
{ // instructions... }
void calcul(int valeur, int autre_valeur)
{ // instructions... }
void calcul(float valeur, int autre_valeur)
{ // instructions... }</pre>La surcharge est utilisée aussi dans la définition des opérateurs dont l'utilisation est différente selon le type des arguments passés. Par exemple, l'opérateur + peut être surchargé et utiliser différemment selon les opérandes : addition pour les entiers ou concaténation pour les chaînes de caractères.</td></tr>
<tr><td>Conversion</td><td>Java permet dans certains cas la conversion implicite et automatique par le compilateur.
<pre>Int x;
X=2.487; // après l'affectation, x contiendra 2.</pre>Java permet également la conversion explicite forcée (cast) :
<pre>Int x;
X=(int) 2.487 ;</pre></td></tr>
</table>

Polymorphisme paramétrique	Java offre une forme limitée de polymorphisme paramétrique. Cela permet de définir des types génériques ou paramétrés, toutes les opérations définies peuvent être appliquées sur n'importe quel type, par exemple : Public Class Collection <T> {... Public Boolean Exists(T aElements); Public void AddElement(T aElements); Public void RemoveElement(T aElements); }
Polymorphisme d'inclusion	Java offre l'héritage simple, sans mécanisme automatisé de gestion des assertions[7].

Ada95

Ada 95 est la deuxième forme normalisée du langage Ada originellement défini par une équipe française du groupe CII-Honeywell-Bull dans le cadre d'un concours organisé par le ministère de la Défense des États-Unis d'Amérique. Le langage offre la majorité des concepts orientés objet (l'héritage, le polymorphisme, la généricité...).

Ada95 est basé sur un système de typage fort et statique.

Tableau 1-4 : Système de typage du langage Ada95

Surcharge	Ada offre des mécanismes de surcharge pour les fonctions, les procédures et les opérateurs, par exemple :
	initialiser("Victor",200.0,"34567",courant);
	initialiser("Pierre",0.0,"78231",0.037,remunere);

[7] Java ne prend pas automatiquement en compte les règles d'héritage, par exemple un antécédent ne peut être qu'affaibli, un conséquent et un invariant renforcé.

<table>
<tr><td>Conversion</td><td>Ada se caractérise par la quasi-absence de conversions implicites, par exemple :
<pre>type Signed_Byte is range -128 .. +127;
U : Unsigned_Byte := 150;
S := Convert_Byte(U);</pre>La méthode de conversion (Convert_Byte) est définie ainsi :
<pre>function Convert_Byte is
 new Unchecked_Conversion(Signed_Byte, Unsigned_Byte);
 function Convert_Byte is
 new Unchecked_Conversion(Unsigned_Byte, Signed_Byte);</pre></td></tr>
<tr><td>Polymorphisme paramétrique</td><td>Ada permet la définition de méthodes génériques et de modules (packages) génériques, mais ne permet pas la définition de classes génériques, par exemple :
<pre>procedure print(C : in Compte'Class) is
begin put(C);new_line; end print;</pre>La procédure print est une méthode polymorphe (le type réel du paramètre est inconnu). Un type Compte'Class est l'union de tous les types dérivés à partir de Compte y compris Compte.</td></tr>
<tr><td>Polymorphisme d'inclusion</td><td>Ada offre un mécanisme analogue à l'héritage simple appelé dérivation de type. Pour être dérivé, un type doit être marqué spécialement lors de sa définition (tagged), par exemple :
<pre>type Compte is tagged
record
titulaire:Unbounded_String;
numero:String(1..5);
solde:Float;
end record;
type CompteRemunere is new Compte with
record
taux:Float;
end record;</pre></td></tr>
</table>

SQL-1992

SQL-1992 est la version « classique » de SQL avant les tentatives d'extension de différents concepts orientés objet.

Tableau 1-5 : Système de typage du langage SQL-1992

Surcharge	SQL permet la surcharge des fonctions et des procédures, par exemple :
	`CREATE FUNCTION test(int, real) RETURNS ...`
	`CREATE FUNCTION test(smallint, double precision) RETURNS ...`

Conversion	SQL permet la conversion implicite de certains types primitifs il permet aussi la conversion explicite. Ainsi, la conversion implicite de SMALLINT vers INT ou de REAL vers FLOAT, par exemple :
	`SELECT titre`
	`FROM projet`
	`WHERE CAST(ecrivain AS editeur_t) = editeur`

Polymorphisme paramétrique	SQL n'offre pas de mécanisme de polymorphisme paramétrique (généricité).
Polymorphisme d'inclusion	SQL ne supporte pas le polymorphisme d'inclusion.

SQL-2003

SQL-2003 comporte un ensemble d'extensions plus ou moins cohérentes en vue d'incorporer les principaux concepts orientés objet sur la base de SQL-1992. Nous n'avons pas pris en compte la version SQL-2008, car elle n'a pas encore fait l'objet d'une mise en oeuvre stable par les principaux éditeurs de SGBDR. SQL-2003 est donc caractérisé par :

- L'intégration complète de SQL-1992.
- Le support de types de données complexes.
- L'ajout de l'héritage.
- L'extension du concept de module (package).

Tableau 1-6 : Système de typage du langage SQL-2003

Surcharge	SQL permet la surcharge des fonctions et des procédures.
Conversion	SQL permet la conversion implicite de certains types primitifs il permet aussi la conversion explicite.
Polymorphisme paramétrique	SQL n'offre pas de mécanisme de polymorphisme paramétrique (généricité).

Polymorphisme d'inclusion	SQL:1999 et les versions ultérieures définissent une fonctionnalité d'héritage de type, par exemple :
	`CREATE TYPE Employe_t AS OBJECT (`
	`codemp CHAR(6), etat_civil etat_civil_t,`
	`adresse adresse_t, salaire NUMBER(6,2))`
	`NOT FINAL`
	
	`CREATE TYPE Pilote_t AS OBJECT UNDER Employe_t(`
	`nbHvol NUMBER, compagnie VARCHAR(6)))`
	`NOT FINAL`

Eiffel

Eiffel a originellement été conçu en 1986 par B. Meyer. C'est un langage orienté objet qui intègre la programmation par contrat et l'héritage multiple dans un cadre rigoureux.

Tableau 1-7 : Système de typage du langage Eiffel

Surcharge	Le mécanisme de surcharge est limité aux seuls opérateurs.

Conversion	Eiffel fournit un mécanisme pour permettre les conversions entre les différents types. Le mécanisme coexiste avec l'héritage et le complète. Pour éviter toute confusion entre les deux mécanismes, Eiffel ne permet la conversion qu'entre types non conformes (sans liens d'héritage), par exemple :
	`expanded class DOUBLE inherit ... create`
	`from_integer·convert {INTEGER},`
	`from_real convert {REAL},`
	`feature - - Initialization`
	`...`
	`end - - class DOUBLE`
	La clause Convert permet de spécifier la conversion explicite du type

	DOUBLE.

Polymorphisme paramétrique	Eiffel offre une forme limitée de généricité en permettant la définition d'une classe en fonction d'autres classes passées en paramètre, par exemple :
	`class ARRAY[G]`
	`item(i: INTEGER) : G is`
	`do`
	`...`
	`end`

Polymorphisme d'inclusion	Eiffel offre un mécanisme complet et rigoureux d'héritage multiple.
	`class`
	`B`
	`Inherit`
	`A`
	`rename`
	`g as f -- g was effective in A`
	`undefined`
	`f`
	`redefine h end`
	`feature -- Element change`
	`h ( ... ) is`
	`Do`
	`... New implementation (see below) ...`
	`End`
	`... Other features ...`
	`end -- class B`

Tutorial D

Tutorial D est langage relationnel, conçu par C. J. Date et H. Darwen pour illustrer les principes mis de l'avant par le troisième manifeste. Selon les auteurs, le langage corrige les mauvais choix historiques de SQL.

Le langage Tutorial D est un langage explicite fortement typé avec un contrôle de type statique. Il en existe une implémentation sous le nom de Rel [S12].

Tableau 1-8 : Système de typage du langage Tutorial D

Surcharge	Tutorial D supporte la surcharge par contre Rel ne l'implémente pas, par exemple : HIGHER_OF(A INTEGER, B INTEGER) HIGHER_OF(A RATIONAL, B RATIONAL)
Conversion	Tutorial D ne permet pas conversion implicite, mais il permet la conversion explicite grâce aux POSSREP et à l'opérateur CAST, par exemple : CAST_AS_CHAR(7) Autre exemple : Soit la définition suivante : TYPE CITY POSSREP (CHAR) Pour définir une contrainte sur le type CITY avec une comparaison avec une chaîne de caractères, il faut écrire : CONSTRAINT THE_CITY (CITY) = 'London' Pour que la comparaison soit valide.
Polymorphisme paramétrique	Tutorial D n'offre pas de mécanisme de polymorphisme paramétrique.
Polymorphisme d'inclusion	Tutorial D offre le mécanisme du polymorphisme d'inclusion grâce l'héritage simple (multiple) La déclaration suivante de Date et Darwen : « we believe that if inheritance is supported at all it must be multiple » [L10]. Montre que Tutorial D prend en charge l'héritage simple et multiple.

1.4.7 Synthèse

Tous les langages étudiés ont opté essentiellement pour le typage fort et explicite. Sur les autres aspects toutefois, ils se distinguent. Le tableau suivant présente la synthèse de l'étude comparative des six langages :

Tableau 1-9 : Aspects distinctifs (Java, Ada95, SQL-1992, SQL-2003, Tutorial D, Eiffel)

	Java	**Ada95**	**SQL-1992**	**SQL-2003**	**Tutorial D**	**Eiffel**
Typage statique	le plus souvent	oui	oui	oui	oui	oui

Surcharge	oui	oui	non	non	oui	partielle (opérateurs)
Conversion	limitée	aucune	implicite	implicite	explicite	explicite
Polymorphisme d'inclusion	héritage simple	dérivation de type	aucun	héritage simple	héritage simple	héritage multiple
Polymorphisme paramétrique	généricité [type]	généricité [type, valeur]	aucun	aucun	aucun	généricité [type]
Interface	oui	oui (la notion de paquetage)	non	non	non	non

Il nous est apparu intéressant de suivre l'approche de C. J. Date et H. Darwen pour faire la synthèse des différentes approches. Leur proposition s'inscrit dans un effort de définir une nouvelle génération de SGBD intégrant les approches relationnelles et objet. Comme eux, nous rejetons a priori toute proposition ignorant le modèle relationnel tout en affirmant la nécessité d'intégrer le modèle objet afin de pouvoir manipuler les données complexes. Les rubriques suivantes présenteront donc une synthèse de chacun des aspects distinctifs sur cette base.

Le langage SQL-1992 n'intègre pas la notion d'objet, contrairement à SQL-2003. La notion d'objet dans SQL-2003 est un peu limitée par rapport à d'autres langages. En effet, SQL-2003 n'intègre pas la notion d'encapsulation de données pour protéger l'accès aux informations contenues dans un objet. De plus, SQL-2003 n'intègre que l'héritage simple qui semble insuffisant pour la programmation orientée objet.

Typage statique

Nous pouvons constater que la majorité des langages possèdent un système de typage statique, contrairement à Java qui n'est pas complètement statique, une partie du typage est faite dynamiquement afin de donner une souplesse au langage.

Surcharge

Seuls Java et Ada95 et Tutorial D supportent la surcharge, l'avantage présenté par la surcharge d'une procédure réside dans la flexibilité de l'appel. Par contre, un système de typage qui ne supporte pas la surcharge nous amène à définir nos opérations de telle sorte

qu'elles n'acceptent que des valeurs des types compatibles (grâce aux mécanismes d'héritage et de généricité) et avoir une seule signature. Cette façon de faire n'est pas vraiment très pratique, prenant l'exemple de l'opération d'addition, pour chaque type il faut définir une opération différente des autres soit au niveau de la signature ou les types des paramètres.

En Java ou en C++, la surcharge est toujours la création d'une nouvelle méthode. En Eiffel, la surcharge est interdite pour éviter toute confusion entre les méthodes.

Conversion

Nous remarquons que seules les deux versions SQL qui permettent la conversion implicite, mais SQL ne peut pas effectuer la conversion implicite lorsque les opérandes contiennent des types de données incompatibles. En général dans Java, il n'est pas permis de convertir un type en un autre. Java prend en charge la conversion implicite d'un type numérique de petite taille vers un type de plus grande taille. Par exemple, le résultat les opérations entre entiers est du type long si l'un des opérandes est un long, sinon le résultat sera du type int. Dans le cas d'une conversion d'un type grand vers un type faible, si le risque de perdre des chiffres significatifs est grand, Java refusera la conversion. Il faudra mettre en place une conversion explicite.

Polymorphisme d'inclusion

Nous constatons que la majorité des langages supporte au moins l'héritage simple afin de rendre le système de typage évolutif et extensible, mais cela semble insuffisant. B. Meyer a intégré l'héritage multiple dans Eiffel malgré sa complexité, mais en offrant des mécanismes rigoureux pour gérer les conflits et la complexité introduite par le mécanisme.

Polymorphisme paramétrique

Nous pouvons remarquer que seuls les langages des bases de données ne supportent pas la généricité (sans pourtant que cette absence ait été justifiée). Le principal avantage de la généricité est l'abstraction d'un ensemble de concepts cohérents pour construire des algorithmes sur ces concepts indépendamment de leur implémentation.

Interface

Nous pouvons remarquer que Java et Ada sont les seuls langages qui proposent le mécanisme d'interface qui permet au système de typage de représenter des situations plus complexes que l'héritage simple et tout évitant la complexité de l'héritage multiple, en effet les interfaces ne permettent pas de partager du code, mais uniquement des spécifications de méthodes et il semble verbeux[8] malgré sa simplicité. Cela rend les applications et les classes moins évolutives que celles spécifiées à l'aide de l'héritage multiple puisque les modifications aux ancêtres doivent être prises en compte dans chacune des implantations des interfaces. En fait, l'héritage multiple, comme en Eiffel, est très utile pour augmenter la réutilisation des classes en permettant de factoriser des parties de code à insérer plusieurs fois ou de propager automatiquement les modifications faites sur les parties factorisées. Cette fonctionnalité est très importante pour l'évolution et l'extension des applications, de plus il permet de propager automatiquement les modifications faites dans les parties factorisées. Même si la combinaison entre l'interface et l'héritage simple nous permet en partie ces avantages, l'héritage multiple offre des économies de développement en temps, en argent et en mémoire ainsi qu'un code plus compact.

1.5 Modélisation de l'annulabilité

L'annulabilité des attributs a été introduite pour prendre en compte une situation fréquente lors du passage du monde réel ou monde virtuel modélisé par une base de données : l'indisponibilité d'une donnée. Cette indisponibilité peut avoir plusieurs causes, un comité ANSI/SPARC en a recensé 14 (voir Annexe A).

Lors de la modélisation, les attributs annulables pourront réduire considérablement le nombre des tables, mais l'introduction du concept d'annulabilité a entraîné beaucoup de problèmes dont les suivants :

---

8 Il existe trois usages de l'interface : un mécanisme syntaxique de compilation séparée (comme en Modula-2), un mécanisme de spécification « late binding » et un mécanisme pour simuler l'héritage multiple. Nous ne nous intéressons ici qu'au troisième aspect celui de palliatif à l'héritage multiple.

- des incohérences logiques en regard de la logique booléenne bivaluée, induisant la nécessité d'utiliser une logique multivaluée non standard;
- des exigences particulières en regard des jointures, induisant la définition d'opérateurs particuliers;
- un traitement spécial des fonctions d'agrégation et de regroupement qui ne peut échapper à certains paradoxes.
- La définition des différents opérateurs, comportements et règles requis semble dépendre de la cause de l'annulabilité.

Il en découle une complexification du modèle relationnel et l'introduction d'incohérences encore considérées inéluctables jusqu'à présent [L10, L9].

La modélisation de l'annulabilité a autre impact, encore plus fondamental : l'utilisation d'une logique multivaluée (trois valeurs ou plus). La logique multivaluée est incompatible avec la propriété du tiers exclu par ailleurs essentielle à l'axiome du monde fermé sur lequel repose le modèle relationnel. Cette propriété permet de garantir que toute assertion qui n'est pas vraie est fausse et réciproquement. Sans cette propriété, il n'est plus possible de garantir que la connaissance du monde stockée dans une base de données est cohérente, rendant ainsi caducs les avantages découlant de l'imposition des contraintes liées à la fermeture du modèle [A14, A23, A24, A25].

Après avoir présenté les stratégies principales de modélisation de l'annulabilité, nous examinerons les propositions de Codd, Zongmin, Vassiliou et Date.

1.5.1 NULL : valeur ou propriété?

La modélisation de l'absence d'une valeur se divise en trois approches : attributs annulables, extension des types et l'approche SQL.

- Attributs annulables : cette approche est basée sur l'inclusion de la propriété d'annulabilité aux attributs. En effet, les attributs des relations peuvent interdire les valeurs absentes ou non.

- Extension des types : Dans cette approche, les types primitifs peuvent être étendus à prendre une valeur nulle et par conséquent avoir une valeur nulle pour chaque type. Formellement, il faut définir les types comme des treillis continus, selon le modèle établi par Scott et Strachey [L14] dans le cadre de la sémantique dénotationnelle. Un langage utilisant cette approche nécessite l'inclusion d'opérateurs et de fonctions spéciales pour pouvoir manipuler les extensions de types et les valeurs nulles. La proposition de Vassiliou [A7] que nous présentons à la prochaine section en est un exemple.
- L'approche SQL : SQL utilise un marqueur spécial NULL pour représenter l'absence d'une valeur. Celui-ci peut avoir plusieurs interprétations (telles que « inapplicable » ou « inconnu »). En SQL, le concept de NULL est une source de confusions dans l'utilisation du langage SQL, NULL n'est pas une valeur dans la plupart des contextes (il est différent d'une chaîne de caractères vide ou du zéro, par exemple), mais il est considéré comme une valeur lors des calculs relationnels. Lors de la définition d'une valeur de colonne dans une table, le mot NULL est utilisé pour identifier l'annulabilité de l'attribut sans permettre pour autant d'en indiquer la raison. Dans ce cas, il s'agit d'une propriété de l'attribut. Lors de la définition d'une table, nous pouvons associer à un attribut la propriété NOT NULL pour spécifier que l'attribut ne peut pas avoir des valeurs absentes. Par ailleurs, il n'est pas possible de spécifier qu'un opérande dans une comparaison vaut NULL par exemple WHERE X = NULL est une expression illégale, la forme correcte est WHERE X IS NULL. Cela signifie que NULL est une propriété de l'attribut. Par contre, dans les opérations de classement, tous les NULL sont considérés comme des valeurs égales entre elles et (selon le type) soit supérieures ou inférieures à toutes les valeurs non nulles. Par ailleurs, dans d'autres contextes, les NULL provenant de types distincts peuvent être (ou non) considérés distincts aux fins de la génération des doublons.

Pour plus de détails sur les incohérences découlant de l'utilisation du NULL en SQL, voir l'annexe B.

1.5.2 Synthèse des propositions principales

La présente section a pour objectif de faire une synthèse des principales propositions amenées pour traiter le problème des informations absentes en vue de la définition d'un nouveau langage de manipulation de bases de données relationnelles. Les propositions suivantes :

- Codd I et Codd II [L9, A5];
- Zongmin [A10];
- Vassiliou [A7];
- C. J. Date et H. Darwen [A11, S3];
- Dominus-1.

sont présentées à l'annexe E.

Nous allons comparer les différentes propositions par rapport à la logique utilisée, la complexité induite (algorithmique et relationnelle) et les adaptations requises aux modèles (relationnel et de typage).

Les points communs :

- Aucune des propositions ne modélise la cause (ou l'origine) de l'annulabilité.
- Toutes les propositions prennent en compte les classes d'annulabilité A (information absente) et B (information inapplicable).
- Aucune des propositions ne traite les classes d'annulabilité C (information inaccessible) et D (information verrouillée).

La comparaison présentée au tableau suivant repose sur trois critères :

- LV : le type de système logique.
- Modélisation : l'approche à la modélisation de l'annulabilité.
- Complexité : l'évaluation de la complexité algorithmique induite par l'approche.

Tableau 1-10 : Comparaison des propositions de modélisation des valeurs absentes

	LV	Modélisation	Complexité
Codd I	3V	Dénotation hybride de l'absence d'information (NULL).	Extension limitée aux opérateurs relationnels et aux expressions logiques.

Codd II	4V	Dénotation hybride de l'absence d'information : A-value (information absente) et I-value (valeur inapplicable).	De même que la logique 3V, la logique 4V nécessite une extension limitée aux opérateurs relationnels et aux expressions logiques. La logique 4V est cependant plus complexe que la logique 3V.
SQL	3V	Dénotation hybride de l'absence d'information (NULL).	Extension (incohérente en pratique) des opérateurs relationnels, d'agrégation et de groupement. Impact sur les expressions logiques et le langage algorithmique.
DD	2V	Modélisation relationnelle par stratification et récursion, sans annulabilité explicite.	Fragmentation des relations, augmentation du nombre de jointures.
Vassiliou	2V	Extension des types avec une valeur nulle.	Inclusion de fonctions et d'opérateurs spéciaux (tant au niveau relationnel qu'algorithmique) pour manipuler les extensions de types avec valeur nulle.
Ma Zongmin	5V	Modélisation de trois types d'absence d'information : I (non applicable), M1 (applicable, mais non disponible) et M2 (on ne sait pas si la valeur est applicable et on ne sait pas sa valeur).	Tableaux de vérité immenses et complexes (25 cas, voir annexe A). De plus, la logique 5V pose les mêmes problèmes que la logique 3V.
Dominus-I	4V	Modélisation de deux types d'absence d'information : ND (non disponible ou absente) et NV (non valide ou inapplicable).	L'impact est limité au côté relationnel, les valeurs ND et NV provoquant des exceptions au niveau algorithmique.

Conclusion

Nous pouvons constater que le traitement de l'information absente dans le modèle relationnel a été largement étudié par les chercheurs. L'annexe E présente les principales approches pour gérer ce problème. L'approche de Vassiliou [A7], propose d'associer l'absence d'une valeur au type des données, mais cette approche est rejetée à cause notamment de sa complexité malgré qu'elle permette un traitement formel irréprochable dans le cadre de la sémantique

dénotationnelle. Les autres approches intégrant l'annulabilité (Codd I, Codd II, SQL, Ma Zongmin, Dominus-1) souffrent toutes d'une complexité pratique considérable tout en n'apportant pas de solution complète au problème de l'annulabilité. La solution DD fait le pari que l'annulabilité peut être ignorée dans un contexte de modélisation adéquate.

Pour conclure, le problème des informations absentes est un problème que l'on rencontre très fréquemment, il est donc nécessaire de pouvoir les gérer dans les nouveaux systèmes de gestion de base de données. Discipulus devra donc intégrer une proposition à cet égard, même si aucune des propositions recensées ne semble satisfaisante. C'est un des aspects importants que nous traiterons aux chapitres 2 et 3.

Chapitre 2

Approche retenue

La définition d'un nouveau modèle de SGBDR doit traiter plusieurs aspects importants, dont ceux-ci :

- le modèle du système;
- le modèle transactionnel;
- le modèle relationnel;
- le modèle de typage;
- le langage algorithmique;
- le langage relationnel;
- le langage transactionnel.

Dans le présent chapitre, bien que notre mémoire ne couvre pas le modèle du système lui-même ni l'aspect transactionnel, nous décrirons d'abord le cadre du système transactionnel dans lequel s'inscrit notre proposition. Nous présenterons ensuite notre approche sous les angles du modèle relationnel, du modèle de typage, des principes sous-jacents à la définition du langage algorithmique (incluant le langage de définition des types) et à ceux du langage relationnel.

2.1 Système transactionnel

Tout SGBD doit communiquer avec d'autres systèmes qu'ils soient des SGBD ou non. Nous présentons dans cette section la structure du système lui-même, le modèle d'interconnexion sous la forme de services et les principales catégories de messages pouvant être échangés.

2.1.1 Le système

L'architecture du modèle DOMINUS se constitue de plusieurs éléments :

- Le système de typage : le système représente l'ensemble des types primitifs ainsi que les nouveaux types définis par les utilisateurs grâce aux mécanismes du langage algorithmique.
- Le langage algorithmique : le langage intègre les mécanismes et les instructions de base permettant de traiter des problèmes du monde réel.
- Le langage relationnel : le langage permettant de définir et manipuler les données.
- Le modèle d'interconnexion : le modèle englobe les composants permettant de communiquer avec d'autres systèmes à travers des structures bien définies.

Ci-dessous l'architecture du modèle SGBD DOMINUS :

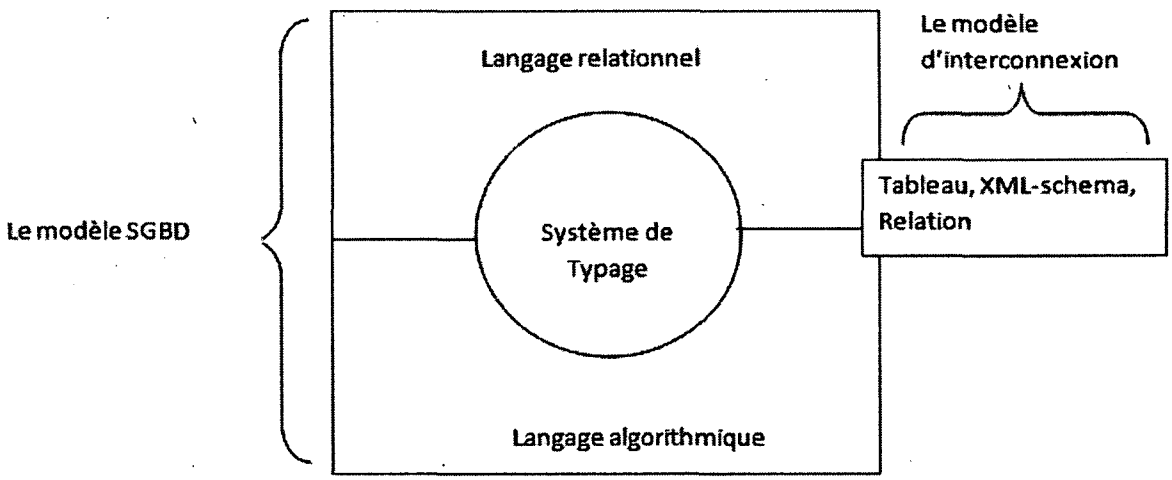

Figure 2-1 : Architecture du modèle de SGBD DOMINUS

Le système de typage représente le noyau du modèle et constitue l'élément commun entre le langage relationnel et le langage algorithmique, c'est-à-dire que les deux langages se basent sur le même système de typage, ensuite le modèle possède aussi un système d'interconnexion permettant de communiquer avec les autres systèmes, les protocoles vont être basés sur un des concepts suivants : la relation, la liste, le tableau ou l'arborescence.

2.1.2 Les services

L'interaction entre le système et l'extérieur passe par des services. Tout appel de service peut être accompagné de données conformes aux structures décrites à la section 2.1.3 Les . Il existe deux types de service : les commandes et les requêtes. Une commande modifie la base de données. Une requête retourne un objet construit à partir des données stockées dans la base de données. Un service est une suite séquentielle de transactions. Chaque transaction maintient les propriétés ACID (Atomicité, Cohérence, Isolation et Durabilité) [L14].

- Atomicité : une transaction est une unité atomique de traitement. Soit toutes les modifications de données sont effectuées, soit aucune ne l'est.
- Consistance : la fin d'une transaction doit laisser la base de données dans un état cohérent afin de conserver l'intégrité de toutes les données.
- Isolation : une transaction ne laisse pas voir ses changements aux autres transactions tant qu'ils ne sont pas confirmés.
- Durabilité : Lorsqu'une transaction est terminée, ses effets sur le système sont permanents. Les modifications ne peuvent être perdues si une défaillance survient ultérieurement.

Une transaction est elle-même composée d'opérations relationnelles.

2.1.3 Les messages

Les messages d'intérêt dans le cadre de notre mémoire sont les appels de service. Un appel de service comprend généralement des données représentées sous la forme d'objets sérialisés. Lorsque le service est une requête, il retourne en outre un objet, lui aussi sérialisé. Quatre types d'objets sont envisagés pour le moment : la relation, la liste (correspondant à une collection ordonnée, donc avec la possibilité de doublons), le tableau (semblable à la liste, mais avec une fonction d'indexation) et l'arborescence.

La relation

La relation est l'objet de prédilection pour la communication entre deux SGBDR Discipulus. Il est le seul à permettre le maintien de la pleine intégrité (relationnelle, fonctionnelle et de domaine).

La liste

La liste est une collection de données ordonnée permettant d'avoir des doublons. C'est l'outil d'échange le plus simple. Typiquement, il devrait être mis en œuvre par un flux (*stream*).

Le tableau

Un tableau est une représentation de relation à la laquelle est adjointe la propriété d'ordonnancement des tuples et des attributs. Pour des raisons de compatibilité avec d'autres SGBDR, les doublons sont aussi permis, ainsi que les attributs annulés.

L'arborescence

L'arborescence permet le maintien de l'intégrité de domaine (complète ou partielle), possibilité d'intégrité référentielle (complète ou partielle), par exemple XML-Schema [S13]. Cette présentation permet de structurer et d'organiser les données ainsi qu'elle facilite l'intégration dans d'autres bases de données même si l'intégrité des données est partielle.

2.2 Modèle relationnel

Depuis sa proposition par E. F. Codd en 1970, le modèle relationnel a été la base autant de la recherche théorique que du développement pratique des SGBD. Il a fait ses preuves et a montré sa puissance et sa fiabilité. Ses qualités reposent notamment sur :

- une théorie rigoureuse et des principes simples;
- une technologie mature, fiable et performante;
- l'indépendance des modèles de données de leur représentation physique avec la possibilité d'optimiser cette dernière, le plus souvent automatiquement.

Il est toutefois remarquable qu'en raison des circonstances historiques, ses nombreuses mises en œuvre commerciales n'aient encore jamais respecté plusieurs éléments du modèle

théorique. Par ailleurs, le langage SQL par sa syntaxe hétérogène et irrégulière par son ampleur[9] et ses contradictions internes présente une complexité aussi considérable qu'inutile et de nombreuses incohérences relativement au modèle relationnel. Pour ces raisons, il nous a semblé important de revenir aux principes théoriques du modèle, adoptant ainsi la voie tracée par Tutorial D :

- relations sans doublons;
- tuples non ordonnés;
- attributs non ordonnés.

Reste les questions de l'annulabilité des attributs et de l'algèbre relationnelle de référence abordées ci-après.

2.2.1 Annulabilité des attributs

Selon E. F. Codd [L9,A5], l'utilisation des NULL causerait des conflits de compréhension et de signification en raison notamment :

- du passage d'une logique standard à une logique multivaluée;
- de son impact sur les jointures;
- du traitement spécial requis par les fonctions d'agrégation et de regroupement.

Après avoir envisagé, une solution reposant sur la catégorisation des cas d'annulabilité et une logique quadrivaluée (voir annexe E, proposition Dominus-1), le maintien d'un système logique standard nous est apparu déterminant dans notre recherche de fiabilité et de validité démontrables. Que faire alors de l'annulabilité? La proposition de C. J. Date et H. Darwen est la seule à maintenir la logique standard. Elle serait toutefois coûteuse en pratique [L10, A23, A15] et introduit une complexité importante lors de certaines manipulations des données.

---

[9] En 2004, la norme ISO 9075:2003, comptait plus de 3800 pages avec seulement 9 des 14 volumes publiés. La publication complète de la norme complète n'est d'ailleurs toujours pas terminée alors que deux versions (2008 et 2011) lui ont succédé sans pour autant être plus complètes. Les copies « de travail » des volumes manquants totalisent déjà plusieurs milliers de pages.

Après avoir analysé au chapitre précédent les différentes propositions de la modélisation de l'absence d'une valeur, nous avons conclu à la nécessité d'un nouveau modèle qui permet de gérer l'absence d'une valeur d'une façon simple et cohérente ainsi que le rejet de SQL vu ces incohérences.

La proposition Dominus-2 se base sur un ensemble de règles servant à guider le choix de la solution finale. En premier lieu, l'indétermination est un fait qui ne peut être occulté, cependant, les informations dont on peut disposer sont souvent incomplètes, imprécises ou émanent de sources hétérogènes, avec une qualification et une fiabilité variables. Ceci ne doit évidemment pas être un obstacle à leur gestion par des systèmes d'informations avancés. En deuxième lieu, l'indétermination ne doit pas être un substitut à la mauvaise modélisation. En effet, l'expérimentation de l'approche de C. J. Date et H. Darwen a montré que de nombreuses situations où la valeur est absente peuvent être traitées au niveau de la modélisation. Finalement, en troisième lieu, le système logique doit être maintenu le plus simple possible de façon à faciliter le raisonnement, la vérification et la validation des programmes.

En conséquence, nous proposons de maintenir le concept d'annulabilité, mais en le traitant comme une exception, de façon analogue à une division par zéro. Autrement dit, un attribut peut être nul tant et aussi longtemps qu'aucune opération relationnelle ne le met en cause. Lorsque cela arrive, une exception est levée et peut être traitée comme telle. Ceci nous semble correspondre au fondement même d'un monde fermé, les données absentes pouvant apparaître de façon transitoire, lors du passage de monde réel ouvert au modèle virtuel fermé.

Le mécanisme d'exception lui-même provient du langage algorithmique et ne nécessite aucune extension ni au modèle relationnel ni au langage relationnel. Différents opérateurs sont prévus afin d'aider à localiser les attributs nuls et à leur associer des marqueurs d'annulabilité distincts en fonction des causes de l'annulabilité, ce qui ouvre le modèle des causes d'annulabilité plutôt que de le contraindre à un modèle particulier (voir Annexe I – pour la classe définissant les opérations liées aux exceptions de Discipulus).

Finalement, tout ceci peut être réalisé dans le cadre de la logique standard bivaluée.

Les marqueurs pour les valeurs manquantes

Un des problèmes de la modélisation de l'absence d'une valeur consiste dans le nombre des situations pouvant mener à l'absence d'une valeur et à la nécessité d'appliquer des traitements différenciés en fonction de celles-ci. La proposition Dominus-2 intègre un marqueur pour les attributs annulables afin de spécifier la raison de l'absence. L'utilisation d'un marqueur permet de garder une trace sur la cause de l'absence de la valeur et, par conséquent, offre la possibilité de définir une stratégie précise de gestion de l'absence de cette valeur. Si, à un moment donné, la cause de l'absence est résolue, l'utilisateur peut la modifier.

Par exemple, soit une relation `Employé` comprenant les attributs : `Id_employé`, `nom` et `commission`, ce dernier étant annulable :

Tableau 2-1 : Relation Employé illustrant le marqueur d'annulabilité.

Id_employé	*Nom*	*commission*
e1	Jean	100
e2	Michel	**NULL (NA[« patron »])**
e3	Marc	200
e4	Fati	**NULL (ND[« renouvellement »])**

Par exemple, la commission de l'employé e2 est non applicable, car l'employé est en fait le patron. Par contre, la commission de l'employé e4 est applicable, mais sa valeur est non disponible, parce que l'employé est cours de renouvellement de contrat. La liste fermée des marqueurs légitimes est établie grâce aux primitives de la classe NullException.

2.2.2 Algèbre relationnelle de référence

L'algèbre relationnelle telle qu'elle a été définie par E. F. Codd comprend huit opérations sur les relations (la sélection, la projection, la jointure naturelle, la division, l'union, l'intersection, la différence et le produit cartésien) ainsi qu'un opérateur de renommage. Cet ensemble ne forme pas un noyau, par exemple la jointure est une projection d'une sélection d'un produit cartésien. En fait, la jointure, l'intersection et la division peuvent être définies en utilisant les cinq autres opérations. Par contre les autres opérations peuvent être considérées comme primitives.

C. J. Date propose une extension de l'algèbre d'origine de E. F. Codd appelée algèbre A [L10], formée d'opérateurs logiques de premier ordre (OR, AND, NOT) ainsi que d'autres opérations permettant de comparer deux relations ou de les étendre. Ces opérateurs ont été définis pour exprimer les différentes catégories de requêtes dans les bases de données objet-relationnelles. Par exemple l'algèbre A remplace l'opération de la différence de l'algèbre d'origine par les deux opérateurs logiques NOT et AND.

L'algèbre A* [A28] est inspirée de l'algèbre A. Elle comprend un opérateur de fermeture relationnelle (TCLOSE) et l'extension des opérateurs logiques en lien avec cet opérateur (NOT*, OR*, AND*). L'algèbre A* offre un moyen de spécifier les domaines comme fonctions et permet d'augmenter l'expressivité du modèle de données.

Dans un premier temps, nous nous en tiendrons à l'algèbre de E. F. Codd. Ceci facilitera la validation de notre approche et le développement du traducteur. Si notre première expérimentation est concluante, nous nous proposons d'examiner les extensions comprises dans A et A*.

2.3 Modèle de typage

Le système de typage est le lien entre le modèle relationnel et le modèle algorithmique. Limité à un ensemble fixe et non extensible de types dans la version originale de SQL, il a été développé de façon anarchique et incohérente au fil des différentes versions du langage. Le système de typage est donc au centre de notre proposition et un des principaux garants de la qualité de notre proposition.

Plutôt que de développer un système de typage propre comme l'on fait C. J. Date et H. Darwen, nous avons préféré utiliser un système existant, bien fondé théoriquement, axé sur la vérifiabilité, mais également éprouvé : le système de typage du langage Eiffel. Les prochaines sections montrent comment nous en sommes arrivés à cette proposition.

2.3.1 Typage fort, statique et explicite

Pour assurer un grand degré de sûreté, le système de typages du nouveau langage doit être un système fortement typé avec un contrôle de type statique. Il est vrai qu'on perd au niveau

d'expressivité, mais il nous permet de détecter toutes les erreurs liées aux types avant toute exécution ou évaluation du code produit par le compilateur et il nous permet aussi d'éviter toutes les opérations dangereuses qu'un système de typage faible pourrait causer. Le système doit par ailleurs être explicite afin de faciliter le raisonnement, la vérification et la validation des programmes.

Dans ce contexte, le polymorphisme doit être strictement encadré et vérifiable statiquement.

2.3.2 Surcharge

La surcharge est un polymorphisme « ad hoc » qui vise à diminuer la charge mnémonique induite par les identificateurs représentant un même concept sémantique, mais différenciés afin de conserver un typage strict. Plusieurs langages l'adoptent, comme C++, Java et Ada, d'autres, comme Eiffel, le limitent aux opérateurs.

Supposons deux fonctions pseudoaléatoires, la première produit un nombre entier simple et la deuxième, un nombre flottant double.

Première approche :

```
int GetRandomInteger( int min, int max );
double GetRandomDouble( double min, double max );
```

Deuxième approche :

```
int GetRandomNumber( int min, int max );
double GetRandomNumber( double min, double max );
```

L'avantage de la deuxième approche est que l'utilisateur ne s'inquiète pas de laquelle des fonctions il faut appeler. Il suffit d'appeler GetRandomNumber avec des paramètres de type entier ou double et obtenir un résultat. Rien ne garantit que les deux versions de la fonction fassent la même chose, la surcharge peut survenir « par accident » plutôt que « par conception ». La première approche est plus explicite dans les noms, elle ne laisse aucune ambiguïté quant à la fonction qui est appelée (et donc quant à son sens).

Bien qu'Eiffel et Tutorial D autorisent certaines formes de surcharge, Discipulus n'en offrira aucune.

Eiffel

Eiffel ne permet pas la surcharge. Chaque nom de fonction est unique au sein de la classe. Ce choix de conception assure la lisibilité des classes, en évitant des ambiguïtés à propos de la sémantique de la fonction lorsque celle-ci sera invoquée par un appel. Selon B. Meyer, il simplifie également la mise en œuvre des mécanismes OO, en particulier, l'héritage multiple [A17]. Les mêmes noms peuvent, bien sûr, être réutilisés dans différentes classes, ce qui ne crée pas d'ambiguïté puisque les noms sont toujours qualifiés. Par contre, Eiffel permet la surcharge dans le cas des opérateurs. Ceci est nécessaire à partir du moment où il est permis de définir, et de redéfinir, des opérateurs. En fait, nous jugeons que la surcharge des opérateurs n'est pas toujours intuitive et risque de causer plus de confusion que de gain en lisibilité.

Tutorial D

Tutorial D [L16] offre un mécanisme de surcharge pour les fonctions et les opérateurs, mais il exige que la modification de la sémantique soit illégale notamment parce qu'elle représente une source d'ambiguïté. Par contre, aucun moyen ne peut être mis en œuvre par le compilateur pour s'en assurer.

Discipulus

Discipulus ne propose aucune forme de surcharge. Tout d'abord parce que nous jugeons que cela entraîne plus d'ambiguïté et de confusion pour les utilisateurs. De plus, nous nous basons sur le modèle d'Eiffel qui utilise la qualification complète des identificateurs et, dans ce cas, la portée de l'utilisation de la surcharge aurait été limitée à l'intérieur d'une classe, un même identificateur pouvant toujours être défini sans ambiguïté dans deux classes différentes. Donc le bénéfice de mettre en place le mécanisme est donc considérablement réduit.

D'autre part, la surcharge d'opérateurs a souvent été critiquée parce qu'elle permet aux programmeurs de donner aux opérateurs une sémantique totalement différente selon les types de leurs opérandes. Une autre question, plus grave, concerne les opérateurs : certaines règles de mathématiques sont violées. Par exemple, les propriétés mathématiques usuelles (commutativité, associativité, distributivité, inversibilité) ne sont pas toujours maintenues

d'une définition à l'autre; par exemple, lorsque « + » est utilisé pour la somme d'entiers et la concaténation de chaînes de caractères.

2.3.3 Conversion

Les conversions sont des fonctions essentielles de tout type abstrait. Il est donc naturel qu'on les retrouve dans les classes. La question est de savoir si elles méritent un mécanisme de définition qui leur soit propre. Pour répondre à cette question, il convient d'abord d'examiner la question du caractère explicite de la conversion. Ce caractère explicite (ou implicite) doit être considéré pour la définition et pour l'utilisation.

Notre étude du chapitre précédent montre les dangers à laisser implicite la définition des conversions, a fortiori dans un contexte où un haut standard de vérification et de validation est exigé. Cela induit-il que l'utilisation d'une conversion doit être explicite elle aussi? Examinons les propositions d'Eiffel et de Tutorial D à cet égard.

Eiffel

La nécessité de définir un mécanisme spécifique de définition de fonction de conversion découle de ces décisions : pour les rendre implicites, les fonctions de conversion doivent être explicitement définies comme telles. De plus, pour maintenir l'intégrité du système d'héritage multiple, une fonction de conversion ne peut être définie entre deux classes parentes (sans quoi il y a ambiguïté quant à l'utilisation du mécanisme à privilégier : conversion ou dérivation). En effet, selon B. Meyer [L13] l'affectation polymorphe contrôlée par héritage est un rattachement simple de référence qui n'implique pas un changement des valeurs référencées, en revanche, les conversions transforment une valeur d'un type dans une valeur d'un autre type. Les mécanismes sont donc distincts et disjoints. En conséquence, le langage doit être garant de cette distinction.

Tutorial D

Tutorial D permet la conversion de type explicite et ne prend pas en charge la conversion de type implicite. En fait, le mécanisme POSSREP permet la définition de plusieurs représentations pour une même classe et impose la définition explicite des fonctions de

conversion. Contrairement à Eiffel qui fournit un mécanisme pour permettre les conversions entre les différents types à l'intérieur du mécanisme d'héritage et de définition de classe, Tutorial D fait coexister le mécanisme POSSREP avec l'héritage et en assure l'indépendance. En particulier, tout langage conforme au Troisième manifeste doit fournir le mécanisme POSSREP alors que l'Héritage n'est qu'une « forte suggestion ».

Discipulus

Discipulus adopte l'approche Eiffel, la conversion est explicite par spécification. Le mécanisme a été soigneusement conçu pour s'assurer que tout est clair, sans ambiguïté, et efficace, la conception applique le principe suivant : *"A type may not both conform and convert to another". (Bertrand Meyer,2001)* [L13]

2.3.4 Polymorphisme d'inclusion

Le polymorphisme est un concept extrêmement puissant, l'approche OO propose d'organiser les types d'objet (appelés classes) par une hiérarchie d'héritage. Les types héritent des propriétés de leurs super-types.

Le polymorphisme d'inclusion est la seule technique solide dans tous les systèmes de typage permettant la définition de nouveaux types par extension d'anciens types. Nous concluons de notre étude comparative du chapitre précédent que l'héritage simple est insuffisant. Si l'interface semble préférable par sa simplicité conceptuelle, l'héritage multiple permet une meilleure modélisation, une meilleure réutilisation et réduit la redondance des interfaces dans les situations complexes. En définitive, l'héritage multiple semble mieux adapté à l'évolution à long terme des classes, une caractéristique importante dans le contexte des SGBDR puisque la pérennité des bases de données est généralement plus longue que celles des applications. L'héritage multiple modélise mieux également les généralisations, en particulier lorsque deux bases de données doivent être fusionnées.

L'argument principal pour l'adoption de l'héritage multiple est qu'il permet des économies d'effort remarquables (que ce soit pour l'analyse, la conception, la mise en œuvre ou l'évolution). Cependant, il y a des problèmes qui se posent lors de son utilisation. La

puissance même de l'héritage multiple demande des moyens adéquats pour le contrôle. En particulier, la question de la vérification statique et celle de la gestion des conflits de nom entre les caractéristiques héritées des parents différents sont cruciales. Eiffel présente une réponse rigoureuse, efficace et éprouvée à ces questions. Il nous est donc apparu prudent de ne pas innover et d'adopter la solution d'Eiffel en bloc. Cette solution passe par la prise en compte des assertions (invariants, antécédents et conséquents) en accord avec le modèle des types abstraits et les principes de la programmation par contrat. Finalement, ce dernier aspect est cohérent avec notre approche voulant faciliter le raisonnement, la vérification et la validation des programmes.

2.3.5 Polymorphisme paramétrique

Le polymorphisme paramétrique ou généricité est un mécanisme qui permet de réduire considérablement la complexité des programmes et d'en augmenter le niveau d'abstraction. Le polymorphisme paramétrique en Ada repose sur une instanciation statique et explicite de chacune des classes paramétrées. Cette exigence simplifie grandement la compilation des unités de programme. En Eiffel, l'instanciation des paramètres est implicite (automatique) grâce à l'inférence de types qui est un mécanisme qui permet de rechercher automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source. Il en découle une diminution appréciable de la quantité de lignes de code requises pour mettre en place les instances génériques au détriment d'une complexité plus grande de l'environnement de compilation et d'édition des liens. Remarquons toutefois que cette complexité n'a aucune incidence négative sur l'efficacité lors de l'exécution du code engendré par Eiffel.

Pour ces raisons, et pour conserver l'unité du système de typage déjà largement inspiré d'Eiffel, nous avons choisi de maintenir dans Discipulus la solution avancée par Eiffel. Il est vrai que l'approche adoptée par Ada est plus générale, c'est-à-dire que les paramètres génériques peuvent être des valeurs, des types et des sous-programmes, mais, pour le moment, ne nous jugeons pas nécessaires de telles fonctionnalités, une décision qui pourra être examinée à nouveau dans le futur.

2.3.6 Interface

Selon l'étude comparative du chapitre précédent, Java et Ada sont les seuls langages qui intègrent syntaxiquement la notion d'interface, cela permet de faire la séparation entre la spécification des objets et l'implantation du code, l'un des objectifs des interfaces [A33, S16] est de simuler la notion de l'héritage multiple et d'éviter la complexité introduite par ce mécanisme.

Cependant, l'héritage multiple et la possibilité de définir des classes abstraites ou différées rendent inutile la notion d'interface qui par ailleurs affaiblit considérablement les possibilités de vérification statique et dynamique du langage. De plus, Eiffel offre les moyens essentiels pour gérer la complexité du mécanisme.

2.3.7 Types de base

Le langage de notre modèle est un langage orienté objet, qui manipule des classes, plus précisément des objets, qui sont des instances de ces classes. Toutefois, il existe quelques types primitifs basés sur des classes, permettant de manipuler directement les données les plus courantes. Ces données sont notamment spécifiées par une représentation en mémoire. L'utilisateur peut étendre cet ensemble, soit en définissant de nouvelles classes, soit en spécialisant certaines classes.

Les types primitifs minimaux que nous devrions avoir doivent exprimer au moins : Booléen, Entiers, Flottants, Texte, Moment (estampille temporelle) [R2].

Nous souhaitons toutefois distinguer le type et la représentation physique de ses valeurs dans le système. Par exemple, le matricule d'un étudiant peut être représenté physiquement par une chaîne de caractère, mais cela ne signifie pas que les opérations applicables sur les chaînes de caractère vont être aussi applicables sur les matricules des étudiants. En fait, les opérateurs vont dépendre de la sémantique du type en question.

Nous examinerons maintenant la solution proposée par Eiffel, celle que nous aimerions implanter en Discipulus et celle que nous avons retenue temporairement aux fins de la première mise en œuvre expérimentale de Discipulus.

Eiffel

Les objets que l'on manipule lors de l'exécution d'un programme Eiffel sont soit des objets que l'on manipule par sémantique de pointeur, soit des objets que l'on manipule directement dans la mémoire. Pour les types de base, aucun autre mécanisme ne peut désigner directement l'objet en question, en particulier par un pointeur. Par exemple, d'une variable de type INTEGER, la seule façon de manipuler l'objet est de pouvoir utiliser la variable en question. Par conséquent, il n'y a pas de liaison dynamique pour un objet manipulé directement dans la mémoire, l'invocation d'une méthode correspond à un appel direct et efficace. Ainsi, la place mémoire pour un objet (EXPANDED) se limite très exactement à l'emplacement mémoire pour ranger ses différents attributs. Un groupe important de types (EXPANDED) basé sur des classes, comprend les types de base BOOLEAN, INTEGER, REAL, DOUBLE, CHARACTER et STRING. De toute évidence, la valeur d'une entité déclarée de type INTEGER doit être un entier, pas une référence à un objet contenant une valeur entière. Eiffel offre la possibilité de définir des classes selon les deux sémantiques, les classes avec sémantique en mémoire étant préfixées du mot réservé EXPANDED. Les bibliothèques standard utilisent d'ailleurs largement ce mécanisme, ce qui évite la définition de nombreuses classes dans le langage lui-même.

Discipulus

Nous désirons retenir l'approche d'Eiffel avec ses deux types de sémantique, la sémantique EXPANDED correspondant au concept de représentations en Tutorial D. Cet aspect est par ailleurs lié aux questions d'atomicité des types primitifs et de performance.

Solution temporaire

Lors de la mise en œuvre expérimentale, nous n'avons pas pu implanter cette proposition, en raison notamment des difficultés inhérentes à sa traduction en SQL. Nous avons donc adopté la solution suivante : faire correspondre les types primitifs de Discipulus à certains types primitifs choisis de SQL, sans possibilité de les modifier ou de les étendre au sein du langage. Ceci permet de prendre facilement en compte la non-normalisation de ces types au sein des différents dialectes SQL.

2.4 Langage algorithmique

Le langage algorithmique est un ensemble d'instructions élémentaires permettant de traiter les problèmes, cet ensemble est constitué essentiellement des instructions pour la déclaration des variables, de l'affectation, de l'évaluation d'expression, de l'appel de procédure, des instructions conditionnelles et des boucles. Les langages OO intègrent la notion de classe qui peut être définie comme une mise en œuvre d'un type de données abstrait. Cela signifie qu'il décrit un ensemble des objets, caractérisé par les opérations qui leur sont applicables, et par les propriétés formelles de ces opérations. Ces opérations peuvent être soit des méthodes appelées variables d'instance ou de routines appelées attributs. Les routines et les attributs d'une classe sont appelés ses caractéristiques. Les objets d'exécution sont appelés les instances directes de la classe. Ces éléments désormais classiques doivent trouver leur place dans le langage algorithmique, ce qui ne représente plus vraiment une problématique. Tout au plus, faut-il proposer une syntaxe uniforme et cohérente.

Par contre, notre recherche des meilleurs moyens permettant de faciliter le raisonnement, la vérification et la validation des programmes nous conduit à deux mécanismes supplémentaires bien intégrés en Eiffel : la programmation par contrat et la gestion des exceptions.

2.4.1 Programmation par contrat

La propriété d'un programme d'être correct ou non est une propriété essentielle. Plus précisément, on peut dire d'un programme qu'il est correct uniquement en faisant référence à la spécification qu'il doit satisfaire. La spécification d'un programme peut s'exprimer à l'aide des antécédents, des conséquents et des invariants; cela représente une forme de contrat. Un contrat entraîne nécessairement certaines obligations, mais rend le programme plus fiable et plus sûr d'où le besoin mécanisme de vérification et de validation de contrat.

En outre, un tel mécanisme contribue à rendre les programmes plus robustes et plus facilement réutilisables. Il pourrait même contribuer à réduire le temps consacré aux tests, en facilitant la détection les erreurs dans les premières phases du cycle de développement.

Finalement, ce mécanisme permet de garantir aussi la rigueur du mécanisme d'inclusion de types. En fait, les antécédents et les conséquents peuvent en effet être étendus ou renforcés. Si, par exemple, nous avons une classe B qui hérite de la classe A, le contrat appliqué sur les fonctions de B sont bine celle de A, mais avec la possibilité d'être renforcer. Pour les invariants ils doivent être respectés à l'entrée et à la sortie de chaque fonction, mais pas nécessairement pendant le temps de traitement, toutefois il est possible de définir de nouveau sur des membres n'existant pas dans A.

Eiffel encourage la programmation par contrat, les développeurs peuvent exprimer les propriétés formelles des classes par des affirmations d'écriture, qui peut notamment apparaître dans les rôles suivants :

- Antécédent : exprime les exigences que les clients doivent satisfaire chaque fois qu'ils appellent une routine.
- Conséquent : exprimer les conditions a vérifié sur le retour d'une routine, si la condition a été satisfaite à l'entrée.
- Invariant : l'invariant d'une classe doit être rempli par chaque instance de la classe après un appel à une routine exportée de la classe. L'invariant représente une contrainte de cohérence générale imposée sur toutes les routines de la classe.

2.4.2 Gestion d'exceptions

Tout programme est susceptible de comporter des erreurs. Le mécanisme de gestion des exceptions a été proposé en tenant compte de cette réalité. Une méthode qui rencontre une d'erreur lève une exception qui génère un objet de type Exception qui permet de traiter l'erreur. La mise en place de ce mécanisme est très importante puisqu'il permet d'éviter qu'une erreur ne se propage sans avoir été prise en compte.

Le mécanisme de gestion des exceptions d'Eiffel emploie le modèle de la reprise. Une routine peut gérer une exception par la clause (rescue). Cette clause optionnelle tente de récupérer en apportant l'objet courant à un état stable (une satisfaction de l'invariant de la classe). Il existe deux façons de le faire :

- La clause de (rescue) peut exécuter une instruction (retry), ce qui provoque le redémarrage de l'exécution de la routine depuis le début pour tenter à nouveau de remplir son contrat, généralement par une autre stratégie.
- Si la clause de (rescue) ne se termine pas avec (retry), puis la routine échoue, elle signale immédiatement une exception au point d'appel.

Tout programme en exécution peut être sujet à des erreurs pour lesquelles des stratégies de détection et de réparation sont possibles. Ces erreurs ne sont pas des bogues, mais des conditions particulières (ou conditions exceptionnelles, ou exceptions) dans le déroulement normal d'une partie d'un programme.

Dans le cas de notre modèle, l'utilisateur d'une valeur absente dans les calculs relationnels pourrait causer des problèmes, ou des incohérences dans les résultats. Cette situation est considérée comme une situation anormale du système et par conséquent nécessite un traitement spécial d'où l'introduction d'un système de gestion d'exception. Par exemple, la division par zéro est une situation anormale qui génère une exception, de même l'utilisation d'un marqueur NULL dans une évaluation donne une valeur inconnue qui devrait aussi générer une exception : tout d'abord pour permettre à l'utilisateur de savoir qu'une valeur manquante existe et, ensuite, qu'il y a eu tentative d'utilisation dans un calcul. En somme, il est proposé qu'il soit légitime d'annuler un attribut, mais pas de l'utiliser dans un calcul.

Le traitement des situations exceptionnelles nécessite de définir une stratégie pour qui permettra de résoudre le problème ou de revenir pour annuler une opération fautive, ou encore modifier les valeurs de certains attributs, puis reprendre l'exécution du programme un peu avant le site de l'erreur. Dans le cas d'une valeur absente, l'utilisateur pourrait, par exemple, substituer une valeur absente par une valeur appropriée ou encore faire propager l'indétermination. En cas de propagation, les utilisateurs savent explicitement que le traitement a rencontré une valeur absente. Nous verrons plus en détail les mécanismes associés à cette proposition au chapitre suivant.

2.5 Langage relationnel

La question du choix de la syntaxe du langage est primordiale et très importante, le langage doit être complet et concis. Prenons l'exemple du langage C++, il est difficile de trouver des utilisateurs qui connaissent la totalité du langage, c'est encore plus rare dans le cas de SQL. Par contre, les langages Algol, Pascal, Modula-2 et Modula-3 sont facilement mémorisables. Même un langage d'une portée aussi générale qu'Ada demeure accessible au professionnel de l'informatique.

Pour la conception du langage relationnel de Discipulus, nous sommes essentiellement confrontés à trois choix :

- Adopter une syntaxe proche de celle de SQL : cela pourrait faciliter la transition des programmeurs et des programmes vers Discipulus, mais cela engendre le problème de confusion avec SQL lorsque la syntaxe est proche, mais le sens est différent (même légèrement). Ce risque est très important : l'exemple du passage de C à C++ l'illustre et justifie de cette approche.
- Concevoir une syntaxe originale : dans cette approche, il faut faire toute une analyse et une étude approfondie afin de définir une syntaxe lisible compréhensible et concise. C'est l'approche retenue par les concepteurs de Tutorial D. Il en résulte une nécessaire période d'instabilité, pendant laquelle les changements aux langages sont nombreux, ce qui entraîne un retard dans l'adaptation des compilateurs. Le cas de Rel l'illustre malheureusement.
- Étendre la syntaxe d'Eiffel : cette approche semble la plus simple et la plus cohérente. Cela nous permet d'avoir une syntaxe éprouvée, cohérente avec celle du langage algorithmique. En fait, pour le langage relationnel nous avons une syntaxe plus orientée vers les instructions (comme SQL) que vers les expressions (comme Tutorial D). Pourquoi? Parce que c'est le choix fait en Eiffel et que nous ne voulons pas nous éloigner du style syntaxique d'Eiffel, par souci de cohérence et de simplicité

Nous présentons dans le chapitre 4 les éléments syntaxiques et sémantiques du langage relationnel.

2.6 Synthèse

Nous avons présenté l'architecture de notre modèle SGBD, en particulier le modèle d'interconnexion, pour ensuite justifier le maintien du modèle relationnel. Nous avons ensuite proposé un modèle original permettant le traitement des valeurs absentes d'une façon efficace et cohérente ne mettant pas en cause l'intégrité du modèle relationnel et préservant l'utilisation de la seule logique standard bivaluée.

Nous avons ensuite présenté un modèle de typage fort, statique, explicite et extensible construit sur les bases du modèle utilisé par le langage Eiffel. Ce modèle inclut le polymorphisme d'inclusion et le polymorphisme paramétrique. Une proposition consistant à différencier les représentations de base des types eux-mêmes est avancée, mais ne sera pas incorporée à la première version du traducteur du langage.

Nous avons ensuite motivé l'adoption du langage algorithmique d'Eiffel, incluant la programmation par contrat et le mécanisme de gestion des exceptions.

Finalement, nous avons motivé l'élaboration d'un langage relationnel proche de la syntaxe d'Eiffel par le souci de garder une homogénéité dans l'ensemble du langage.

Chapitre 3

Langage proposé

Le présent chapitre est une adaptation de la section 2 du rapport de recherche [R2] rédigé par l'auteur en collaboration avec C. Khnaisser et L. Lavoie.

Définir un nouveau langage est une tâche ardue, complexe et qui force l'humilité. D'emblée, nous avons voulu limiter le plus possible l'innovation autre que celle qui motivait notre projet, refusant ainsi d'expérimenter de nouvelles avenues relativement au système de typage, au modèle algorithmique ou à la syntaxe. Nous nous sommes concentrés sur l'expression du modèle relationnel, l'incarnation de notre proposition concernant l'annulabilité et le modèle transactionnel encapsulé par les services d'un système ouvert conforme au modèle OSI.

De façon générale, nous avons adopté le système de typage, modèle algorithmique et le modèle syntaxique (et le plus souvent la syntaxe concrète) d'Eiffel tel que décrit par la norme ISO 25436 [S14]. Nous avons modifié minimalement cette base en écartant ou en adaptant certains éléments lorsqu'ils s'intégraient mal à notre proposition.

Pour les autres éléments du langage, nous avons donc cherché à incorporer les éléments de notre proposition présentés au chapitre 2 en prolongeant au mieux le modèle syntaxique et sémantique d'Eiffel.

Le chapitre comporte cinq sections :

- la description de ce qu'est un serveur de bases de données Dominus;
- le rappel et la motivation des modifications apportées au langage Eiffel;
- la description des éléments principaux des langages relationnel et transactionnel;
- la gestion des bases de données et des accès;

- une synthèse.

Soulignons finalement que le présent chapitre est une rapide présentation partielle de Discupulus insérée dans le mémoire par souci d'en faciliter la lecture autonome. Il ne remplace pas la lecture du rapport de recherche [R2] définissant le langage.

3.1 Le serveur de bases de données Dominus

Un serveur de bases de données Dominus (SBDD) héberge des bases de données dont il offre les services aux clients qui interagissent avec lui par l'entremise de messages décrits à l'aide du langage Discipulus. Tout SBDD héberge minimalement une base de données appelée Catalogue qui décrit l'ensemble des bases de données hébergées par le SBDD.

Un SBDD est dépositaire au plus haut niveau de trois catégories d'entités, appelées `server_object` : les schémas, les utilisateurs et les groupes. Un schéma correspond à une base de données au sens du modèle Dominus. Un utilisateur représente un processus externe habilité à interagir avec le service, il comprend l'ensemble des informations (paramètres d'interaction) conditionnant les accès au cours d'une session. Un groupe est un ensemble d'utilisateurs auxquels des paramètres communs d'interaction peuvent être associés. Le concept de groupe n'est pas essentiel, mais constitue un moyen pratique d'uniformiser et de faciliter la gestion des paramètres d'interaction.

```
message :
     (notes)?
     'with' server_designator 'do'
     message_body
     (rescue)?
     (notes)?
     'end'
     ;
message_body :
     (server_instruction)*
     ;
server_instruction :
     simple_instruction
     | cloning_or_renaming_instruction
     | evolution_instruction
     | authorization_instruction
     ;
```

Le `message` décrit par `message_body` est transmis pour traitement au serveur désigné par `server_designator`. Le `message_body` est une suite d'instructions décrivant le traitement. Les `simple_instruction` ont pour but de créer, activer, désactiver et détruire des `server_object`, les `cloning_or_renaming_instruction` de les cloner et de les renommer, les `evolution_instruction` de les modifier et `authorization_instruction` de leur associer des règles d'accès. Les `server_object` sont dotés deux états « défini » et « indéfini ». Par ailleurs, un `server_object` défini peut être soit « actif » ou « inactif ». Après sa création, un objet doit être défini avant de pouvoir être activé. Seuls les objets actifs sont accessibles aux fins d'interaction et de traitement.

En cas d'échec du traitement prévu, la clause `rescue` décrit un traitement alternatif qui s'ensuit : soit constater l'échec du message soit modifier le contexte et amorcer la reprise du traitement prévu. Comme toutes les clauses `rescue` de Discipulus (à l'instar d'Eiffel), la clause `rescue` du `message_body` ne peut donc avoir pour effet que la réussite totale ou l'échec. Par ailleurs, le serveur doit maintenir les propriétés ACID, notamment en cas d'échec : les `server_object` doivent être dans l'état dans lequel ils auraient été si la tentative de traitement n'avait pas eu lieu.

Les `notes` sont des directives destinées aux logiciels qui traitent les programmes Discipulus et n'ont pas d'incidence sur le sens de ceux-ci.

3.2 Le langage algorithmique

La classe est l'unique moyen de structuration en Eiffel, elle est même la catégorie initiale de la définition du langage. Une classe définit des propriétés et un invariant. Les propriétés d'une classe sont dénommées features et désignent les attributs comme les opérateurs. Les opérateurs sont soit des créateurs (créant un objet), des commandes (modifiant l'objet) ou des fonctions (avec valeur de retour, mais sans effet de bord). Les convertisseurs sont des fonctions identifiées comme telles afin de désambiguïser les adaptations implicites (soit par dérivation soit par conversion). Fait à noter, la syntaxe d'une référence à un attribut ou à une fonction sans paramètre est identique en Eiffel, ce qui facilite l'évolution du code.

L'en-tête `class_header` permet de spécifier le nom et la nature de la classe : **`deferred`** lorsqu'elle est abstraite, du moins en partie (et doit donc nécessairement être complétée par héritage), **`frozen`** si elle ne peut être dérivée, **`expanded`** si elle n'utilise pas la sémantique par pointeurs (dans ce cas, elle est nécessairement **`frozen`** également).

```
class :
     (notes)?
     class_header
     (formal_generics)?
     (obsolete)?
     (inheritance)?
     (creators)?
     (converters)?
     (features)?
     (invariant)?
     (notes)?
     'end'
     ;
class_header :
     (header_mark)? 'class' class_name
     ;
header_mark :
     'deferred' | 'expanded' | 'frozen'
     ;
```

La section **`formal_generics`** est la déclaration des paramètres polymorphiques, la section **`obsolete`** permet d'indiquer l'obsolescence de certaines propriétés de classes (elles demeurent utilisables, mais induisent un avertissement lors de la compilation). Elle comprend ensuite la section **`inheritance`** définissant l'héritage avec les adaptations requises en regard de chacune des ancêtres (redéfinition, indéfinition, renommage, sélection).

Nous allons présenter maintenant les éléments que nous n'avons pas conservés ou traduits dans le langage algorithmique de Discipulus, ainsi que les problèmes rencontrés et les solutions envisageables pour les résoudre.

Dans la version actuelle du traducteur, nous n'avons pas pris en charge la traduction des éléments suivants d'Eiffel : l'héritage multiple, les types encapsulés (EXPANDED), les tuples, les agents, la définition et la surcharge d'opérateurs syntaxiques, certaines formes lexicales des chaînes de caractères et les interfaces aux langages tiers et aux bibliothèques externes.

L'héritage multiple

La mise en œuvre de l'héritage multiple nécessite une réflexion approfondie et un effort considérable dans un contexte de traduction vers SQL. Il nous est apparu préférable de nous concentrer dans un premier temps, et dans le cadre de ce mémoire, aux autres aspects plus novateurs de notre proposition.

L'état courant du traducteur n'offre donc pas ce mécanisme, mais nous réitérons son caractère essentiel. Conséquemment, sa mise en œuvre devrait être la prochaine étape dans l'évolution du traducteur Discipulus.

Les types encapsulés

Là aussi, le choix d'une traduction vers SQL rend la mise en œuvre difficile. Cette fonctionnalité n'est toutefois pas essentielle, si un choix approprié de types de base prédéfinis est inclus dans le langage. L'expérimentation permettra d'affiner le choix initial que nous avons fait et permettra vraisemblablement de reporter encore longtemps la mise en œuvre de la fonctionnalité.

Les tuples

Tout d'abord, il faut faire la distinction entre le concept du tuple dans le modèle algorithmique et celui dans le modèle relationnel, dans le modèle algorithmique un tuple est un type de conteneurs dont ces éléments peuvent avoir des types différents. Tandis que dans le modèle relationnel un tuple est un composant du corps d'une relation avec certaines propriétés relationnel. La question est de savoir si cette distinction pertinente, sinon il faut définir à nouveau ce concept tout en respectant les propriétés du modèle relationnel.

Rappelons également que l'ajout des tuples est relativement récent en Eiffel et ne constitue pas une fonctionnalité fondamentale. Nous avons donc privilégié de restreindre le concept des tuples au seul langage relationnel.

Les agents

La notion d'agent permet de décrire et de manipuler des opérations comme des données ordinaires, ces opérations peuvent être passées en paramètre comme des données et voir leur

exécution retardée. Bien qu'ils soient considérés comme des objets normaux, ils peuvent amener des problèmes spécifiques liés au système de typage. L'introduction récente et très contestée [R3] de cette fonctionnalité nous amène à ne pas l'inclure dans Discipulus pour le moment. Nous estimons que ce mécanisme nécessite une étude approfondie afin d'en confirmer la définition actuelle ou de proposer des adaptations. Il faut également prendre en considération que le mécanisme de transaction joue déjà le rôle des agents au niveau relationnel et que l'agent ferait en quelque sorte double emploi.

La définition d'opérateurs syntaxiques

La notation infixe issue des expressions mathématiques à l'aide de symboles particuliers (+, −, ×, ÷, etc.) est un accommodement syntaxique aux usages qui introduit redondance et complexité. S'il nous apparaît prématuré de la retirer pour les types de bases, nous avons cependant pris la décision de ne pas en permettre la définition, la redéfinition ni la surcharge en Discipulus.

Les dénotations de chaînes de caractères multilignes

La norme ISO permet la définition de plusieurs formes de chaînes de caractères multilignes. L'intérêt de cette fonctionnalité nous apparaît au mieux marginal et elle ne pouvait être commodément mise en œuvre avec JavaCC. Nous l'avons donc exclue de Discipulus, d'autant plus que les versions d'Eiffel antérieures à la norme ISO ne l'offraient pas.

L'intégration des langages tiers et des bibliothèques externes

Dans un premier temps, nous avons décidé de reporter l'étude de cette fonctionnalité accessoire aux fins de l'expérimentation quoiqu'essentielle dans un contexte d'application réel.

3.3 Le langage relationnel et les transactions

Dans cette section, nous présentons une syntaxe du langage relationnel inspirée de la syntaxe concrète d'Eiffel. Nous abordons le langage dans l'ordre suivant : la relation et le tuple, la transaction, la variable et l'expression relationnelle.

Nous avons retenu pour le moment l'algèbre relationnelle telle qu'elle a été définie par E. F. Codd, elle est comprend les opérateurs : Projection, Restriction, Jointure, Union, Intersection, Différence, Division, Produit. Les opérateurs de sélection et de groupement proviennent du calcul relationnel et constituent une forme commode cohérente avec l'esprit général de la syntaxe d'Eiffel.

3.3.1 La relation et le tuple

La relation est un des éléments clés de notre proposition. Discipulus, à l'instar de Tutorial D, distingue le concept de variable de celui de relation (la relation est en fait un générateur de variable d'un type particulier). Ci-après la syntaxe pour définir une relation :

```
relation :
     relation_header
     (formal_generics)?
     (obsolete)?
     (inheritance)?
     (creators)?
     (converters)?
     (attributes)?
     (invariant)?
     'end'
     ;
relation_header :
     relation_designator
     ;
attributes :
     (attibute_definition)+
     ;
attibute_definition :
     identifier ':' type
     ;
relation_assertion :
     key_assertion | fk_assertion | null_assertion
     ;
key_assertion :
     'key' attribute_selection
     ;
fk_assertion :
     'fk' attribute_selection
     'to' relation_name (('.' attribute_name) | attribute_name_list)?
     // si la deuxième attribute_selection est omise, reprendre la première
     ;
null_assertion :
     'nullable' attribute_name_list
     ;
attribute_selection :
     attribute_name | attribute_name_list
     ;
```

```
attribute_name_list :
     '(' attribute_name (',' attribute_name)* ')'
     ;
```

Les sections **formal_generics**, **obsolete** et **inheritance** sont identiques à leurs contre-parties algorithmiques. Les sections **creators** et **converters** sont analogues à leurs contre-parties algorithmiques à cette nuance près que l'objet retourné doit appartenir à un type de relation. La section **invariant** semblable à sa contre-partie algorithmique peut faire appel non seulement aux expressions booléennes algorithmiques, mais également aux contraintes propres aux relations et décrites sous **relation_assertion**. La section **attributes** est une restriction de la section algorithmique **features**.

Bien que retenues au sein de Discipulus, les parties suivantes ne sont présentement pas prises en charge par le traducteur : la forme générique, les créateurs, les convertisseurs, la propriété obsolète et l'héritage multiple.

Attibuts

Une relation est définie par une liste d'attributs, un attribut est un identificateur associé à un type, le nom de chaque attribut doit être au sein de (l'en-tête de) la relation. L'annulabilité est exprimée par une contrainte **null_assertion** de la section **invariant** de sorte qu'elle n'apparaît pas dans la liste des attributs.

Contraintes relationnelles

Pour assurer la cohérence et l'intégrité, chaque relation possède ses propres invariants. Un invariant est défini par une contrainte qui permet de restreindre les valeurs à l'aide d'expressions (calculables) du langage (en pratique des expressions booléennes auxquelles s'ajoutent les **relation_assertion**).

Nous distinguons trois types de contraintes relationnelles spécifiques :

- **key_assertion** : cette clause permet définir les clés candidates; si aucune clé candidate n'est définie, l'ensemble des attributs de la relation forme l'unique clé candidate de celle-ci (par définition de la relation). La présence d'attributs annulables introduit un cas particulier : un attribut annulable ne peut faire partie d'une clé

candidate, corollairement, en l'absence de clé candidate explicite, la clé candidate unique ne comprend que les attributs non annulables.

- `fk_assertion` : les attributs de cette clause doivent à ceux d'une clé candidate de la relation référencée.
- `null_assertion` : la liste des attributs annulables.

Tuples

Le tuple est défini de façon analogue (remplacement du mot réservé `relation` par le mot réservé `tuple`, limites des contraintes relationnelles à la seule `null_constraint`), type des objets retournés limités aux tuples. Sa mise en œuvre n'est pas encore disponible dans la version courante du traducteur.

3.3.2 La transaction

La transaction est l'unité de base décrivant un traitement. Le tuple et la relation forment l'essentiel du langage de description de données (LDD) et la transaction l'essentiel du langage de modification de données (LMD). Il n'y a pas de confirmation (COMMIT) explicite, elle est implicite au moment de la terminaison normale de la transaction. C'est dire qu'une transaction qui échoue ne peut que tenter une reprise (retry au sein de la clause rescue ou propager son échec en restaurant son état inital (ROLLBACK). Cette proposition peut sembler hardie, mais elle correspond à la théorie du contrat qui au coeur même d'Eiffel et de notre proposition visant à augmenter la fiabilité des SGBDR. Seule l'expérience pourra montrer son expressivité en pratique. Après tout, certains (et pas des moindres) s'étaient bien opposés à l'élimination des GOTO préconisée par Dijkstra et Hoare.

L'emboîtement des transactions suit le principe de précaution. Toute transaction comprise dans une autre ne peut être confirmée que si la transaction de niveau le plus élevé l'est.

```
transaction :
    transaction_header
    (formal_generics)?
    (formal_arguments)?
    (transaction_type_mark)?
    (obsolete)?
    (precondition)?
    (local_declarations)?
```

```
     transaction_body
     (postcondition)?
     (rescue)?
     'end'
     ;
transaction_type_mark :
     type_mark
     ;
transaction_header :
     transaction_designator
     ;
transaction_body :
     deferred | effective_transaction
     ;
effective_transaction :
     'do' (effective_command | effective_query)
     ;
effective_command :
     (rel_modification)*
     ;
effective_query :
     rel_query
     ;
deferred :
     'deferred'
     ;
```

Les sections **formal_generics**, **formal_arguments**, **obsolete**, **preconditons**, **local_declarations**, **postconditions** et **rescue** sont identiques à leurs contre-parties algorithmiques. Le **transaction_type_mark** (dont la présence indique qu'il s'agit d'une transaction de requête plutôt qu'une transaction de commande) est limité aux seuls types de relation. La section **transaction_body** constitue la description du traitement associé à la transaction et est spécifique à celle-ci.

Bien que retenues au sein de Discipulus, les parties suivantes ne sont présentement pas prises en charge par le traducteur : la forme générique, les arguments et la propriété obsolète.

Corps d'une transaction

Le corps d'une transaction est soit une expression relationnelle (sans effet de bord) dans le cas d'une transaction de type requête, soit une commande relationnelle (qui modifie généralement certaines des variables de relation constituant la base de données) dans le cas d'une requête de commande. Il y a trois commandes relationnelles : **insert**, **update** et **delete**; elles sont présentées à la section 3.3.5. Une transaction peut être différée, c'est-à-

dire que sa mise en œuvre n'est pas encore disponible (et non pas diffiérée dans son exécution, cet aspect étant automatiquement pris en charge par le mécanisme de gestion des transactions du SBDD). Ceci est un élément visant à faciliter le développement itératif, provenant d'Eiffel et conservé par souci de compatibilité.

3.3.3 La variable

Les variables se devise en trois catégories, les variables de base, les variables de type table, les variables de type Vue.

```
variable :
     variable_type identifier ':' relation_name
     ;
variable_type :
     'table' | 'variable' | 'view'
```

La relation est un générateur qui permet de définir des variables possédant le même en-tête. La valeur initiale de toute variable est une relation vide du type approprié.

- Variable : Une variable de relation est associée à une relation. Dans notre modèle, une base de données comprend une ou plusieurs variables de relation.
- Vue : Une vue est une relation virtuelle ou une représentation dont les données ne sont pas stockées dans une relation de la base de données. Une vue pourrait être définie à partir d'une expression relationnelle ce qui permet de rassembler des informations provenant de plusieurs relations.
- Table : une représentation d'une relation, mais pas équivalente. En fait, une table peut potentiellement contenir des lignes en double et implique un ordre particulier pour les lignes et les colonnes, alors qu'une relation est explicitement non ordonnée.

3.3.4 L'expression relationnelle

Vu l'adoption du modèle relationnel, les opérateurs de base de notre proposition sont ceux de l'algèbre relationnelle.

```
rel_exp :
     ('let' rel_equations 'in')? rel_op
     ;
rel_equations :
     rel_equation (',' rel_equation)*
     ;
rel_equation :
     IDENT '=' rel_exp
     ;
rel_op :
     rel_query | rel_modification | rel_denomination
     ;
rel_denomination :
     ('rename' att_renaming_list 'in')? rel_expression
     ;
rel_expression :
     rel_term (add_rel_op rel_term)*
     ;
rel_term :
     rel_factor (mul_rel_op rel_factor)*
     ;
rel_factor :
     relation_name | '(' rel_exp ')'
     ;
add_rel_op :
     'union' | 'minus'
     ;
mul_rel_op :
     'join' | 'inter' | 'cross' | 'by'
     ;
rel_query :
     'from' rel_list ( ':' relation_name )? rel_construction
     // En l'absence de relation_name, la dénotation des qualified_attribute_name de
     // chacune des attribute_value_def des attribute_value_def_list est obligatoire.
     // En présence de relation_name, les qualified_attribute_name sont soit tous
     // présents, soit tous absents (et alors l'association est positionnelle).
     ;
rel_list :
     (rel_item (',' rel_item)*)?
     ;
rel_item :
     rel_simple_item | '(' rel_exp ')' 'as' relation_name
     ;
rel_simple_item :
     rel_var_name
     ;
rel_construction :
     selection | groupement
     ;
selection :
     'select' attribute_value_def_list 'where' boolean_expression
     ;
groupement :
     'group' qualified_attribute_name_list 'adding' '(' attribute_value_def_list ')'
     ;
rel_item :
     rel_var_name
```

```
    ;
att_renaming_list :
    '(' ( attribute_renaming (',' attribute_renaming)* )? ')'
    ;
attribute_renaming :
    attribute_name 'as' attribute_name
    ;
attribute_value_def_list :
    (attribute_value_def (',' attribute_value_def)*)?
    ;
attribute_value_def options { k=2; } :
    (attribute_name ' :')? expression
    ;
qualified_attribute_name_list :
    (qualified_attribute_name (',' qualified_attribute_name)*)?
    ;
qualified_attribute_name options { k=2; } :
    (relation_name '.')* attribute_name
    ;
rel_modification :
    rel_insert | rel_update | rel_delete
    ;
rel_update :
    'update' rel_item 'set' set_list 'where' boolean_expression
    ;
set_list :
    (set_element (',' set_element)*)?
    ;
set_element :
    attribute_name ':=' expression
    ;
rel_delete :
    'delete' rel_item 'where' boolean_expression
    ;
rel_insert :
    'into' rel_item 'insert' (rel_expression | rel_tuple_list)
    ;
rel_tuple_list :
    rel_tuple (',' rel_tuple)*
    ;
rel_tuple :
    '[' attribute_value_def_list ']'
    ;
```

Définitions auxiliaires

Dans le cas des requêtes complexes nécessitant plusieurs sous-requêtes, la clause `let` permet d'accroître la lisibilité et de factoriser les sous-expressions communes.

Sélection

La sélection met en oeuvre conjointement la projection, la restriction et l'extension en construisant une relation à partir d'un sous-ensemble des attributs d'autres relations en fonction d'un critère de restriction (prédicat ou expression logique de prédicats).

Groupement

Le groupement construit une relation B en projetant les tuples d'une relation A sur un ensemble d'attributs qui formera la clé candidate de B, les tuples de B pouvant être étendus à l'aide de fonctions d'agrégation.

Opérateurs ensemblistes

Les opérations ensemblistes (union, intersection, différence, produit) sont présentes. Les opérations proprement relationnelles join (pour la jointure naturelle[10]) et by (pour la division de Codd[11]) aussi.

Commandes de modification

La définition d'une affectation permettrait de rendre compte de toutes les possibilités de modification des variables de relation. Nous avons préféré nous en tenir à l'approche de SQL, autant pour des raisons « d'habitude » que par souci de faciliter le travail d'optimisation du traducteur. Les trois opérations de modification sont donc :

- Ajouter des tuples : l'opération d'insertion permet d'ajouter un ou plusieurs tuples à une relation, les tuples peuvent être spécifiés d'une de façon explicite ou exprimés par une expression relationnelle. Dans ce cas l'en-tête de la relation résultante doit être le même que celui de la relation modifiée.

---

[10] La jointure naturelle partielle (à un sous-ensemble des attributs communs) sera incluse dans la prochaine version de Discipulus ; le produit cartésien pourrait alors être retiré.

[11] Les variantes ultéireurs de la division (small divide et large divide) pourraient être considérées dans un version ultérieure.

- Modifier des tuples existants : l'opération de mise à jour permet de modifier des valeurs des attributs d'une relation. Une opération de mise peut contenir une clause `where`.
- Supprimer des tuples : la suppression porte sur tous les tuples de la relation pour laquelle la clause `where` est vraie.

3.4 La gestion des bases données et des accès

Le langage de gestion des bases de données (LGBD) regroupe l'ensemble des possibilités permettant de créer, activer, désactiver, cloner, renommer, détruire, sauvegarder et restaurer une base de données

Nous pouvons aussi considérer une extension possible de ces opérations permettant de traiter une BD virtuelle (une BD virtuelle est une image locale d'une BD hébergée sur un autre serveur)[12].

Le schéma de bases de données

Le LGBD de Discipulus comprend les instructions permettant de modifier la structure d'une base de données et de ses relations. Les actions de modification d'une relation sont les suivantes :

- Ajouter, supprimer, changer le nom ou changer le type d'un attribut.
- Ajouter ou retirer une contrainte.

Le langage définit aussi les instructions pour renommer ou cloner une relation. Aucune des instructions de modification n'est présentement supportée par le traducteur.

ON a aussi prévu d'autres instructions pour la gestion des accès et des objets qui lui sont liés.

---

[12] La question à savoir si cette image comprend ou non une copie totale ou partielle de la BD réelle ne doit pas être définie dans Dominus (c'est une question de mise en œuvre qui doit demeurer opaque).

Les accès

Le LGBD comprend des instructions permettant d'ajouter, de supprimer, de modifier et de configurer un compte d'utilisation (appelé le plus souvent « utilisateur »). Un compte d'utilisation comprend l'information requise pour le contrôle d'accès aux services offerts par les bases de données. Ce contrôle est exercé par le SBDD en fonction des droits accordés au compte ou selon un politique d'administration des rôles et des privilèges accordés des groupes de comptes.

Les éléments de gestion de l'accès ne sont pas présentement pris en charge par le traducteur Discipulus.

La syntaxe envisagée

```
schema_evolution :
     (schema_instruction)*
     ;
schema_instruction :
     component_definition | component_importation | component_alteration |
component_remove | component_cloning_or_renaming
     ;
component_definition :
     ('define'|'replace') component
     ;
component_cloning_or_renaming :
     ('clone'|'rename') component_designator 'as' identifier_list
     ;
component :
     policy | eiffel_class | relation | variable | transaction
     ;
component_type :
     'policy' | 'class' | 'relation' | 'variable' | 'transaction'
     ;
component_importation :
     import_action component_designator import_version 'from' external_ressource
     ;
import_action :
     'import' ('again')?
     ;
import_version :
     'version' string_constant
     ;
component_remove :
     'remove' component_designator
     ;
component_alteration :
     'alter' component_designator 'do' component_action_list 'end'
     ;
component_action_list :
```

```
     (component_action)*
     ;
component_action :
     part_obsolete | part_definition | part_alteration | part_remove
     ;
part_obsolete :
     'obsolete' part_designator
     ;
part_definition :
     'define' part_designator 'as' part_body
     ;
part_alteration :
     'alter' part_designator 'as' part_body
     ;
part_remove :
     'remove' part_designator
     ;
part_designator :
     part_type identifier
     ;
part_body :
     role_part | invariant_part | creator_part | converter_part | precondition_part |
postcondition_part | transaction_body_part | feature_part | attribute_part
     ;
part_type :
     'role' | 'invariant' | 'creator' | 'convertor' | 'precondition' | 'postcondition' |
'transaction' | 'feature' | 'attribute'
     ;
```

Le LGBD de Discipulus n'est présentement pas pris en charge par le traducteur.

3.5 Synthèse

Nous avons présenté un bref survol du langage Discipulus et dressé l'inventaire des éléments qui ne sont pas pris en compte par le traducteur courant. Pour conclure, nous présentons à l'annexe K un petit exemple de définition d'un schéma de bases de données comprenant quelques tables et quelques transactions.

Chapitre 4

Mise en œuvre expérimentale

Pour valider nos propositions, nous avons besoin d'exécuter des programmes écrits en Discipulus. Plutôt que de réaliser un système original complet, nous avons opté pour une traduction des programmes Discipulus en SQL suivie de leur exécution par un SGBDR existant. Ce chapitre présente les modalités et les choix d'implémentation de cette solution.

4.1 Caractérisation du banc d'essai du langage

Ultimement, nous désirons pouvoir soumettre un SGBDR mettant en œuvre Discipulus au banc d'essai développé par A. Ottavi [L12] afin de le comparer à des SGBDR existants tels que DB2, Oracle, PostgreSQL, MySQL et Rel. Dans un premier temps toutefois, il nous faut valider la définition même du langage (et y apporter les changements qui s'avéreront nécessaires) en développant un SGBDR qui servira de banc d'essai au langage lui-même. C'est ce dernier but que nous nous sommes fixé ici.

4.1.1 Objectifs

Nous avons retenu les objectifs d'évaluation du langage Discipulus suivants :

- l'expressivité de la portion relationnelle du langage Discipulus;
- l'arrimage de la portion relationnelle avec la portion algorithmique du langage Discipulus grâce à un système de typage adéquat;
- l'adéquation de la gestion des attributs annulables;
- les gains et les pertes d'efficacité induites par l'adhésion stricte au modèle relationnel (omission des doublons, absence d'ordonnancement des attributs et des tuples);
- l'efficacité du mécanisme de la gestion des exceptions;

- l'exactitude des résultats de la traduction.

Nous délaissons pour le moment les objectifs suivants :

- l'adéquation du langage algorithmique (issu d'Eiffel et déjà évalué indépendamment);
- l'adéquation du langage transactionnel (hors de la portée du mémoire);
- la traduction du mécanisme de l'héritage multiple (non encore mis en œuvre);
- les instructions pour la gestion des agents (non encore mis en œuvre);
- l'intégration de code et de bibliothèques provenant d'autres langages et systèmes (non encore mis en œuvre);
- la gestion des utilisateurs, de l'accès aux données et de la sécurité en général (non encore mis en œuvre);
- la gestion des transactions et des services (non encore mis en œuvre);
- la modification, le clonage et le renommage des composants relationnels (non encore mis en œuvre).

4.1.2 Système visé

Développer un système de compilation et d'exécution de programmes Discipulus est un projet dont l'envergure dépasse le cadre de la maîtrise compte tenu du temps et des moyens que nous possédons. De plus, un projet de cette envergure ne saurait être engagé avant d'avoir validé la pertinence de notre proposition. Un banc d'essai d'envergure plus modeste s'impose donc.

Nous avons décidé de construire le système de compilation et d'exécution de programmes Discipulus à l'aide de deux composants : un compilateur du langage Discipulus vers le langage SQL et un interprète de programmes écrits en SQL. L'utilisation d'un SGBDR existant comme interprète allégera considérablement l'effort requis.

4.1.2.1 Compilateur

Le compilateur Discipulus prend en charge la vérification des programmes Discipulus et leur traduction en SQL. Le compilateur est donc divisé en deux composants principaux : l'analyseur et le traducteur. Le traitement passe par plusieurs étapes, tout d'abord l'analyseur

vérifie la syntaxe des programmes Discipulus et leur validité. Si le programme est valide, la main est passée au traducteur pour produire du code SQL.

4.1.2.2 Interprète

L'interprète se charge de l'exécution des résultats de la traduction. Dans cette étape, nous faisons appel directement à un SGBDR indépendant afin d'exécuter le code SQL résultant de la traduction. Dans un premier temps, nous avons choisi Oracle 10g comme SGBDR et une exécution manuelle, en dehors du système, utilisant un logiciel tiers, tel SQL Developper ou Navicat.

4.1.3 Exigences fonctionnelles

- F1 — Langage source. Le langage analysé est celui décrit dans le rapport de recherche [R3], mais prenant en compte les exclusions mentionnées au chapitre 3.
- F2 — Vérification des règles de validité. L'analyseur doit vérifier les règles de validité (autant celles provenant d'Eiffel que celles découlant des parties propres du langage Discipulus). Ainsi, le traducteur demeure simple puisqu'il n'a à traiter que des programmes corrects.
- F3 — Langage cible. La traduction des programmes Discipulus doit être réalisée vers le dialecte SQL d'Oracle. Ce choix est conditionné par celui fait par Aurélie Ottavi, afin de faciliter éventuellement l'adaptation de son banc d'essai. Il impose l'utilisation de PL/SQL à titre de langage de définition de PSM.
- F4 — Exactitude des résultats. Les résultats de l'exécution de la traduction SQL des programmes Discipulus doivent donner des résultats exacts permettant de répondre à la question exprimée en Discipulus.
- F5 — Gestion des attributs annulables. En fait, l'application doit pouvoir gérer la détection des attributs annulables dans chaque requête et traiter toute évaluation avec une valeur absente de cet attribut comme une exception de type NullException.
- F6 — Gestion stricte du modèle relationnel, plus particulièrement l'interdiction des doublons dans les relations c'est-à-dire que les résultats de l'exécution du code SQL produit ne comportent pas de doublons.

4.1.4 Exigences non fonctionnelles

- NF1 — L'utilisation du système doit être simple et bien documentée. Idéalement, pour permettre à des étudiants universitaires de l'utiliser dans le cadre d'un laboratoire.
- NF2 — Le système doit être portable. Ultimement, il doit pouvoir être utilisé sur les plateformes Windows (XP, Vista et 7), Mac OS (10.4 et suivants) et Linux (Ubuntu 10.4 et suivants).
- NF3 — La conception du système doit être bien documentée. Le système est expérimental, donc appelé à de fréquentes modifications et évolutions. La documentation est essentielle afin d'assurer sa compréhension, sa maintenabilité et son évolution par les différents collaborateurs qui participeront au projet.
- NF4 — La performance de traduction doit être acceptable. Le seuil a été établi à au moins 1000 lignes à la minute sur un poste de travail semblable à ceux fournis aux étudiants dans les laboratoires du Département d'informatique.
- NF5 — La performance d'interprétation des programmes doit être acceptable. Le facteur de ralentissement par rapport à un même programme écrit en SQL à l'origine doit être au plus 5 sur un poste de travail semblable à ceux fournis aux étudiants dans les laboratoires du Département d'informatique.
- NF6 — Grammaire LL(1). Cette famille de grammaires et les langages qu'elles décrivent sont conceptuellement simples et beaucoup plus faciles à analyser et finalement beaucoup plus intuitifs pour un être humain. Les contraintes qu'elles imposent augmentent la lisibilité, donc l'exactitude et la fiabilité des programmes, un des objectifs de notre projet. Finalement, les grammaires LL offrent la possibilité d'évaluer autant les attributs hérités que synthétisés en cours d'analyse, ce qui simplifie la mise en œuvre.
- NF7 — Arbre syntaxique abstrait. L'analyseur doit faire une vérification lexicale et syntaxique complète et produire un arbre syntaxique abstrait. Ceci permet d'utiliser un système de générateurs pour cette partie du code et sépare clairement l'analyseur en composants minimalement couplés.

4.2 Stratégie globale

Ayant choisi la voie de la traduction vers SQL, nous ensuite avons ensuite étudié deux avenues pour développer le traducteur et faire le lien avec le système d'exécution :

- Modifier un compilateur Eiffel existant.
- Écrire un traducteur Discipulus vers SQL à l'aide d'un système d'écriture de compilateur.

Nous avons évalué ces deux stratégies en regard des critères suivants :

- minimiser l'effort de développement requis;
- obtenir un système transportable (Windows, Mac OS, Linux);
- obtenir un système fiable;
- obtenir un système facilement modifiable.

4.2.1 Modification d'un compilateur Eiffel

L'approche consiste à modifier un compilateur existant de façon à :

- ajouter les éléments propres au langage Discipulus;
- adapter les éléments d'Eiffel aux exigences de notre modèle;
- produire du code correspondant à la partie relationnelle (vraisemblablement en utilisant une bibliothèque telle que Berkeley DB [S8]).

Pour des raisons économiques, nous sommes limités aux compilateurs dont le code est disponible gratuitement (*Open source*). L'inventaire de tels compilateurs est réduit, nous en avons recensé trois et ils sont distribués sous les termes de la licence LPG.

Le compilateur GOBO

- Description : le projet GOBO [S9] consiste à mettre en place un ensemble de bibliothèques compatibles avec les principaux compilateurs Eiffel. Le compilateur GOBO est un compilateur entièrement compatible avec ISE Eiffel et conforme à la norme ISO 25436 [S14]. Il utilise les techniques de compilation différentes d'ISE Eiffel.

- Limitations : le compilateur souffre présentement de plusieurs limitations, bogues et incompatibilités avec la norme ISO 25436 [S14]. Ainsi, les exceptions et les assertions ne sont pas prises en compte.
- Statut : le projet semble suspendu depuis 2008, la dernière version (3.9) est incomplète et datée de 2008.
- Références : http://www.gobosoft.com/eiffel/gobo/gec/index.html.
- Utilisabilité dans le cadre de notre projet : l'utilisation de ce compilateur nécessite d'une part l'implémentation des constructions qui manquent et qui sont importantes dans notre proposition comme les exceptions et les assertions.

Le compilateur TECOMP

- Description : Le compilateur TECOMP [S10] est conforme à la norme ISO 25436. Il compile les programmes Eiffel dans une représentation interne, il valide, puis exécute le programme dans une machine virtuelle Eiffel.
- Limitations au niveau du langage : des fonctionnalités pas encore implémentées dans le langage (les instructions else et then, les instructions de débogage, les contraintes associées à une classe, etc.).
- Limitations au niveau des bibliothèques du noyau (des classes non encore disponibles) : STORABLE, MEMORY, ARGUMENTS, PLATFROM, ONCE_MANAGER.
- Statut : le projet est en cours du développement la dernière version du compilateur est 0.24.1 daté du 21-01-2011. Mais celle-ci n'est pas une version finale, le compilateur reste toujours très incomplet.
- Références : http://tecomp.sourceforge.net/index.php?file=doc/tecom.
- Utilisabilité dans le cadre de notre projet : ce compilateur n'est clairement pas assez complet pour le moment.

Le compilateur SmartEiffel

- Description : SmartEiffel [S9] est un compilateur complet, petit et plus rapide que les autres compilateurs open source. Il fonctionne sur plusieurs plates-formes. En fait,

SmartEiffel devrait fonctionner sur n'importe quelle plate-forme si un compilateur C ou une JVM existe.

- Limitations : le compilateur n'implémente pas la norme ISO 25436 [S14], mais une version antérieure du langage définie dans la version 2.0 publiée en 2004 [S9].
- Statut : le développement est arrêté depuis 2005.
- Références : http://smarteiffel.loria.fr/.
- Utilisabilité dans le cadre de notre projet : même si le compilateur est complet, il nécessite une adaptation pour le rendre conforme à la norme ISO 25436; ceci nous semble d'autant plus périlleux que le soutien et la communauté d'utilisateurs sont réduits puisque le projet est arrêté depuis plus de 7 ans.

Conclusion

Cette approche implique des modifications ou des adaptations de certaines fonctions au niveau du langage algorithmique, et l'ajout des fonctionnalités associées au langage relationnel. Cela nécessite une analyse en profondeur du compilateur sélectionné. Nous remarquons également que chacun des compilateurs à des limitations par rapport à nos besoins.

Cette solution est envisageable, mais est incertaine quant au temps et aux ressources qui seront requis pour la bonne compréhension de la base de code existante, compte de l'absence de soutien effectif (les trois projets étant suspendus).

Finalement, le choix de la bibliothèque relationnelle est déterminant par rapport aux performances (NF5), si on se réfère à celles (très décevantes) du système Rel qui a retenu cette approche et la bibliothèque BerkeleyDB [S8]. Compte tenu des risques importants liés à l'utilisation d'un compilateur existant, cette avenue a été écartée et l'étude du choix de la bibliothèque la plus appropriée n'a pas été entreprise.

4.2.2 Utilisation d'un système d'écriture d'analyseurs

L'utilisation des outils de construction du compilateur nous permet d'automatiser presque toutes les phases de l'écriture du composant analyseur et fournit une base pour l'écriture du composant traducteur, l'outil choisi est présenté dans la section 5.4.1.

Ayant opté pour ces outils, nous avons retenu l'architecture suivante :

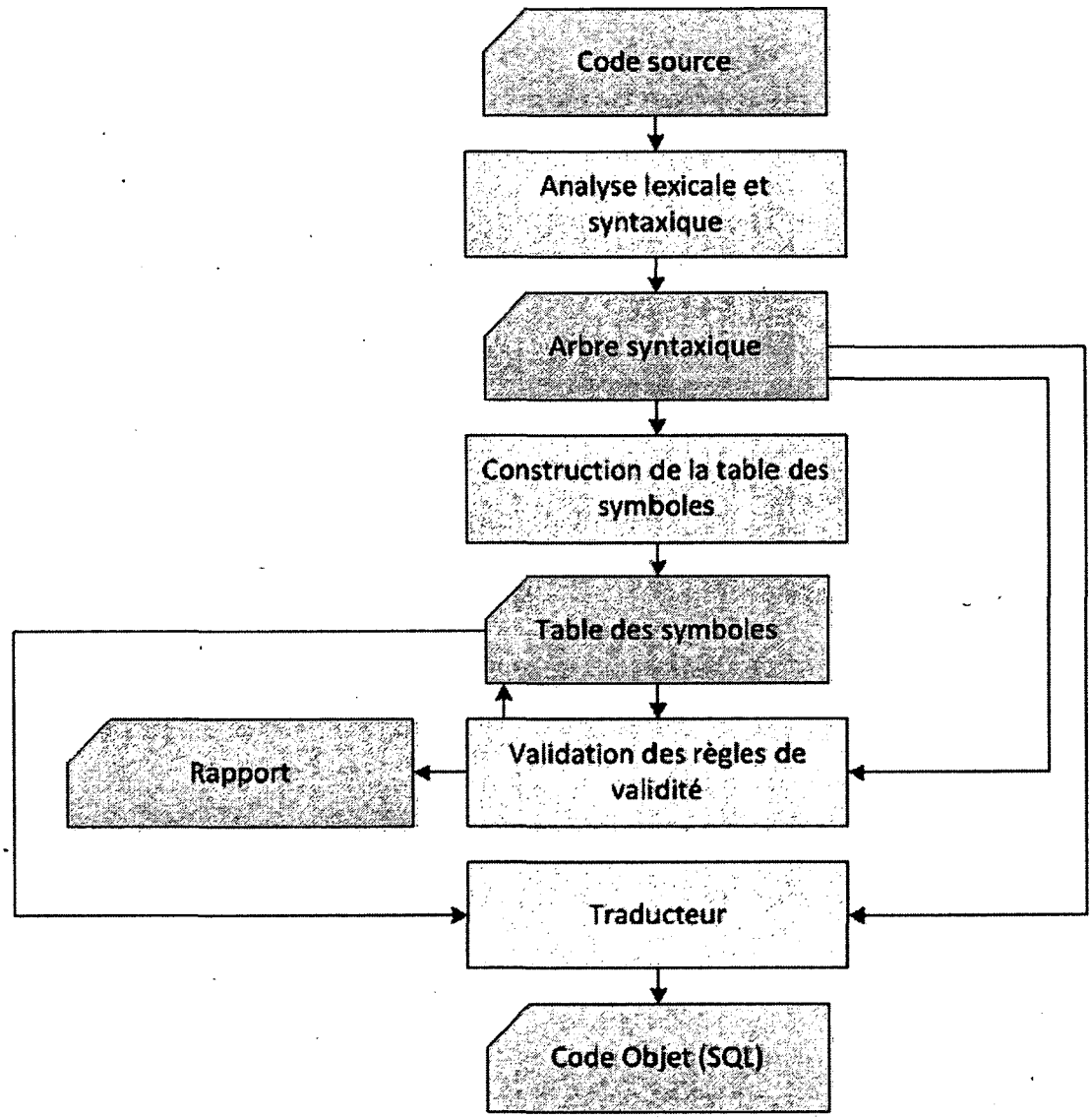

Figure 4-1 : Flux des données majeures du traducteur

La figure ci-dessus représente les principaux processus du traducteur, l'analyse lexicale est syntaxique sont effectués par les modules générés à l'aide de JavaCC [S3] présentés dans la section 4.4.2 Outil de génération d'analyseurs, tandis que la construction de la table des symboles (annexe F), module responsable du contrôle des règles de validité et les règles de traductions ont été développées en Java. L'implémentation de ces modules est basée sur le patron de conception visiteur qui nous permet de parcourir les nœuds de l'arbre syntaxique.

L'ordre d'exécution des processus est le suivant :

- analyse lexicale,
- analyse syntaxique,
- construction de la table des symboles,
- contrôle des règles de validité,
- traduction.

4.2.3 Architecture

Le traducteur est composé des six modules suivants, décrits ci-après :

- analyseur lexical et syntaxique,
- contrôleur des règles de validité,
- gestionnaire des symboles,
- traducteur,
- gestionnaire des erreurs,
- gestionnaire de configurations
- et console.

Analyseur lexical et syntaxique

Ce module construit l'arbre syntaxique du programme à partir de l'analyse syntaxique et lexicale du code source.

Le principe de l'analyse lexicale est de transformer une suite de symboles en terminaux (nombres, symboles (-, +, etc.), identifiants...). Le but de l'analyseur lexical est de découper le fichier source en symboles terminaux.

Après l'analyse lexicale, la main est passée à l'analyse syntaxique qui permet de vérifier la validité d'un programme par rapport à la grammaire requise reconnue par JavaCC [S3]. Cette tâche est accomplie par un analyseur syntaxique qui reconnaît une grammaire de la forme BNF. Au cours de cette analyse, il est possible de définir des actions lorsqu'une règle est reconnue. La table des symboles est construire grâce à de telles actions.

Contrôleur des règles de validité

Le contrôle des règles de validité se charge de vérifier si une instruction vérifie une condition en particulier. Par exemple, il est impossible de définir deux classes, variables ou deux fonctions ayant le même nom. Toutes les règles de validité sont identifiées et décrites dans la norme ISO 25436 [S14].

Après l'analyse du programme et la construction de la table des symboles, le traducteur passe encore une fois sur le programme afin de vérifier les règles de validité du programme.

Exceptionnellement, certaines règles sont vérifiées au fur à mesure de la construction de la table des symboles :

- Les noms des classes : on ne peut pas ajouter une classe avec un nom déjà enregistré dans ma table de symboles.
- Les noms des attributs, des fonctions, des procédures et des constantes : on ne peut pas avoir une classe constitue de deux propriétés avec les mêmes noms.
- Les noms des classes hérités : on ne peut pas avoir une classe qui hérite d'une classe qui n'existe pas déjà.

Ensuite un passage à travers l'arbre syntaxique est nécessaire afin de compléter la vérification des règles de validité par exemple les règles de validité associée à la boucle ou aux instructions conditionnelles. Le parcours de l'arbre syntaxique est basé sur le patron de conception VISITOR. L'outil JJTree [S6] ajoute dans chaque classe de nœud une méthode jjtAccept() et génère aussi une interface VISITOR qui peut être implantée et passée aux nœuds à accepter, cela nous permet d'appliquer la logique de la vérification des règles de validité sur les éléments que nous n'avons pas pu traitée à partir de la table de symboles.

Pour une question de cohérence nous avons maintenu la même stratégie pour le contrôle des règles de validité associées au langage relationnel.

Gestionnaire des symboles

Le module responsable de la construction de la table des symboles. La table des symboles est une structure complexe qui peut évoluer en fonction des besoins. Chaque symbole sera identifié par un nom et décrit par une structure définie au préalable (annexe F). La structure rassemble toutes les informations utiles concernant les classes et les relations, cela inclut le nom et le module et toutes les informations des composants, une classe nous est constitue essentiellement pas les variables, les fonctions, procédures. Pour toute variable, elle garde l'information de son nom, son type. De plus pour toute fonction ou procédure, elle garde l'information de son nom, le nom et le type de ses arguments, ainsi que le type du résultat qu'elle fournit. De même pour une relation, il conserve les noms et le type de ces attributs, leur statut ainsi que l'ensemble des contraintes appliquées à chaque attribut. Il sera possible d'ajouter un symbole ou de le retirer ou de rechercher un symbole avec son identifiant.

Finalement, la table des symboles sera utilisée par les deux derniers composants : le contrôleur des règles de validité et le traducteur.

Traducteur

Ce module comporte toutes les classes nécessaires pour la traduction des composants du langage Discipulus. Après avoir terminé l'analyse lexicale et syntaxique ainsi que la construction de la table des symboles, le module procède à la traduction du code source en SQL. Une description détaillée sur la stratégie de traduction est définie dans la section 4.3.

Le parcours de l'arbre syntaxique utilise les mêmes techniques de programmation que pour l'évaluation des règles de validité.

À cette étape, nous faisons appel au gestionnaire de configuration (voir plus bas) afin de prendre en charge les éléments de la traduction concernant le SGBD ciblé; par exemple, la traduction des types primitifs et les fonctions d'agrégation.

Gestionnaire des erreurs

Ce module est représenté par un singleton, utilisé tout au long de l'exécution pour accumuler les signalements d'erreur à la fin de chaque opération.

Ce module permet de produire à la fin de chaque traduction un rapport sur les erreurs rencontrées. Le rapport est sous forme d'un fichier texte, chaque erreur est enregistrée avec le fichier source et le type d'erreur.

Gestionnaire de configuration

Ce module permet de paramétrer la traduction du langage DISCIPULUS en fonction du choix de représentations et, dans le futur, du SGBD ciblé. La configuration comporte les éléments différents d'un SGBD à un autre, il suffit de spécifier le SGBD ciblé pour pouvoir chargé les éléments nécessaires pour effectuer la traduction. Pour le moment, les éléments configurables sont : les types primitifs et les fonctions d'agrégation. La traduction des types primitifs est séparée selon la cible de la traduction, en fait la traduction n'est pas la même si l'élément est associé au langage relationnel ou au langage algorithmique. Par exemple, la traduction des types dans une table n'est pas la même pour les arguments d'une fonction ou d'une procédure.

Par exemple, la traduction du type INTEGER n'est pas la même, dans le cas d'un paramètre de fonction ou de procédure, la traduction est NUMERIC sans précision, tandis que dans le cas d'un attribut de table, la traduction est NUMERIC(10) avec une précision. Cette différence est due au comportement des SGBD tel que Oracle et MySQL.

Nous avons utilisé des fichiers XML afin de permettre à un utilisateur de lire, écrire et modifier des informations de configuration. Ces informations sont chargées pendant la phase de la traduction.

Console d'exécution (non réalisé)

Ce module permet de soumettre la traduction SQL à un SGBD et en récupérer les résultats. Il comprend la gestion des interfaces entre notre traducteur et les SGBD (avec des pilotes JDBC, par exemple).

Nous avons reporté le développement de ce module, mais nous jugeons qu'il sera très utile de le compléter, car il permettra une réduction considérable du temps d'exécution des tests.

Pour le moment, l'application est livrée sous format d'une archive Java (discipulus.jar). La commande d'exécution est :

```
java –jar discipulus.jar _in _out
```

où

- _in : représente le chemin du dossier contenant les programmes à traduire.
- _out : représente le chemin du dossier dans lequel sont déposées les traductions.

Les traductions doivent ensuite être soumises à une console d'exécution Oracle (SQL Developper, SQL*Plus, Navicat, etc.).

4.3 Stratégie de traduction

Les règles abstraites de la traduction en SQL-PSM de chaque instruction du langage Discipulus sont définies dans le rapport de recherche [R4] rédigé par l'auteur en collaboration avec C. Khnaisser et L. Lavoie.

4.3.1 Traduction des types primitifs

Pour traduire les types primitifs de notre modèle, nous avons étudié trois possibilités :

Définition des types primitifs grâce au mécanisme de classe

Adopter l'approche d'Eiffel pour définir les types primitifs par des classes serait très souple en offrant la possibilité d'adapter la mise en œuvre dans chacun des contextes. Par contre, cela semblait difficile à réaliser dans une approche par traduction en SQL (non-accès à la représentation interne des types, par exemple) et cela demandait en soi un effort de programmation beaucoup plus grand.

Définition implicite des classes en PL/SQL

Définir implicitement les classes de chacun des types primitifs et les mettre en œuvre en PL/SQL est envisageable, mais cela nécessite de faire la mise en œuvre pour chaque dialecte SQL, ce qui demandera un grand effort de programmation.

Correspondance avec les types primitifs de SQL

Faire correspondre les types primitifs de Discipulus à des types choisis au sein de l'offre du dialecte SQL ciblé. Pour prendre en compte la non-normalisation de ces types, les fichiers de configuration ont dû être introduits (voir section 4.2 sur le gestionnaire de configuration), la correspondance pour Oracle est la suivante :

Tableau 4-1 : Correspondance des types primitifs en SQL-Oracle

Discipulus	*SQL-Oracle*
INTEGER	NUMERIC
NATURAL	NUMERIC
CHARACTER	NCHAR
REAL	REAL
STRING	NVARCHAR2
BOOLEAN	BOOLEAN

Mise en œuvre retenue

Nous avons retenu la troisième solution parce que c'est la solution la plus facile à mettre en place selon l'approche retenue pour la traduction en SQL. Cette solution ne pourra toutefois pas être utilisée à long terme, car elle impose trop de restrictions par rapport aux définitions de Discipulus. En fait, chaque type peut avoir des contraintes de représentation physique, par exemple, pour le type STRING, le compilateur peut changer la taille en fonction des contraintes de représentation physique. En SQL, le choix est explicite et non modifiable (CHAR, VARCHAR et CLOB). La première approche semble la plus cohérente et la plus flexible avec les exigences du modèle proposé.

La traduction d'un même type est différente selon le contexte :

- Au sein d'une expression relationnelle.
- Lors de la définition d'un attribut de classe.

- Lors de la définition d'un paramètre de fonction ou de procédure.

Par exemple, la traduction d'un type numérique est différente selon le contexte, pour les arguments des fonctions ou des procédures il ne faut pas indiquer la précision du type par contre pour les attributs il faut l'indiquer, de même Oracle ne dispose pas de type BOOL pour le langage relationnel, mais nous pouvons l'utiliser dans le langage algorithmique.

4.3.2 Langage algorithmique

La classe constitue le concept fondamental du langage algorithmique, les sections suivantes montrent la stratégie utilisée pour la traduction des constructions du langage algorithmique, il faut noter que nous avons reporté l'étude et la traduction des éléments suivants pour un travail futur, la liste des éléments est présentée dans le chapitre 3.

Nous présentons ci-après la stratégie de traduction de chaque élément algorithmique ainsi que les motivations de nos choix. Pour ne pas alourdir le texte, nous avons regroupé à l'annexe J les différents exemples illustrant nos stratégies de traduction. Dans le corps de la section, nous référerons.

4.3.2.1 Classe

Une classe est simplement traduite par un type objet. Les propriétés d'une classe comprennent les attributs, les constantes, les fonctions, les procédures et les invariants.

La définition d'un type en SQL est divisée en deux parties :

- La spécification du type qui est constituée des déclarations des attributs, des constantes, et des signatures des routines (fonctions et procédures).
- Le corps du type qui comprend la définition du corps des routines déclarées dans la spécification du type.

La traduction d'une classe sera devisée selon les deux mêmes parties.

Les constantes et les attributs sont traduits en PL/SQL par des constantes et des attributs. Il en est de même pour les procédures et les fonctions puisque PL/SQL permet de définir les

procédures et les fonctions. La définition des antécédents, des conséquents et des invariants est traitée ci-après.

Exemple 1 à l'annexe J.

4.3.2.2 Antécédents et conséquents

Les antécédents et les conséquents sont des assertions à vérifier respectivement au début et à la fin de chaque procédure ou fonction. On s'assure ainsi que le traitement de la fonction ou de la procédure respecte le contrat.

Chaque assertion est traduite par une fonction (de type BOOLEAN) dont le corps est une instruction conditionnelle correspondant à la condition de l'assertion. Si l'assertion est valide la fonction retourne TRUE, dans le cas contraire une exception est levée pour aviser que le contrat est non valide.

Les assertions et de gestion des exceptions sont deux mécanismes complémentaires. Le mécanisme des assertions nous permet d'exprimer et de déclarer des contraintes tandis que le mécanisme des exceptions permet de traiter la violation de ces contraintes par la spécification d'actions de reprise. La présence d'affirmations contribue à une spécification plus formelle par l'expression des propriétés sémantiques de classes. Si la vérification des antécédents et des conséquents échoue, une exception est levée au point de l'appel.

Le traitement des exceptions qui découle de la violation des antécédents et des conséquents au niveau de la transaction n'est pas étudié dans ce mémoire.

Voir l'exemple 1 à l'annexe J.

4.3.2.3 Invariants

Un invariant de classe est une assertion portant sur l'état de chaque instance de la classe et qui doit être vrai en tout temps depuis le moment où l'instance est créée jusqu'au moment où elle est détruite. Si l'instance ne peut être modifiée que par les procédures de la classe, il est suffisant de vérifier l'assertion à la fin de chacune des procédures de la classe. Par contre, si les fonctions peuvent avoir des effets de bord, il faut aussi le faire à la fin de celles-ci.

Finalement, si une instance peut être modifiée sans passer par les routines de la classe, la vérification ne pourra être que partielle, au mieux est-il possible de vérifier en plus l'assertion au début d'une routine.

Nous ne pouvons pas définir directement des invariants dans un type en PL-SQL. Mais nous avons décidé de les représenter sous forme de routines permettant de faire les vérifications sur l'invariant de la classe.

Ces procédures sont appelées au début et à la fin de chaque appel de fonction ou procédure pour s'assurer que l'invariant et toujours respecter au début et à la fin de l'exécution. Cela entraîne trop de code pour la traduction des fonctions. La deuxième possibilité est de vérifier l'invariant juste à la fin des fonctions et des procédures, mais il faut s'assurer que l'invariant est préservé initialement juste après à la création de l'objet.

Ainsi, si on considère une méthode F, où l'on a appliqué un contrat d'antécédents, de conséquent et d'invariance, l'ordre de vérification des contrats est le suivant :

```
F () {
      // Vérifie l'invariant : dpl_invariant ()
      // Vérifie la précondition : dpl_assertion _Preconditon()

      // Contenu de la méthode

      // Vérifie la postcondition : dpl_assertion _Postcondition()
      // Vérifie l'invariant en sortie : dpl_invariant ()
}
```

Par contre dans le système transactionnel que nous n'avons pas pu étudie dans le cadre de ce mémoire, les invariants des objets utilisés dans une transaction peuvent ne pas être vérifié qu'à la fin de la transaction.

Voir les exemples 1 à l'annexe J.

4.3.2.4 Exceptions

Lorsque le traitement est transféré du bloc BEGIN dans le BLOC EXCEPTION, il n'est plus possible de revenir dans le corps du bloc BEGIN, ce qui ne répond pas exactement a la stratégie de reprise définie dans Discipulus, la solution retenue pour simuler ce système est

l'utilisation d'une boucle qui nous permet de reprendre le traitement de la fonction ou de la procédure avec la nouvelle stratégie, le nombre d'itérations est limité a 3.

Voir l'exemple 1 à l'annexe J.

4.3.2.5 Instructions et déclaration de variables locales

Il y a correspondance étroite entre les possibilités offertes par Discipulus et PL/SQL.

Voir les exemples 1, 2 à l'annexe J.

4.3.2.6 Héritage simple

Pour l'héritage simple, nous avons simplement utilisé le mécanisme déjà présent en PL/SQL.

Voir l'exemple 2 à l'annexe J.

4.3.2.7 Héritage multiple

L'héritage multiple est un des éléments les plus importants de notre proposition, sa traduction nécessite cependant une réflexion approfondie que nous amorçons ici, sans la compléter ni la mettre en œuvre.

La première possibilité pour traduire le mécanisme de l'héritage est de fusionner le code des parents dans la classe fille. Cela entraîne une quantité du code énorme et nécessite aussi de « convertir » les représentations des objets lorsqu'ils sont passés dans des contextes ou un objet apparentant à un ancêtre ou à un descendant est requis. La tâche est titanesque et cette solution semble d'emblée devoir être écartée.

La deuxième solution consiste à simuler de la sémantique par pointeur à l'aide des relations et des clés. Tous les objets d'une classe sont stockés dans une table associée à la classe. À chaque objet, on associe une clé primaire (artificielle automatique) qui joue le rôle d'adresse dans le cadre de la sémantique des pointeurs. La mise en oeuvre est alors la même que celle normalement utilisée en mémoire, notamment pour l'héritage multiple (on ne passe jamais un objet, mais un pointeur sur l'objet – ici, la clé). Cette solution présente toutefois deux problèmes : un premier problème pressenti au niveau de la performance, un deuxième au

niveau de la gestion transactionnelle. En effet, la sémantique de classes doit être en dehors de la portée transactionnelle (principe de la validation immédiate des assertions). Il n'est pas clair que cela pourrait être réalisé, à partir du moment où les objets sont simulés par des lignes au sein de tables.

La troisième solution consiste à utiliser l'héritage simple pour simuler l'héritage multiple. En fait, on utilise l'héritage simple pour une des classes parentes alors que les autres héritages sont réalisés par fusion de code, comme pour la première technique. Cette technique a l'avantage de traiter simplement et efficacement le cas le plus fréquent (l'héritage simple) tout en permettant de traduire l'héritage multiple.

Pour conclure, nous constatons que la question demeure ouverte dans un contexte de traduction vers SQL.

4.3.3 Langage relationnel

Dans cette section nous traitons les éléments suivants :

- relation,
- variable,
- opérateurs relationnels et de mise à jour,
- traitement d'annulabilité et
- traitement d'exception .

Nous avons mis de côté les tuples et les opérateurs transactionnels.

Nous présentons ci-après la stratégie de traduction de chaque élément ainsi que les motivations de nos choix. Pour ne pas alourdir le texte, nous avons regroupé à l'annexe J les différents exemples illustrant nos stratégies de traduction. Un exemple complet, Distribution, est également proposé à l'annexe K.

4.3.3.1 Relation

Toutes les propriétés de la relation sont sauvegardées dans la table de symboles. Ces propriétés sont requises, chaque fois qu'une variable est de type relation est traduite, c'est-à-

dire que la relation n'est pas traduite, mais la définition est répétée autant de fois qu'il y a de variables. En fait, une relation est un générateur qui permet de définir des variables d'un type particulier.

Pour chaque variable de type relation, la traduction des composants de la relation est la suivante :

Discipulus	**SQL**
Attributs	Attribut d'une table avec la propriété NOT NULL.
Attribut annulable	Un attribut annulable est traduit par deux attributs dans une table en SQL. Le premier représente l'attribut et il peut contenir des marqueurs NULL pour exprimer l'annulabilité; le deuxième attribut nous permet de savoir le type ou la cause exacte de l'absence.
Clé candidate	La clé candidate est traduite par la contrainte UNIQUE appliquée aux attributs qui représente la clé candidate.
Clé étrangère	La clé étrangère est traduite par une clé étrangère en SQL : FOREIGN KEY .... REFERENCE.
Contrainte générale	Les contraintes sur les valeurs sont exprimées par l'instruction CHECK en SQL.

Voir l'exemple Distribution (la définition des relations Pièce, Fournisseur et Approvisionnement) à l'annexe K.

4.3.3.2 Variable

Il existe trois types de variable : variable de base, table, vue. Nous pouvons avoir plusieurs variables du même type, à condition d'avoir des noms uniques.

Tableau 4-2 : Équivalence de la traduction des variables de relation en SQL

Variable	**SQL**
Base	La variable de base est représentée par une TABLE en SQL tout en respectant les propriétés d'une relation (pas de doublons).
Table	La table est représentée par une TABLE en SQL sans restriction (donc avec la possibilité de comporter des doublons).
Vue	La vue est représentée par une VIEW en SQL.

Voir l'exemple Distribution (la définition des variables P : Pièce, F : Fournisseur, A : Approvisionnement) à l'annexe K.

4.3.3.3 Le corps de la transaction

Le corps de la transaction est traduit par une procédure stockée (PSM) exécutant l'expression relationnelle définie dans le corps de la transaction, cela nous permet de définir en particulier le traitement de la gestion des attributs annulables.

Le marqueur de type de la transaction n'est pas mis en œuvre pour le moment.

4.3.3.4 Les opérations relationnelles et de mise à jour

L'une des propriétés du modèle Discipulus est de s'assurer de ne pas avoir des doublons dans les résultats des requêtes, la variante SELECT DISTINCT est donc utilisée systématiquement.

Tableau 4-3 : Équivalence de la traduction des opérations relationnelles en SQL

	SQL
Sélection / Projection	SELECT DISTINCT ... FROM .... WHERE ...
Groupement	SELECT DISTINCT ... FROM ... GROUP BY ...
Tables virtuelles avec la clause LET	WITH ... AS .... SELECT DISTINCT... FROM ...
Opérations ensemblistes	UNION, INTERSECT, MINUS, ...
Suppression	DELETE ... FROM ... WHERE ...
Insertion	INSERT VALUE ... INTO ...
Mise à jour	UPDATE ... SET ... WHERE ...

Voir l'exemple Distribution (les requêtes de (a) à (n)) à l'annexe K.

4.3.3.5 Le traitement de l'annulabilité

Le traitement des attributs annulables est l'un des objectifs principaux de notre modèle, il repose sur deux mécanismes : les détections des attributs annulés et le déclenchement d'une exception.

Intéressons-nous maintenant aux effets des attributs annulables sur les opérations de l'algèbre relationnelle. Pour simplifier, nous nous limitons aux opérateurs de sélection, de projection, de produit, aux opérations ensemblistes et aux fonctions d'agrégation. Les effets des attributs annulables sur les autres opérateurs peuvent être dérivés de leurs effets sur ceux-ci.

- L'opération de sélection est définie de manière à retourner uniquement les tuples pour lesquels la condition de sélection est vraie. Dans le cas où la condition comporte un attribut annulable effectivement annulé, il faut lever une exception.
- Sauf le produit, tous les opérateurs ensemblistes reposent l'évaluation de l'égalité entre tuples. Dans ce contexte, si l'une des relations comporte ne serait-ce qu'un attribut annulé, une exception doit être produite.
- Si un des attributs ciblés par une fonction d'agrégation est annulé, une exception doit être levée. L'opérateur `card`, lorsqu'appliqué à une relation, fait exception puisqu'il dénombre les tuples sans examiner leur contenu.
- La projection n'est pas affectée.
- Le produit n'est pas affecté.

La vérification de l'annulabilité effective procède par une requête préalable qui déclenchera une exception s'il y a un attribut effectivement annulé. Le traitement de l'exception nécessite la définition d'une exception et les informations associées, le mécanisme requis est le même que pour le mécanisme général d'exception exposé à la section suivante.

Pour la gestion des valeurs absentes, nous devrons disposer des opérateurs spéciaux pour le traitement des NULL, IS NULL et IS NOT NULL pour tester la présence ou l'absence des attributs annulés.

La classe NullException présentée à l'annexe I regroupe les définitions requises par le traitement des exceptions.

Voir l'exemple Distribution (requêtes (a) à (n)) à l'annexe K.

4.3.3.6 Le traitement des exceptions

Le traitement des exceptions permet de gérer les conditions exceptionnelles pendant l'exécution du programme, dans le langage relationnel nous avons trois cas :

- les exceptions induites par une tentative d'évaluation d'un attribut annulé;

- les exceptions des opérateurs algorithmiques appelés par une expression relationnelle;
- les exceptions issues des transactions (non mises en œuvre pour le moment).

Pour la traduction du système des exceptions nous faisons appel au système de gestion des exceptions du SGBD cible, en fait PL/SQL permet de définir des exceptions propres. L'instruction RAISE permet d'interrompre le programme et de transférer le traitement au gestionnaire d'exceptions. La déclaration du nom de l'exception doit se trouver dans la partie déclarative. Pour la gestion des attributs annulables, nous avons défini une nouvelle exception appelée NullException, mais nous pourrions définir plusieurs selon le besoin.

Voir l'exemple Distribution (requêtes (a) à (n)) à l'annexe K.

4.4 Choix techniques

Les choix techniques en matière de développement dépendent des besoins et des objectifs que l'on s'est fixés. Ils dépendent aussi de la complexité et du temps à consacrer pour la formation.

4.4.1 Langage d'implémentation

Pour rendre le programme portable sur plusieurs plateformes, nous avons choisi d'utiliser le langage Java d'abord en raison de son indépendance de toute plateforme qu'elle soit Windows, Unix, Macintosh ou autre. En effet, le compilateur génère un format de fichier instructions en bytecode exécutable sur de nombreux processeurs, à partir du moment où le système d'exécution de Java est présent.

Ensuite, Java possède une grande bibliothèque des pilotes permettant de se connecter au SGBD, cette propriété est très importante pour les phases ultérieures du projet, la programmation de la console en particulier. En fait, il existe des pilotes pour de très nombreux SGBD : Oracle, MySQL, PostgreSQL, SQLServer, DB2, SQLite, Ingres ...

Finalement, nous avons voulu créer un système qui puisse être programmé simplement sans nécessiter un apprentissage important. Notre maîtrise du langage nous a permis de nous

concentrer plus sur problème d'architecture et non pas sur les problèmes techniques. Le choix du langage a aussi été influencé par le choix du système d'écriture de compilateurs (JavaCC).

4.4.2 Outil de génération d'analyseurs

La grammaire est basée essentiellement sur celle d'Eiffel décrite dans la norme ISO 25436 [S14]. Il fallait prévoir de la modifier afin de satisfaire le critère LL(1). L'outil recherché devait permettre non seulement la génération des analyseurs eux-mêmes (et du gestionnaire d'arbres syntaxiques), mais aussi l'élaboration d'une grammaire LL(1).

Trois solutions ont été envisagées :

- JavaCC (Java Compiler Compiler) [S3]
- Cocktail (COmpiler Compiler ToolkIt KArLsruhe) [S4]
- ANTLR (ANother Tool for Language Recognition) [S5]

JavaCC

JavaCC est développé en Java, il est donc facilement exploitable à l'aide d'une interface Java. JavaCC intègre les fonctions d'analyse lexicale et syntaxique. En plus de convertir des spécifications d'une grammaire en programme Java, JavaCC fournit des outils de génération d'arbre, des outils de débogage.

JavaCC intègre également l'outil JJTree [S6] qui permet d'utiliser dans l'analyseur toutes les fonctions de gestion de l'arbre syntaxique. JJTree ajoute dans chaque classe de nœud une méthode jjtAccept() et génère aussi une interface Visitor qui peut être implantée et passée aux nœuds (patron de conception Visitor, voir annexe E).

Toutes les règles de la grammaire du langage sont définies dans le fichier Parser.jjt, l'outil produit une classe Parser.Java, cette classe implémente l'interface ParserConstants définie dans ParserConstants.Java et qui contient les définitions des mots clés de la grammaire.

JavaCC est incroyablement facile d'apprendre, la syntaxe est tout à fait semblable à la syntaxe standard de Java, et les différences sont intuitives.

Cocktail

La boîte à outils Cocktail est un ensemble de générateurs de programmes, les outils peuvent générer des modules du compilateur dans les langages C, C + +, Modula-2, ou Java.

Son architecture est complexe même s'il est relativement petit. Il comporte plusieurs modules ayant de nombreux liens entre eux. C'est cela qui rend sa compréhension un peu complexe pour un utilisateur non initiée à telle approche de construction de compilateurs.

La boîte à outils contient les outils Cocktail principaux suivants :

- Rex : générateur d'analyseurs lexicaux.
- Lark : générateur d'analyseurs syntaxiques LR (1) et LALR(2).
- Ell : générateur d'analyseurs syntaxiques LL (1).
- Ast : générateur des arbres de la syntaxe abstraite.
- Ag : générateur pour les évaluateurs d'attributs.
- Puma : la transformation des arbres attribués en utilisant le filtrage.

ANTLR

ANTLR, est un outil libre (open-source) de construction de compilateurs utilisant une analyse LL(*).Le code source est par défaut émis en Java, mais ANTLR est aussi capable de générer du code pour les langages suivants : C, C++, Python, C#, Objective C.

ANTLR supporte la description EBNF (Extended Backus-Naur Form), le résultat fourni par les analyseurs créés avec ANTLR sont des arbres syntaxiques abstraits (Abstract Syntactic Tree).

ANTLR est très utilisé, pour citer deux exemples, l'analyseur HQL de Hibernate et celui de Groovy sont écrits avec ANTLR.

Choix

Nous n'avions pas d'expérience avec les deux derniers outils et avons craint que la courbe d'apprentissage soit trop importante, compte tenu des délais impartis. Nous avons donc opté pour JavaCC.

À l'expérience, l'utilisation de JavaCC pour des analyseurs dont l'envergure est importante, soulève des problèmes :

- les outils pour faciliter l'élaboration d'une grammaire LL(1) sont réduits;
- la grammaire est difficilement dissociable du code, ce qui rend l'évolution du langage et sa documentation très lourde.

Par contre, le mécanisme d'analyse locale LL(2) pour lever certains conflits à l'aide de la clause LOOKAHEAD s'est avéré simple et efficace.

L'utilisation des autres outils demeure cependant envisageable et pourra être reconsidérée lors d'étapes ultérieures du projet.

4.4.3 Documentation de l'application

Pour cette partie du projet nous avons choisi JavaDoc [S2], un outil qui permet de créer une documentation des applications en format HTML depuis les commentaires insérer dans le code source en Java. En plus de cet outil, nous avons ajouté des commentaires détaillés pour les portions de code qui méritent d'être expliquées.

Cette documentation nous permettra de bien motiver et de bien comprendre le fonctionnement du programme ainsi que les choix techniques et l'architecture, ce qui facilitera son évolution.

L'évolution rapide des fonctionnalités de Doxygen [S15] pourrait en faire un candidat de choix pour la suite du projet.

4.5 État courant du traducteur

Nous avons tenté de mettre en place un traducteur qui nous permet de traduire toutes les instructions Discipulus en SQL en particulier les exigences fonctionnelles et non fonctionnelles. Nous avons pu atteindre une grande partie de nos objectifs. Ci-dessous la liste des fonctionnalités que nous avons pu implémenter ainsi que celles qui restent à implémenter ou à tester.

Ce qui est fait

La stratégie de la traduction pour chacun des éléments est présentée dans la section 5.3 :

- Traduction des instructions de base (condition, boucle, déclaration de variable, fonction, invariant, antécédent, conséquent).
- Traduction d'une classe simple.
- Traduction d'une classe avec héritage simple.
- Programmation d'au moins 80 % des règles de validité définies dans la norme ISO 25436 (voir annexe G).
- Traduction de la définition d'une relation (attributs et contraintes).
- Traduction de la définition de variable.
- Traduction des expressions d'interrogations (sélection, groupement).
- Traduction des expressions de mise à jour (insertion, suppression, mise à jour).

Ce qui reste à faire

- Compléter les tests sur les règles de validité.
- Développer les règles de validité pour l'héritage multiple.
- Ajouter plus de tests sur la détection des attributs annulables dans les expressions Discipulus notamment pour les requêtes imbriquées : nous avons remarqué que dans certains cas la détection des attributs annulables dans les requêtes imbriquées n'est pas traduite correctement.
- Compléter la traduction des éléments du langage non encore traduits, le chapitre 4 présente l'ensemble de ces éléments.
- Compléter le module de la configuration par SGBD : pour le moment la gestion de la configuration selon le SGBD ciblé (Oracle 10g) est incomplète puisque seuls les types de base et les fonctions d'agrégation sont couverts.

Chapitre 5

Essais et résultats

Deux séries d'essais ont été développées pour vérifier et valider notre mise en œuvre : des essais unitaires principalement axés sur la vérification et des essais de système visant plus particulièrement la validité de la traduction. Les traductions produites ont ensuite été exécutées sous *10g Enterprise Edition (Release 10.2.0.1.0)* et leurs résultats analysés pour conduire à nos conclusions.

5.1 Essais

Cette section présente la spécification des essais développés pour évaluer la qualité de la traduction ainsi que la modélisation de la proposition présentée au chapitre 3. Nous commençons par la stratégie des essais unitaires, pour aborder ensuite les essais de système.

5.1.1 Tests unitaires

Les tests unitaires sont utilisés au cours du développement pour s'assurer du bon fonctionnement du code. La stratégie est définie en fonction des instructions de la grammaire. Ce type de tests est basé sur une matrice des cas de tests dont la couverture est déterminée par la grammaire du langage. On peut ainsi facilement accumuler des mesures de couverture et affiner les cas de tests au besoin. L'objectif premier est d'arriver à une couverture totale en utilisant des critères de plus en plus exigeants. Pour les tests unitaires appliqués sur les règles de validité du langage algorithmique, nous avons utilisé (CodeCover) [S11] disponible sous Eclipse, qui permet de donner la couverture du code en fonction des critères suivants, dans l'ordre : Instruction, Branch, Loop et Condition.

Un cas de test représente soit un ensemble d'instructions à tester spécifiant un cas d'utilisation du langage Discipulus, soit un programme construit sur un ensemble de règles de la grammaire de Disciuplus. Avec ces deux approches, il devrait être possible d'avoir une couverture complète, eu égard aux critères choisis. Les mêmes tests peuvent ensuite être utilisés à des fins de validation partielle en exécutant le résultat à l'aide d'Oracle et en comparant avec le résultat attendu.

Les cas de test incluent toutes les règles de la grammaire regroupées en catégories, à savoir : les notes, les types de classes, les types propriété d'une classe (constantes, attribut, fonction, procédure), les différentes instructions. Nous avons utilisé des cas de teste permettant de couvrir toutes les règles de la grammaire, les suites de tests sont disponibles sur le serveur GLOGUS.

5.1.2 Tests de système

L'objectif des tests de système est de s'assurer du bon fonctionnement du traducteur. Et de sa fiabilité autant au point de vue de sa capacité de valider des programmes écrits en Discipulus et la fiabilité de la traduction.

Il n'est pas possible de tester toutes les entrées possibles (c'est-à-dire tous les programmes pouvant être écrits en Discipulus), nous sommes donc amenés à sélectionner des cas de test représentatifs.

Dans un premier temps, nous avons divisé les tests de système en deux parties : les tests pour le langage algorithmique et les tests pour le langage relationnel. Cette séparation a pour objectifs de tester les propriétés de chaque langage avant de faire leur intégration, elle a aussi pour objectifs de simplifier l'exécution et l'analyse des tests.

5.1.2.1 Langage algorithmique

Pour évaluer la traduction du langage algorithmique, nous avons utilisé un exemple proposé par B. Meyer [L13] que nous avons adapté et complété pour nos fins. Nous avons essayé de

développer une série de tests pour couvrir les éléments développés dans le traducteur et cela inclut la définition d'une classe, les attributs, les procédures, les fonctions, les antécédents, les conséquents, les invariants, les exceptions et finalement les instructions conditionnelles et les boucles.

5.1.2.2 Langage relationnel

Les cas de tests effectués ont été séparés en trois ensembles : les opérations de définition de données, les opérations d'interrogation et les opérations de mise à jour.

Nous avons repris trois exemples : Distributeur, Ferrailleurs et le Patinage. Le premier exemple est celui fréquemment utilisé par C. J. Date dans ses ouvrages et les deux derniers sont des exemples réalisés pour des fins pédagogiques par le groupe de recherche Μῆτις. Nous avons tenté à travers ces exemples de couvrir les parties essentielles du langage afin d'en évaluer la définition, la traduction et la gestion des attributs annulables. Nous nous sommes concentrés sur les opérations de création des relations et des variables, les opérations relationnelles d'insertion, de modification et d'interrogations.

Nous avons implémenté ces exemples en Discipulus pour les soumettre ensuite au traducteur et produire le code SQL équivalent. Par la suite, nous exécutons ces résultats dans un SGBD pour évaluer l'exactitude des résultats obtenue. Aux fins de l'évaluation de l'expressivité, nous avons aussi comparé nos programmes avec leurs correspondants rédigés en SQL et Tutorial D.

5.2 Résultats

Nous présentons dans cette partie les résultats des tests de système afin d'en tirer des conclusions pour les prochaines étapes du projet DOMINUS.

5.2.1 Tests unitaires

La partition des tests unitaires est fondée sur les règles de la grammaire du langage. Les critères de couverture choisis sont les suivants : Instruction, Branch, Loop et Condition pour

les règles de validité. Le test est considéré comme exhaustif si tous les éléments sont exécutés.

5.2.1.1 Langage algorithmique

Ci-dessous la couverture des séries de tests.

Tableau 5-1 : Couverture des tests du langage algorithmique.

	Série T00	Série T01
Classe	X	X
Features	X	X
Invariant	X	-
Antécédent	X	X
Conséquent	X	X
Exception	X	-
Héritage simple	X	-
Condition	X	X
Boucle	X	X

Tout d'abord, les tests effectués ne couvrent pas la totalité du langage puisque nous n'avons pas implanté la traduction de tous les éléments du langage dans le cadre de notre expérimentation, en particulier l'héritage multiple et les tuples. Cela est principalement dû au temps qui nous était imparti. Ces éléments sont cependant très importants pour juger la fiabilité du traducteur et ce travail devra être réalisé avant de porter un jugement fiable.

L'annexe G montre en pourcentage la couverture des règles de validité par les tests unitaires. Il reste donc des cas de tests à concevoir afin d'améliorer la couverture des éléments qui ne sont pas à 100 % pour les critères définis.

5.2.1.2 Langage relationnel

Ci-dessous la couverture du langage relationnel par l'implémentation des trois exemples.

Tableau 5-2 : Couverture des tests du langage relationnel.

	Distribution	**Ferrailleurs**	**Patinage**
Définition de relation	X	X	X

Définition de variable	X	X	X
Insertion de données avec valeurs directes	X	X	X
Insertion avec expression relationnelle	X	-	X
Insertion de données avec valeurs absente	X	-	-
Mise à jour des données	X	-	X
Suppression des données	X	X	-
Projection	X	X	X
Sélection	X	X	X
Union	X	X	-
Intersection	X	X	-
Différence	X	-	-
Produit	X	-	-
Fonction d'agrégation	X	X	X

Nous pouvons remarquer que nous n'avons pas couvert toutes les instructions Discipulus notamment celles présentées dans la section 3.2 du chapitre 3. Il reste d'implanter ces instructions et à concevoir des cas de tests afin d'améliorer la couverture de tout le langage. Le taux de couverture obtenu est documenté à l'annexe G.

5.2.2 Tests de système

Nous devons ici tester le traducteur à partir de ses spécifications.

5.2.2.1 Langage algorithmique

Pour évaluer la fiabilité du traducteur, nous avons soumis les traductions des tests algorithmiques au SGBDR Oracle. Les résultats de l'exécution de la traduction des exemples de test compilent sans erreurs et leur exécution calcule les bons résultats.

5.2.2.2 Langage relationnel

Pour évaluer la fiabilité du traducteur à donner des résultats corrects et sans erreur lors de l'exécution nous avons procédé à une évaluation qualitative. L'annexe H montre les résultats de l'exécution obtenus à partir des exemples Distributeur, Ferrailleurs et le Patinage ainsi que la comparaison de ces résultats avec celle de l'implémentation directe en SQL.

Selon ces résultats, nous pouvons conclure dans un premier temps que la traduction des programmes Discipulus est fiable. Nous estimons toutefois que les tests effectués ne sont pas suffisants et que la couverture reste aussi insuffisante.

Ceci a néanmoins permis de détecter des problèmes relatifs au traitement de l'annulabilité et des exceptions que nous analysons ci-après.

Gestion des attributs annulables

Prenons un exemple pour mieux illustrer les résultats concernant la détection des attributs annulables dans une transaction Discipulus. Supposons que nous voulions avoir la liste statuts et des villes des fournisseurs dont le statut est strictement supérieur à 10 et dont le nom du fournisseur est différent de 'El Khoury'. Le code de la transaction en Discipulus est le suivant :

```
with service test_1 do
 with schema Distribution do
  define transaction requete_1 do
       from F
       select statut , ville
       where statut > 10 and nomF /= 'El khoury'
  end
 end
rescue
       inspect exception
            when NullException then
                 error = " OOoops valeur NULL détecté !!!"
       end
end
```

Le résultat de la traduction et le suivant :

```
CREATE OR REPLACE PROCEDURE dpl_requete_1 IS

Attribut_Name VARCHAR2;
Relvar_Name VARCHAR2;
Null_Exception EXCEPTION;
Number_null INTEGER;

BEGIN
```

```
SELECT count(*) INTO Number_null FROM dpl_F WHERE dpl_statut_n > 0 ;
IF Number_null > 0 THEN
 Attribut_Name := 'dpl_statut';
 Relvar_Name := 'dpl_F';
 Raise Null_Exception;
END IF;

EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_1 AS (

SELECT DISTINCT dpl_statut , dpl_ville
FROM dpl_F_t
WHERE dpl_statut > 10 and dpl_nomF <> "El khoury" )';

EXCEPTION
WHEN Null_Exception THEN
 dbms_output.put_line( ' OOoops valeur NULL detecte !!!' );
 dbms_output.put_line( ' detail :' );
 dbms_output.put_line( ' Relvar :' + Relvar_Name);
 dbms_output.put_line( ' Attribut :' + Attribut_Name);

END dpl_requete_1;
```

Si la gestion de la détection de l'annulabilité est correcte et efficace dans les cas simples, nous montrons à l'annexe J que la détection des attributs annulables n'est pas efficace dans tous les cas, car elle ne permet pas toujours de détecter a priori tous les attributs annulables mis en cause dans des requêtes emboîtées. Les tests effectués jusqu'à présent ne nous ont pas permis de cerner le problème.

Les valeurs absentes sont traitées comme des exceptions. Dans Discipulus l'utilisation d'une valeur absente dans un calcul relationnel déclenche une exception qui permettra à l'utilisateur de savoir la cause de l'exception et dont prendre la stratégie du traitement. Pour une description détaillée de ce mécanisme, voir les sections 4.3.2.4 et 4.3.3.4.

Dans l'exemple ci-dessus, nous vérifions si un des attributs annulables associés à la transaction contient un attribut annulé auquel cas une exception est lancée permettant de récupérer les informations rattachées à l'exception. Dans ce cas, il permet de récupérer le nom de la variable, de l'attribut et la raison de l'annulation. De cette façon, l'utilisateur

dispose de l'information nécessaire pour définir une stratégie permettant de résoudre l'exception.

5.3 Premiers constats

Il est encore trop tôt pour tirer des enseignements définitifs de nos premiers essais. Nous pouvons néanmoins dégager quelques éléments qui mériteraient d'être approfondis.

5.3.1 La qualité de la traduction

La traduction constitue la partie la plus importante et la fonction essentielle du traducteur. Selon les résultats obtenus, nous constatons que l'exécution des fichiers SQL produits par le traducteur est essentiellement exacte. Nous n'avons cependant pas pu tester la performance de la traduction SQL obtenue. Cela nécessite un volume de données beaucoup plus grand que celui que nous avons utilisé.

Pour la majorité des composants du langage Discipulus, nous avons pu implémenter des solutions de traduction adéquates et efficaces. Cependant, pour certains autres mécanismes; il nous a été impossible de les implémenter, en raison des limites en temps et en ressource de notre projet.

En général, les résultats que nous avons pu avoir montrent que l'approche utilisée pour faire la traduction des éléments du langage Discipulus est faisable, en revanche les éléments clés de notre proposition que nous n'avons pas pu traiter sont les éléments les plus complexes à traduire.

Recommandations

- Compléter l'implémentation de la traduction des parties manquantes. Plusieurs approches ont été documentées dans ce mémoire, notamment celle de l'héritage multiple. Cela pour pouvoir faire des tests de la qualité de la traduction sur l'ensemble du système qui intègre tous les composants du langage. Il faudra donc réfléchir à la solution la plus adaptée à nos besoins en fonction des critères suivants : la quantité de code engendré et la transportabilité en regard des SGBDR SQL existants.

- Compléter l'évaluation de toutes les règles de validité. Cela est très important et permettra au traducteur de ne traiter que des programmes corrects. L'analyseur doit vérifier les règles de validité d'Eiffel présentées dans la norme ISO 25436 [S14] et celles des parties propres du langage Discipulus.
- Programmer un ensemble minimal de bibliothèques. Dans le cadre de ce mémoire, nous n'avons pas abordé le développement des bibliothèques de base que nous pouvons intégrer au langage, par exemple la gestion des dates et la gestion des chaînes de caractères. Cela évite au programmeur de devoir pour chaque programme réécrire des fonctions déjà connues et communément utilisées.
- Faciliter la validation expérimentale des tests. Pour cela, il faudra tout d'abord implémenter la console basée sur les pilotes JDBC qui permettra au traducteur d'exécuter automatiquement les résultats sur un SGBD.

5.3.2 La modélisation de l'annulabilité des attributs

La gestion des attributs annulables est une fonctionnalité très importante dans notre traducteur. Selon les résultats de tests que nous avons pu avoir pour les trois exemples, nous avons remarqué que la détection des attributs annulables dans des transactions simples sans requêtes imbriquées est effectuée à 100 %, mais lors de l'utilisation des requêtes imbriquées la détection des attributs annulables n'est plus efficace.

Recommandations

- Développer des séries de tests permettant de mieux couvrir la gestion des attributs annulables. Afin de définir une stratégie efficace qui détecte tous les attributs annulables dans une requête.
- Définir une stratégie permettant de détecter les attributs annulables pour des requêtes complexes notamment pour des requêtes imbriquées.

5.3.3 La définition du langage

Tout d'abord, nous pouvons constater que nous avons pu répondre et exprimer les trois exemples (Distribution, Ferrailleur, Patinage) avec les requêtes associées, mais cela semble

insuffisant pour tester l'expressivité du langage. En effet, il faut évaluer la facilité avec laquelle un utilisateur peut exprimer un problème particulier, ainsi que la facilité pour un autre utilisateur de comprendre cette expression.

Recommandations

- Intégrer une couche supérieure d'instructions permettant de faciliter le travail du programmeur comme celle présente en SQL : NOT EXIST, EXIST. Ces instructions peuvent réduire la charge de travail du programmeur et augmenter l'expressivité du langage.
- Effectuer les tests du banc d'essai développé par Aurélie Ottavi, en particulier les tests de performance.

Conclusion

Contributions

Dans ce mémoire, nous avons tout d'abord effectué une analyse détaillée des points sensibles du modèle relationnel en particulier : l'absence des valeurs, les doublons, l'ordre des attributs dans les tuples et l'ordre des tuples dans une relation. Nous avons cité plusieurs propositions pour chacune de ces problématiques et nous avons essayé de les analyser afin de tirer des conclusions pour notre nouvelle solution.

Nous avons alors proposé une nouvelle solution pour améliorer la gestion des valeurs absente. L'approche proposée par SQL engendre des incohérences lors de la manipulation des données et introduit une complexité inutile.

Nous avons montré aussi la nécessité du concept de classe dans le modèle d'où l'adoption d'Eiffel qui intègre un modèle général, rigoureux et éprouvé. Cette conclusion est basée sur l'étude effectuée sur plusieurs langages pour analyser les concepts retenus et les comparer.

L'implémentation de notre proposition à travers un traducteur en SQL nous a permis d'en prouver la faisabilité et d'en évaluer l'apport, les qualités et les déficiences.

Critique du travail

Tout d'abord, l'objectif principal du mémoire est de définir un nouveau langage de définition et de manipulation des bases de données relationnelles. Toutefois, les résultats obtenus montrent que nous n'avons pas pu compléter ce travail et qu'il manque des éléments que nous n'avons pas abordés dans le cadre de ce mémoire autant pour le langage algorithmique que le langage relationnel.

Ensuite, nous avons eu également comme objectif de traduire l'ensemble du langage Discipulus en SQL. Or nous n'avons pas pu compléter la traduction de certains éléments plus particulièrement l'héritage multiple. Ainsi, le critère de validité du traducteur demande à ce que toutes les fonctionnalités du langage soient traduites et testées. Cependant, ces éléments ne sont pas pris en compte dans nos tests système, il faudra probablement reprendre ces tests et ajouter des cas de tests spécifiques.

Nous avons aussi rencontré quelques problèmes avec la stratégie de détection des attributs annulables dans les requêtes complexe. Cependant, cette fonctionnalité est un des éléments clés de notre proposition. Il faudra donc approfondir cet aspect avant de conclure.

Au final, compte tenu du temps et des ressources disponibles, les résultats sont satisfaisants.

Travaux futurs de recherche

Le travail réalisé dans ce mémoire est une contribution à l'élaboration d'un nouveau langage de définition et de manipulation des bases de données. Ce faisant, nous avons mis de côté certains éléments. Il faudra reprendre l'étude de ces éléments afin de compléter la définition du langage. Pour cela, nous divisons les travaux futurs en quatre phases :

- Phase 1 : Compléter la définition du langage : les tuples, le mécanisme de généricité, la gestion des agents et l'intégration des bibliothèques externes.
- Phase 2 : Compléter le traducteur pour intégrer les nouveaux éléments ainsi que la stratégie que nous avons proposée pour l'héritage multiple.
- Phase 3 : Appliquer le blanc d'essai de A. Ottavi pour évaluer d'importants critères caractérisant les principaux groupes d'exigences applicables à un SGBD.
- Phase 4 : Compléter la configuration du traducteur pour supporter les dialectes SQL des SGBD adoptant la norme SQL.

Perspectives

Tout d'abord, nous espérons que cette étude sera poursuivie et que d'autres chercheurs analyseront mieux les solutions proposées en utilisant le traducteur que nous avons

développé. En particulier, la proposition relative au système de typage, la gestion de l'annulabilité, et l'abandon des doublons nous semblent des avenues prometteuses. Cella permettra de développer un modèle de SGBD qui donnera naissance à des produits basés sur des notions théoriques solides tout en évitant les faiblesses des anciennes applications.

Tout en prenant en compte les recommandations présentées toute au long du mémoire, il serait intéressant d'étendre cette étude afin de compléter la définition du modèle, en particulier approfondir la question sur le système de gestion transactionnel et faire l'étude de l'impact de la proposition sur ce système.

Annexe A
Cas d'annulabilité recensés par ANSI/SPARC

Au total, 14 cas d'annulabilité ont été recensés dans le ANSI/SPARC Interim Report [1] :

1. L'information est applicable, mais la valeur n'existe pas encore (ex. : nom de mariage pour une femme)
2. L'information est inapplicable (ex. : nom de jeune fille pour un homme)
3. L'information existe, mais il n'est pas permis (légalement) de l'enregistrer (ex. : religion d'un employé)
4. L'information existe, mais on n'a pas les moyens de trouver la valeur (ex. : dernier taux d'efficacité d'un employé qui à travaillé pour une autre organisation)
5. L'information existe, mais elle n'est pas encore enregistrée (ex. : historique médical d'un nouvel employé)
6. L'information est enregistrée, mais pas encore disponible (ex. : article de blogue qui n'est pas encore publié)
7. On avait enregistré la valeur, mais elle a été supprimée (ex. : un utilisateur ne veut plus que son adresse soit présente dans la base de données)
8. L'information est disponible, mais en changement et donc potentiellement invalide (ex. : solde d'un compte bancaire sur lequel on effectue une opération)
9. L'information est disponible, mais on ne sait pas si elle est fiable (ex. : la note d'examen non encore confirmée)
10. L'information est disponible, mais invalide (ex. : si une erreur s'est produite lors du calcul de la valeur)
11. La classe est sécurisée (verrouillée) (ex. : les informations personnelles des profs ne sont en aucun cas accessible aux élèves).
12. L'objet est sécurisé (ex. : un utilisateur bloque l'accès à ces données sur un réseau social).
13. Une information est sécurisée pour un certain laps de temps (ex. : lecture puis modification du solde d'un compte bancaire).
14. Information calculée à partir d'au moins une information manquante ou incertaine (ex. : l'âge en fonction d'une date de naissance par ailleurs manquante).

Au final, on peut regrouper les 14 cas recensés par ANSI/SPARC en quatre catégories :

A. L'information est absente (regroupe les cas 4, 5, 6, 7, 8, 9, 10...), dans ce cas la, l'utilisation de NULL pourrait être légitime, la question est de savoir comment représenter le NULL pour que cela pose le moins de problèmes possible.

B. L'information n'est pas applicable (représente les cas 1, 2, 3), dans ce cas l'utilisation des NULL est à remettre en question, une bonne modélisation permettrait d'éviter l'utilisation de NULL.
C. L'information n'est pas accessible, car l'utilisateur n'a pas les droits d'accès (représente les cas 12 et 13), dans ce cas, le transactionnel permet d'éviter l'utilisation des NULL.
D. L'information est en train d'être utilisée et on veut la verrouiller (représente le cas 14), dans ce cas, le transactionnel permet d'éviter l'utilisation des NULL.

Tableau 5-3 : Catégorisation des 14 cas d'utilisation du NULL par ANSI/SPARC.

A	L'information est absente	Les cas : 4, 5, 6, 7, 8, 9, 10	dans ce cas la, l'utilisation de NULL pourrait être légitime, la question est de savoir comment représenter le NULL pour que cela pose le moins de problèmes possible
B	L'information n'est pas applicable	les cas : 1, 2, 3	dans ce cas l'utilisation des NULL est à remettre en question, une bonne modélisation permettrait d'éviter l'utilisation de NULL.
C	L'information n'est pas accessible, car l'utilisateur n'a pas les droits d'accès	les cas 12 et 13	dans ce cas, le transactionnel permet d'éviter l'utilisation des NULL. L'information peut être partagée entre plusieurs utilisateurs en même temps avec un contrôle des accès concurrents. L'exécution d'une transaction doit préserver la cohérence de la BD
D	L'information est en train d'être utilisée et on veut la verrouiller	Le cas 14	

Dans la majorité des recherches, cette liste est généralement réduite en deux types de valeur plus générale (inconnue, inapplicable). Toutefois, cette condensation perd beaucoup de sémantique par rapport aux 14 cas.

Annexe B
Requêtes SQL et annulabilité

Jointure avec des valeurs nulles

En SQL les jointures gauches ou droites produisent automatiquement des valeurs NULL pour les valeurs manquantes dans le tableau résultant de la requête contrairement aux jointures normales. Pour illustrer cette différence dans le traitement des valeurs absentes par l'opération de jointure, prenons l'exemple ci-dessous.

Supposons les deux tables suivantes :

Table_1

At1	*At2*
1	Un
NULL	Deux
4	Quatre

Table_2

At3	*At4*
NULL	Two
4	Four

La requête avec une jointure simple :

```
SELECT t1.At1, t1.At2, t1.At3, t2.At4
FROM Table_1 t1 JOIN Table_2 t2.At3
ON t1.At1 = t2.At3;
```

Le résultat est :

At1	At2	At3	At4
4	Quatre	4	Four

Le problème avec les NULL dans les jointures c'est qu'il faut faire attention lors de l'utilisation des colonnes qui peuvent avoir des valeurs NULL dans les jointures. En effet lors d'une opération de comparaison la valeur NULL n'est pas égale à toute autre valeur NULL.

D'où l'importance de la distinction entre les valeurs absentes applicables et non applicables, avec cette distinction nous pouvons par exemple légitimement considérer deux valeurs

absentes non applicables comme égales, contrairement à toute comparaison impliquant une valeur nulle applicable dont le résultat est forcément inconnu. Lorsqu'un résultat est inconnu, la condition est-elle vraie ou fausse ? En SQL, cela varie selon le contexte comme le montrent les exemples suivants.

On distingue trois types de jointure externes :

- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

La requête avec une jointure LEFT JOIN:

```
SELECT t1.At1, t1.At2, t1.At3, t2.At4
FROM Table_1 t1 LEFT JOIN Table_2 t2.At3
ON t1.At1 = t2.At3;
```

Le résultat est :

At1	*At2*	*At3*	*At4*
NULL	Trois	NULL	NULL
1	Un	NULL	NULL
4	Quatre	4	Four

On remarque déjà une anomalie, les colonnes At1 et At3 ne sont pas identiques ! Dans ce cas, on postule donc un NULL applicable et l'égalité doit être interprétée comme « il n'est pas impossible que ces attributs soient égaux ».

La requête avec une jointure RIGHT JOIN:

```
SELECT t1.At1, t1.At2, t1.At3, t2.At4
FROM Table_1 t1 RIGHT JOIN Table_2 t2.At3
ON t1.At1 = t2.At1;
```

Le résultat est :

At1	*At2*	*At3*	*At4*
NULL	NULL	NULL	Two
4	Quatre	4	Four

Les colonnes At1 et At3 sont ici identiques. Par contre, l'interprétation de l'égalité n'est peut-être plus la même : les NULL pourraient être interprétés comme étant inapplicables.

La requête avec une jointure FULL JOIN:

```
SELECT t1.At1, t1.At2, t1.At3, t2.At4
FROM Table_1 t1 FULL JOIN Table_2 t2.At3
ON t1.At1 = t2.At3;
```

Le résultat est :

At1	*At2*	*At3*	*At4*
4	Quatre	4	Four
NULL	Deux	NULL	NULL
1	Un	NULL	NULL
NULL	NULL	NULL	Two

De nouveau, les colonnes At1 et At3 sont distinctes et une seule interprétation de l'égalité demeure « il n'est pas impossible que ces attributs soient égaux ». Les NULL doivent donc être interprétés comme applicables.

La requête avec une jointure utilisant la clause WHERE:

```
SELECT t1.At1, t1.At2, t1.At3, t2.At4
FROM Table_1, Table_2
WHERE t1.At1 = t2.At3;
```

Le résultat est :

At1	*At2*	*At3*	*At4*
4	Quatre	4	Four

Nous retrouvons ici le résultat « normal » et une interprétation cohérente de l'égalité.

L'interprétation de l'égalité est donc différente selon le type de jointure. Si l'interprétation des NULL est « inapplicable », le RIGHT JOIN donne un résultat cohérent dans notre exemple, mais incohérent dans le cas des LEFT et FULL JOIN. Au contraire, si l'interprétation est « applicable », l'interprétation est incohérente quoiqu'uniforme entre les trois types de JOIN externes.

Finalement, les NULL insérés par une jointure externe sont-ils de la même nature que ceux qui préexistaient avant la jointure ? L'interprétation des NULL insérés par les jointures est « inapplicable » (comme celui issu d'une division par zéro). On se retrouve donc devant une contradiction : les JOIN externes traite les NULL comme étant inapplicables, mais insère es NULL inapplicable. SQL propose ainsi une modélisation incohérente, utilisant tantôt une interprétation, tantôt une autre. Finalement, l'absence de distinction entre les deux marqueurs d'annulabilité empêche par ailleurs toute traçabilité du sens même de l'annulabilité.

Les opérateurs mathématiques et la concaténation sur les valeurs nulles

En général, SQL se base sur l'approche de l'ajout de la propriété d'annulabilité aux attributs. Cette approche implique que NULL n'est pas une valeur, mais un marqueur pour représenté la valeur inconnue, l'utilisation les opérateurs mathématiques sur NULL peut conduire à des résultats inattendus

Exemple 1 :

NULL * 0	Résultat est NULL (au lieu de 0)

Normalement quelque soit la valeur inconnue la multiplication par 0 doit résulter 0.

Exemple 2 :

NULL / 0	Résultat est NULL (plutôt qu'une exception)

Lors d'une opération de division de NULL par zéro, le résultat de l'opération retourne NULL au lieu de lancer une exception de division par zéro. Bien que ce comportement n'est pas défini par la norme ISO SQL, certains SGBD traitent cette opération de la même façon (Oracle, PostgreSQL, MySQL et SQL Server).

Les fonctions d'agrégation sur les valeurs nulles

Les fonctions d'agrégation accomplissent un calcul sur plusieurs valeurs et retournent un résultat. Elles sont principalement utilisées avec les commandes GROUP BY et SELECT. Les calculs effectués par ces fonctions consistent à faire sur une colonne, la somme SUM, la

moyenne des valeurs AVG, le décompte des enregistrements ou encore l'extraction de la valeur minimum MIN ou maximum MAX.

En SQL, presque toutes les fonctions d'agrégation effectuent une étape d'élimination des NULL de sorte que les valeurs NULL ne sont pas incluses dans le résultat final du calcul. Cependant, cette élimination implicite de NULL, peuvent avoir un impact et introduire une incohérence sur les résultats fonction d'agrégation.

```
AVG(At1) = (1+4)/2 = 2.5
SUM(At1)/COUNT(*) = (1+4)/3 = 1.66...
```

On remarque que AVG(At1) est différent de SUM(At1)/COUNT(*), une contradiction avec les concepts mathématiques de base. La fonction COUNT(*) est la seule fonction qui n'élimine pas les NULL.

Il faut toutefois noter que AVG(At1) = SUM(At1)/COUNT(At1) livre un résultat exact.

Regroupement et le tri des valeurs nulles

Puisque SQL définit que tout marqueur NULL est distinct d'un autre, une définition spéciale est nécessaire afin de regrouper les valeurs NULL lors de l'exécution de certaines opérations : *"any two values that are equal to one another, or any two Nulls", as "not distinct".* [A27]

Cette définition est utilisée lorsqu'on essaye de grouper ou trier les valeurs NULL avec la clause GROUP BY.

D'autres opérateurs ne font pas la distinction, entre les valeurs NULL, par exemple :

- UNION, INTERSECT et EXCEPT considèrent deux valeurs NULL comme égales lors des opérations d'élimination ou de comparaison.
- DISTINCT lorsqu'il est utilisé dans l'opération de sélection.

La norme SQL ne définit pas explicitement un ordre de tri par défaut pour les valeurs NULL. Dans certains SGBD, les valeurs NULL peuvent être triées avant ou après toutes les valeurs données en utilisant NULLS FIRST, NULLS LAST de la liste ORDER BY.

Polarité du traitement de la valeur nulle

En plus des incohérences démontrées ci-haut, la valeur nulle n'est pas traitée de la même façon par tous les opérateurs de SQL, en fait une valeur nulle doit être considérée comme différente de toutes les autres valeurs nulles, ce qui fait que le résultat de la comparaison de deux valeurs nulles est inconnu.

Tous les SGBD existants respectent cette règle lorsqu'il s'agit d'une clé UNIQUE, sauf dans le cas du mot clé DISTINCT ainsi que l'opérateur UNION qui considèrent les valeurs nulles comme identiques. Cet écart rend la totalité des implantations de SQL irrespectueux à la 4^{e} règle de E. F. Codd :

The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values (for example, "distinct from zero or any other number", in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

Et dans une certaine mesure, de la 3^{e} règle :

All information in the database is to be represented in one and only one way, namely by values in column positions within rows of tables.

Conclusion

Cependant, les informations dont on peut disposer sont souvent incomplètes, imprécises ou émanent de sources hétérogènes de fiabilité incertaine et inégale. Ceci ne doit évidemment pas être un obstacle à leur gestion par des systèmes d'informations avancés, mais le nombre des problèmes produit lors de l'utilisation des nulles et beaucoup plus complexe. Les auteurs du Troisième Manifeste, C. J. Date et H. Darwen, ont proposé que la mise en œuvre de la valeur nulle dans SQL soit être complètement éliminée à cause de ces incohérences et de ces lacunes.

Annexe C
Requêtes SQL et doublons

Cette annexe illustre 12 formulations pour répondre à une question précise et en particulier pour illustrer le problème des doublons dans les résultats SQL. En effet, les douze différentes formulations suivantes, relationnellement équivalentes, produisent neuf résultats différents par rapport à leur degré de duplication.

Requêtes	résultat
SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw') OR P.PNO IN (SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1'))	P1 * 3, P2 * 1.
SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1') OR SP.PNO IN (SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw'))	P1 * 2, P2 * 1.
SELECT P.PNO FROM P, SP WHERE (SP.SNO = SNO('S1') AND SP.PNO = P.PNO) OR P.PNAME = NAME('Screw')	P1 * 9, P2 * 3
SELECT SP.PNO FROM P, SP WHERE (SP.SNO = SNO('S1') AND SP.PNO = P.PNO) OR P.PNAME = NAME('Screw')	P1 * 8, P2 * 4.
SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw') UNION ALL SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1')	P1 * 5, P2 * 2.
SELECT DISTINCT P.PNO FROM P WHERE P.PNAME = NAME('Screw') UNION ALL SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1')	P1 * 3, P2 * 2.
SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw') UNION ALL	P1 * 4, P2 * 2.

SELECT DISTINCT SP.PNO FROM SP WHERE SP.SNO = SNO('S1')	
SELECT DISTINCT P.PNO FROM P WHERE P.PNAME = NAME('Screw') OR P.PNO IN (SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1'))	P1 * 1, P2 * 1.
SELECT DISTINCT P.PNO FROM P WHERE P.PNAME = NAME('Screw') OR P.PNO IN (SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1'))	P1 * 1, P2 * 1.
SELECT DISTINCT SP.PNO FROM SP WHERE SP.SNO = SNO('S1') OR SP.PNO IN (SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw'))	P1 * 1, P2 * 1
SELECT P.PNO FROM P GROUP BY P.PNO, P.PNAME HAVING P.PNAME = NAME('Screw') OR P.PNO IN (SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1'))	P1 * 1, P2 * 1.
SELECT P.PNO FROM P, SP GROUP BY P.PNO, P.PNAME, SP.SNO, SP.PNO HAVING (SP.SNO = SNO('S1') AND SP.PNO = P.PNO) OR P.PNAME = NAME('Screw')	P1 * 2, P2 * 2.
SELECT P.PNO FROM P, SP GROUP BY P.PNO, P.PNAME, SP.SNO, SP.PNO HAVING (SP.SNO = SNO('S1') AND SP.PNO = P.PNO) OR P.PNAME = NAME('Screw')	P1 * 2, P2 * 2.
SELECT P.PNO FROM P WHERE P.PNAME = NAME('Screw') UNION SELECT SP.PNO FROM SP WHERE SP.SNO = SNO('S1')	P1 * 1, P2 * 1.

Annexe D
Exemples d'incohérences en SQL

Cette annexe présente les incohérences du traitement des valeurs nulles dans les requêtes SQL. Nous donnons des exemples en SQL et nous essayons d'expliquer la cause de l'incohérence.

SELECT * FROM "Fournisseur" LEFT OUTER JOIN "Livraison" ON "Fournisseur"."SNO" = "Livraison"."SNO";	Jointure : la jointure produit automatiquement des valeurs NULL pour les valeurs manquantes dans le tableau résultant de la requête.
SELECT "SNO", COUNT("CITY") FROM "Fournisseur" GROUP BY "SNO"	Group by : Parce que SQL : 2003 définit tout marqueure NULL comme étant inégal à un autre, une définition spéciale a été nécessaire afin de regrouper les valeurs NULL.
SELECT COUNT("CITY") AS "NBCITY" FROM "Fournisseur"; SELECT "SNO",SUM("QTY")/COUNT("SNO") AS "MQTY" FROM "Livraison" GROUP BY "SNO" ; SELECT "SNO", AVG("QTY") AS "MQTY" FROM "Livraison" GROUP BY "SNO" ;	Fonction d'agrégation : les valeurs NULL ne sont pas incluses dans le résultat final du calcul. Cette élimination implicite des NULL, peut avoir un impact sur les résultats des fonctions d'agrégation. (Non équivalence AVG() avec SUM()/COUNT())
NULL / 0 NULL * 40 SELECT "SNAME"||':'||"CITY" AS "SNAMECITY" FROM "Fournisseur"	Les Operateurs mathématiques : puisque NULL n'est pas une valeur, une opération mathématique sur NULL, retourne toujours NULL. La concaténation : même chose pour les opérations de concaténation des chaînes de caractère.

SELECT CASE "CITY" WHEN NULL THEN 'la valeur est Nulle' WHEN 'Paris' THEN 'a Paris' WHEN 'London' THEN 'a London' END FROM "Fournisseur";	L'opérateur CASE peut être évalué comme une série de conditions de comparaison d'égalité, une simple expression CASE ne peut pas vérifier l'existence de la valeur NULL directement. Une condition avec NULL dans une expression CASE simple entraîne toujours UNKNOWN, par conséquent la valeur ('la valeur est Nulle') ne peut jamais être retournée.
CREATE TABLE DELIVRY (..... QL INTEGER, CONSTRAINT CK_QL CHECK (QL < 0 AND QL = 0 AND QL > 0));	Opérateur Check : Une contrainte de vérification fonctionne différemment d'une clause WHERE. En effet, la clause doit être TRUE pour qu'une ligne soit sélectionnée. Par contre, la contrainte de vérification CHECK doit être évaluée à FALSE. Cela signifie qu'une contrainte de vérification sera acceptée si le résultat de la vérification est vrai ou inconnu. Telle que formulée, la contrainte CK_QL interdit toute insertion... sauf si QL est NULL !
SELECT * FROM "Fournisseur" WHERE "STATUS" = 23; SELECT * FROM "Fournisseur" WHERE "STATUS" <> 23;	Une ligne dont STATUS est NULL ne se trouve dans aucun des deux SELECT.

Annexe E
Quelques modèles d'annulabilité

La présente annexe quelques propositions historiques majeures relativement à la modélisation de l'annulabilité, soit :

- Codd I et Codd II [L9, A5]
- Zongmin [A10]
- Vassiliou [A7]
- C. J. Date et H. Darwen [A11, S3]

Nous présenterons ensuite une nouvelle approche envisagée dans le cadre du projet DOMINUS.

Propositions de E. F. Codd

Dans son modèle initial, E. F. Codd [L9] avait introduit une seule valeur nulle représentée par ω. L'extension à l'algèbre relationnelle qui en découle est fondée sur une logique tri-valuées (vrai, faux, inconnu), la valeur inconnu étant obtenue par la compariaons d'une quelconque valeur avec ω dont les tables de réviré sont données ci-après.

Tableau 5-4 : Table de vérité de l'opérateur ET (Codd I)

ET	***V***	***F***	***X***
V	V	F	X
F	F	F	X
X	X	X	X

Tableau 5-5 : Table de vérité de l'opérateur OU (Codd I)

OU	***V***	***F***	***X***
V	V	V	V
F	V	F	X

X	V	X	X

Tableau 5-6 : Table de vérité de l'opérateur NOT (Codd I)

	NOT
V	F
F	V
X	X

Dans sa deuxième version [L9, A5], E. F. Codd montre que la mise en œuvre de la valeur nulle dans SQL était erronée et il suggère d'introduire deux marqueurs distincts : A-VALEUR pour les attributs applicables et I-VALEUR pour les attributs inapplicables qui correspondent respectivement aux catégories des situations applicable (A) et inapplicable (B) de la catégorisation ANSI (voir Annexe A). Les catégories C et D ne sont pas modélisées pusiqu'elles découlent de règles de contrôle d'accès et non de lamodélistion. La recommandation de E. F. Codd, nécessite la mise en œuvre d'une logique à quatre valeurs.

Tableau 5-7 : Table de vérité de l'opérateur ET (Codd II).

ET	***V***	***F***	***A***	***I***
V	V	F	A	I
F	F	F	F	I
A	A	F	A	I
I	I	I	I	I

Tableau 5-8 : Table de vérité de l'opérateur OU (Codd II)

OU	***V***	***F***	***A***	***I***
V	V	V	V	V
F	V	F	A	I
A	V	A	A	I
I	V	I	I	I

Tableau 5-9 : Table de vérité de l'opérateur NOT (Codd II)

	NOT
V	F
F	V
A	A
I	I

D'autres ont suggéré d'ajouter plus de marqueurs à la recommandation de E. F. Codd pour indiquer la raison de l'absence; ces propositions entraînent l'augmentation de la complexité du système, le cas de la proposition de Zongmin.

Proposition de Zongmin

Pour Zongmin [A10], la logique 4V bine que plus précise que la logique 3V demeure incomplète pour couvrir toute la sémantique de la valeur NULL, il propose donc une logique à 7V. Examinons l'expression X=1 avec la logique à 7V.

Tableau 5-10 : La logique 7V appliquée sur l'expression X=1

	Existence	Égalité	Entier (compatibilité)	La vérité de l'expression X = 1
T	oui	oui	oui	TRUE
F	oui	non	oui	FALSE
I	non	-	-	INAPPLICABLE
M1	oui	IN	oui	TRUE or FALSE (M1)
M2	IN	IN	oui	INAPPLICABLE or TRUE or FALSE (M2)
M3	IN	IN	oui	(INAPPLICABLE or TRUE) or not FALSE
M4	IN	IN	oui	(INAPPLICABLE or FALSE) or not TRUE

Ces 7 valeurs induisent donc des tables de 49 résultats pour définir les opérateurs AND et OR. C'est très complexe et difficile à mettre en œuvre. Zongmin propose donc de réduire les 7 valeurs à 5 de la façon suivante : on retient les quatre valeurs suivantes I (INAPPLICABLE), F (FALSE), M1 (TRUE or FALSE) et T (TRUE) et on introduit une

nouvelle valeur (INAPPLICABLE or TRUE or FALSE), exprimée par M2. Les cas M3 et M4 sont donc couverts par M2 (les trois dernières lignes du tableau précédent sont donc compressées en une seule).

Tableau 5-11 : Tables de vérité de la logique 5V de Zongmin (opérateur OU)

OU	***I***	***M2***	***F***	***M1***	***T***
I	I	M2	F	M1	T
M2	M2	M2	F	M1	T
F	F	F	F	M1	T
M1	M1	M1	M1	M1	T
T	T	T	T	T	T

Tableau 5-12 : Tables de vérité de la logique 5V de Zongmin (opérateur ET)

ET	***I***	***M2***	***F***	***M1***	***T***
I	I	I	I	I	I
M2	I	M2	M2	M2	M2
F	I	M2	F	F	F
M1	I	M2	F	M1	M1
T	I	M2	F	M1	T

Tableau 5-13 : Tables de vérité de la logique 5V de Zongmin (opérateur NOT)

	NOT
I	M1
M2	M1
F	T
M1	M2
T	F

Proposition de Vassiliou

La proposition de Vassiliou est fondée sur la sémantique dénotationnelle et elle est considérée comme la seule proposition qui représente l'absence d'une valeur comme une propriété du type et d'avoir un type annulable et un type non annulable. Cette approche est différente de celle qui représente l'annualité par une propriété de l'attribut, elle est très complexe dans le fait qu'il faut définir des opérations primitif de chaque type pour traiter les valeurs nulles.

Dans son article Vassiliou [A7], traite une autre interprétation de la valeur nulle : « inconsistant » (attribut non applicable). Puisque cette proposition est fondée sur la sémantique dénotationnelle [L18], l'évaluation d'une requête est considérée comme une fonction continue entre les types des données (domaines). Chaque occurrence de la valeur nulle admet une information indéterminée ou surdéterminée représentée respectivement par l'élément minimal ou maximal de la structure (ensemble ou le domaine), dans cette structure chaque élément admet une borne supérieure et une borne inférieure.

L'approche est rejetée notamment à cause de sa complexité malgré qu'il permet un traitement formel dans le cadre de la sémantique dénotationnelle.

Proposition de C. J. Date et H. Darwen

Dans le troisième manifeste [L10, A11, S3] de (1995) C. J. Date et H. Darwen prennent fermement position pour l'élimination des NULL, cette position été maintenue dans chacune des révisions subséquentes du troisième manifeste, dont la plus récente date du 30 octobre 2011. L'objectif des manifestes est de regrouper les principes des langages de programmation orientée objet et les systèmes de gestion de base de données. Les deux chercheurs proposent de maintenir le modèle relationnel des bases de données et de soutenir des objets comme des types définis par l'utilisateur. Dans leur mise en contexte ils essayent d'expliquer l'insuffisance des systèmes de gestion des bases de données relationnelles existantes et montrer que langage SQL que la plupart de ces systèmes utilisent est complexe et incohérent. Le manifeste décrit une spécification des caractéristiques souhaitables d'un langage de base de données appelée D.

C. J. Date et H. Darwen, [L10, A11, S3] montrent que la mise en œuvre de la valeur nulle dans SQL est fondamentalement ne respecte pas les règles du modèle relationnel et en concluant que le concept devrait être complètement éliminé. Les deux auteurs ont justifié leur proposition en montrant les incohérences et les lacunes dans la mise en œuvre de la valeur nulle en particulier dans les fonctions d'agrégation.

Exemple :

Supposons qu'on a une table avec des valeurs manquantes, la solution de C. J. Date se situe au niveau de la modélisation via une décomposition verticale et horizontale.

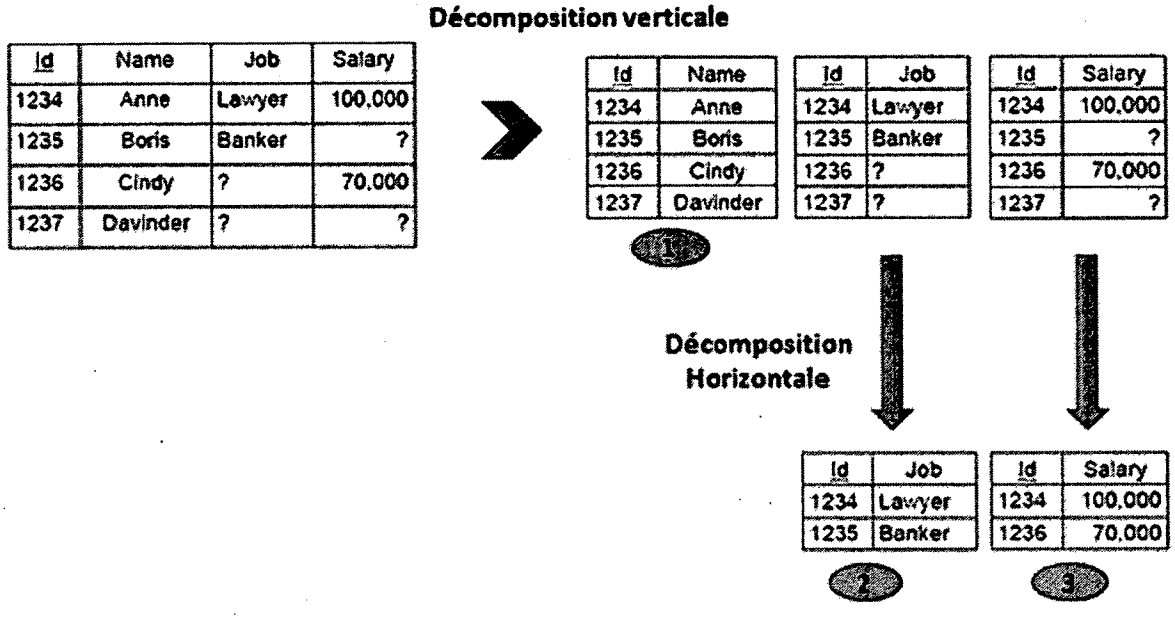

Figure 5-1 : Modélisation de l'absence d'une valeur : proposition C. J. Date et H. Darwen

- Décomposition verticale : normaliser la relation sans prendre en compte la 4NF et la norme de BC.
- Décomposition horizontale : éliminer les valeurs manquantes des relations et isoler les attributs qui peuvent avoir des valeurs manquantes.

Finalement afin de conserver la pertinence et éviter les redondances, de nouvelles contraintes doivent être définies, C. J. Date a défini un nombre de contraintes pour faciliter la tâche.

Cette solution peut être dangereuse et coûteuse. La décomposition réalisée dans l'exemple de C. J. Date rend les requêtes beaucoup plus complexes, par exemple si on veut retrouver la table initiale nous devrons écrire une requête assez complexe.

Propostion Dominus-1

Dans la classification Dominus-1 les situations menant à l'indisponibilité ou l'inapplicabilité d'une valeur sont classées dans un premier temps en 6 catégories :

- Non applicable : la valeur n'est pas applicable par exemple la date de décès pour un vivant.
- Non disponible : la valeur est existe, mais non disponible pour l'instant.
- Non valide : la valeur est non valide parce qu'elle ne respecte pas certaines contraintes déjà définies.
- Non fiable : la valeur est non fiable lorsqu'elle n'est pas éprouvée.
- Non accessible : dans cette catégorie on peut faire la distinction entre deux cas, Objet non accessible (sécurisé) et classe non accessible (verrouillé).
- Calculé : la valeur est calculée à partir d'une valeur d'un attribut qui est absent.

Chacune des catégories est divisée en deux classes, une classe des cas ou l'absence des valeurs est considérée comme durable et une autre classe des cas ou l'absence des valeurs est considérée comme temporaire. Finalement, une sous-classification basée sur le facteur du temps est appliquée sur la deuxième classe afin de déterminer relativement la disponibilité de la valeur dans le futur. La classification est présentée au tableau 28.

L'analyse de cette classification montre que tous les marqueurs temporaires sont internes au système transactionnel et que les catégories non fiable et non accessible doivent pouvoir être restreintes au seul traitement transactionnel interne (comme pour les marqueurs temporels). Il est alors possible de simplifier la classification, voir tableau 29.

Tableau 5-14 : Classification des cas d'annulabilité Dominus-1

	AINSI	Cas	Temporaire	Modélisation	Annulabilité	Hybride	Transaction
Essentiel							
Non applicable				x			
durablement	2			x			
temporairement				x			
ps (passé simple)				x			
fa (futur antérieur)	1			x			
fs (futur simple)	3			x			
Circonstanciel							
non disponible							
durablement	4	ND			x		
temporairement						x	
ps	7	ND.ps	x			x	
fa	6	ND.fa	x			x	
fs	5	ND.fs	x			x	
invalide							
durablement	10	IN			x		
temporairement	8					x	
ps		IN.ps	x			x	
fa		IN.fa	x			x	
fs		IN.fs	x			x	
non fiable	9						x
durablement		NF					x
temporairement							x
ps		NF.ps	x				x
fa		NF.fa	x				x
fs		NF.fs	x				x
non accessible (classe)	11						x
durablement		CNA					x
temporairement							x
ps		CNA.ps	x				x
fa		CNA.fa	x				x
fs		CNA.fs	x				x
non accessible (obj)							x
durablement	12	ONA					x
temporairement	13						x
ps		ONA.ps	x				x
fa		ONA.fa	x				x
fs		ONA.fs	x				x
claculé	14						?
durablement		CA	ssi au moins un (non légitime) durable				?
temporairement							x
ps		CA.ps	x	tes données OK ou ps			x
fa		CA.fa	x	s données OK ou ps ou fa			x
fs		CA.fs	x	onnées OK ou ps ou fa ou fp			x

Tableau 5-15 : Catégorisation finale Dominus-1 des 14 cas d'AINSI/SPARC

Non applicable	Une bonne modélisation permettrait d'éviter la représentation de l'absence de la valeur.
Non disponible	Dans ce cas, l'utilisation d'un marqueur pourrait être légitime, la question est de savoir comment représenter pour que cela pose le moins de problèmes possible.
Non valide	
Non fiable	La non-fiabilité et l'inaccessibilité des données pourront être traitées au niveau transactionnel.
Non accessible	
Calculé	La bonne gestion des autres cas permet de traiter cette catégorie correctement.

Cette classification sert de base à la définition d'une logique à quatre valeurs présentée ci-après. Une comparaison peut correspondre à l'une des quatre valeurs : Vrai (V), faux (F), non disponible (ND) et non valide (NV).

Tableau 5-16 : Table de vérité du ET (logique à 4 valeurs)

ET	***V***	***F***	***ND***	***NV***
V	V	F	ND	NV
F	F	F	F	F
ND	ND	F	ND	NV
NV	NV	F	NV	NV

Tableau 5-17 : Table de vérité du OU (logique à 4 valeurs)

OU	***V***	***F***	***ND***	***NV***
V	V	V	V	V
F	V	F	ND	NV
ND	V	ND	ND	ND
NV	V	NV	ND	NV

Tableau 5-18 : Table de vérité du NON (logique à 4 valeurs)

	NOT
V	F
F	V
ND	ND
NV	NV

La proposition Dominus-1 intègre la signification de chacune des situations. Cette signification pourrait être une partie de la solution, en effet l'utilisation de classification Dominus-1 (figure 4) permet de garder une trace sur la cause de l'absence de la valeur, et par conséquent la possibilité de définir une stratégie précise qui permet de gérer l'absence de la valeur. Toutefois, si la cause de l'absence est résolue, l'utilisateur peut définir l'attribut. Il peut également définir une stratégie d'attribution de valeur par défaut.

Exemple récapitulatif

Par exemple, supposons une relation Naissance (Nom, Prénom, Date_Naissance, Âge), l'attribut Date_Naissance est annulable et l'attribut Âge est un attribut annulable et calculé avec l'expression : (Âge = Date_systeme – Date_Naissance) si l'attribut Date_naissance est NULL, quelque soit la raison, l'attribut Âge est nécessairement NULL avec une trace de la cause de l'absence (calculé à partir d'un attribut NULL).

Tableau 5-19 : Exemple de modélisation Dominus-1 pour les NULL

Nom	**Prénom**	**Date_Naissance**	**Âge**
Ottavi	Aurélie	1986-01-01	24
Abouaddaoui	Zouhir	NULL(ND)	NULL(NV)

Si la Date_Naissance de Abouaddaoui Zouhir est rendue disponible, l'attribut Âge le devient aussi.

Annexe F
Structure de la table de symboles

Les diagrammes suivants représentent la structure de la table des symboles implantée dans l'application de la traduction ainsi que le patron de conception Visitor utilisé pour parcourir l'arbre syntaxique généré par JavaCC.

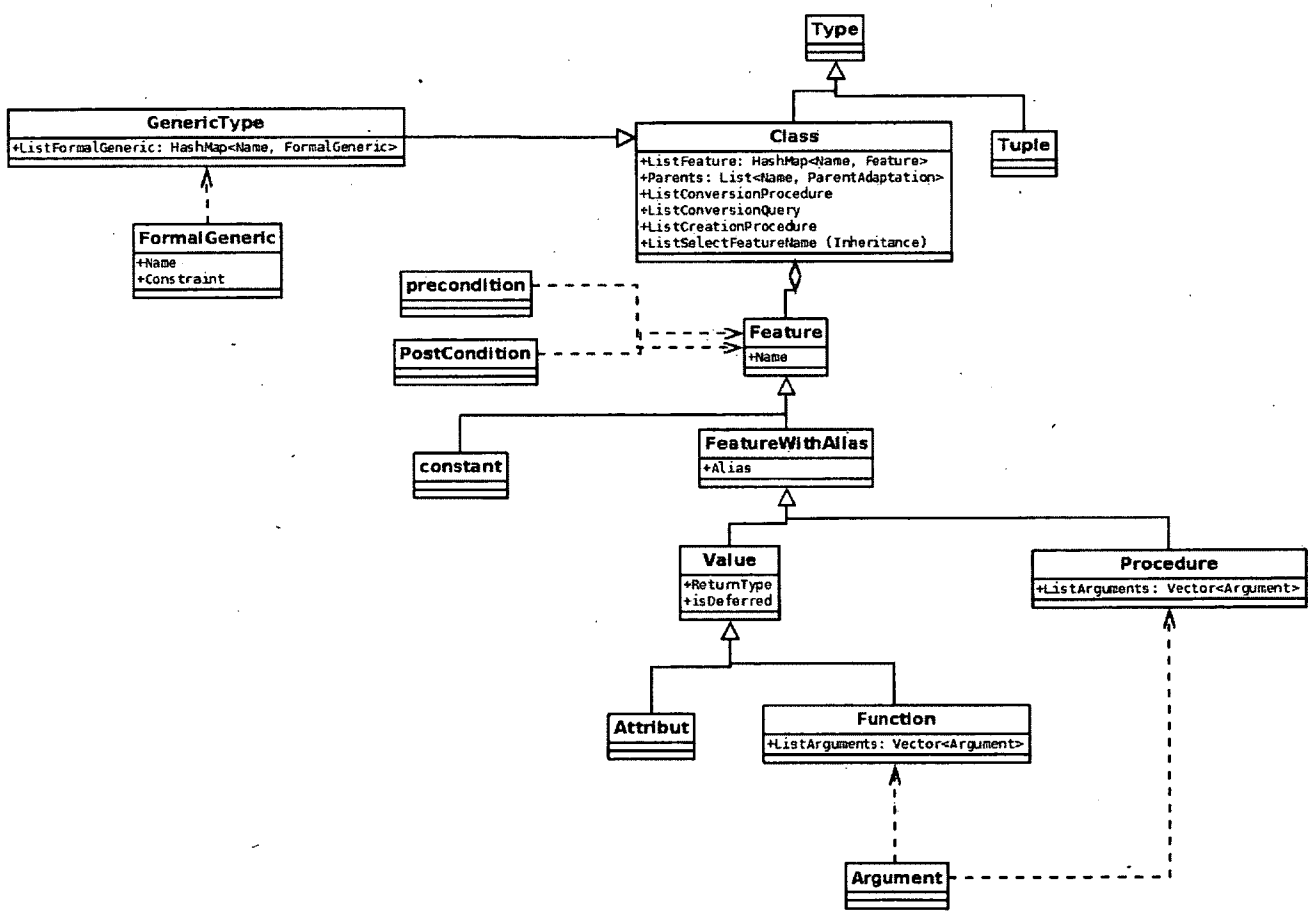

Figure 5-2 : Diagramme UML présentant la structure de la classe.

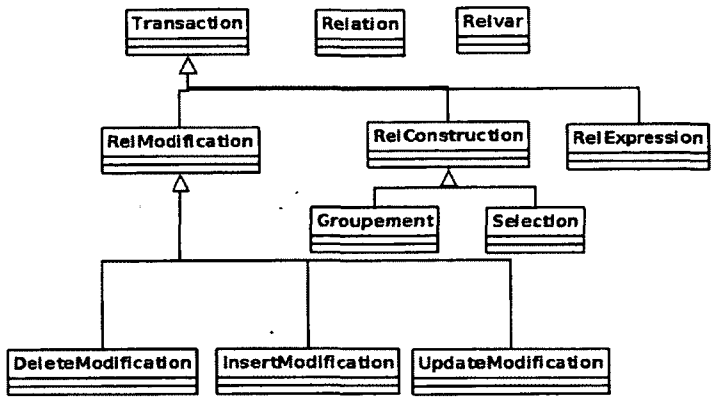

Figure 5-3 : Diagramme UML présentant la structure de la partie relationnel.

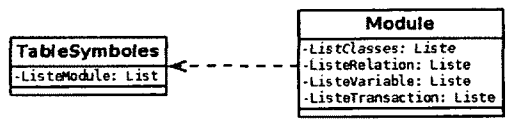

Figure 5-4 : Diagramme UML présentant la structure de la table des symboles.

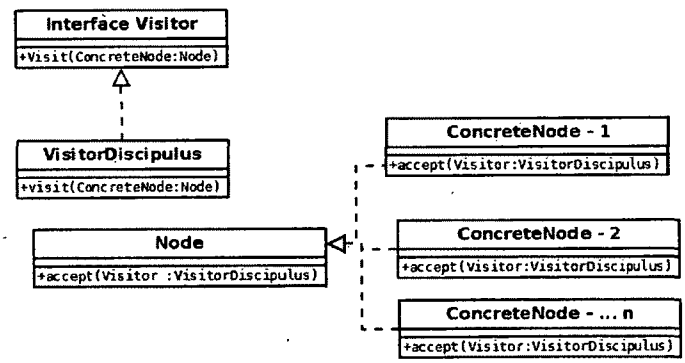

Figure 5-5 : Patron de conception Visitor

Annexe G
Résultats des tests unitaires

Le tableau suivant représente les résultats de couverture des tests unitaire. Les critères choisis sont : Instruction, Branch, Loop et Condition. L'outil utilisé est CodeCover sous Eclipse.

Règles de validité	priorité	réalisée	Instruction	Branch	Loop	Condition
Anchored Type rule	1	oui	100	100		100
Argument rule	1	oui	100			
Assigner Call rule	1	oui	100	100		100
Assigner Command rule	1	oui	57	75	58	78
Assignment rule	1	oui	100	100		100
Boolean Expression rule	1	non				
Bracket Expression rule	1	non				
Call Sharing rule	1	oui	100			
Class ANY rule	1	oui	100			
Class Header Rule	1	oui	67	75	34	75
Class Name Rule	1	oui	100	50	50	50
Class Type rule	1	oui	100	100		100
Class-Level Call rule	1	oui	100	50		50
Conversion Asymmetry principle	1	oui	67	50		50
Conversion principle	1	oui	50	50		50
Conversion Procedure rule	1	oui	41	14	16	18
Conversion Query rule	1	oui	41	14	16	18
Creation Clause rule	1	oui	50	50		50
Creation Expression rule	1	non				
Creation Instruction properties	1	oui	67	50		50
Creation Instruction rule	1	oui	80	50		25
Creation Precondition rule	1	oui	100			
Direct conformance: formal generic	3	non				
Direct conformance: reference types	1	non				
Direct conformance : tuple types	1	oui	60	25	33	25
Entity Declaration rule	1	oui	100			
Entity rule	1	oui	40	50		50

Export List rule	1	oui	100			
Export rule	1	oui	100	50		50
Expression convertibility	1.5	non				
Feature Body Rule	1	oui	100			
Feature Declaration rule	1	oui	100			
Feature Identifier principle	1	oui	100			
Formal Argument rule	1	oui	100			
General conformance	1	oui	100	50		50
Generic Constraint rule	3	oui	33	25	33	25
Generic Derivation rule	3	non				
Interval rule	2	non				
Join rule	3	non				
Local Variable rule	1	oui	25	16	40	12
Manifest Constant rule	1	oui	100			
Multi-branch rule	2	oui	100			
Name Clash rule	1	oui	80		44	50
Non-Object Call rule	1	oui	50	20		
Object Test rule	1	oui	50	20		
Old Expression rule	1	oui	100			
Only Clause rule	1	oui	100			
Operator Expression rule	2	non				
Parent rule	1	oui	33		44	16
Precondition Export rule	1	oui	100			
Precondition-free routine	1	oui	40	0	34	0
Redeclaration rule	2	oui	20			
Redefine Subclause rule	1	oui	40			
Rename Clause rule	1	oui	100			
Repeated Inheritance Consistency constraint	1	oui	100	0	0	
Rescue clause rule	1	oui	100			
Select Subclause rule	1	oui	20	0	11	0
Signature conformance	1	oui	40	0	8.3	0
Undefine Subclause rule	1	oui	40	0	58	10
Universal Conformance principle	1	oui	100			
Variable Initialization rule	1	oui	100			
Variable rule	1	oui	60	50		50
Variant Expression rule	1	oui	100	100	55	100

Annexe H
Résultats des tests de système

Ci-dessous le tableau de montre l'exactitude de nos résultats pour les trois exemples : Distribution, Ferrailleurs et Patinage. En fait, nous avons essayé de faire une analyse qualitative de nos résultats de traduction en les exécutant sur le SGBD Oracle 01g et par la suite analyser les résultats pour vérifier s'ils sont corrects ou non.

Distribution

Tableau 5-20 : Exactitude de la traduction de l'exemple Distribution

requêtes	exactitude
Création	
création de la table/relation : Fournisseur	OUI
création de la table/relation : Pièce	OUI
création de la table/relation : Approvisionnement	OUI
Insertion	
données pour la table/relation : Fournisseur	OUI
données pour la table/relation : Pièce	OUI
données pour la table/relation : Approvisionnement	OUI
requêtes	
a : sélectionner le statut et la ville des fournisseurs qui ont un statut supérieur à 10 et le nom différent de (El Khoury)	OUI
b : le nombre des noms des pièces distinctes livrées	OUI
c : sélectionner le fournisseur, le nom de la pièce et la quantité livrée si le poids de la pièce estsupérieur à 15	OUI
d : le nom des pièces et la quantité livrée de cette pièce par tous les fournisseurs	OUI
e : la quantité livrée de toutes les pièces pour chaque fournisseur	OUI
f : mettre à jour le statut du fournisseur si la ville fait partie des villes des pièces	OUI
g : Le plus grand statut d'un fournisseur	OUI
h : Les pièces fabriquées dans au moins deux villes	OUI
i : Les fournisseurs ayant fourni pour plus de 5 000 kg de pièces.	OUI

J : Les pièces distribuées par un seul fournisseur et produites dans une seule ville.	NON
k : Le nombre de pièces différentes distribuées par chaque fournisseur.	OUI
l : La liste des noms des fournisseurs ailleurs que Londres.	OUI
m : La quantité de vis disponibles.	OUI
n : les fournisseurs qui n'ont pas d'approvisionnement	OUI
o : les pièces qui ont un poids supérieur à toute pièce de couleur bleue.	OUI
p : Les noms des fournisseurs qui n'approvisionnent pas au moins une pièce.	OUI
q, r, s : Les noms des fournisseurs dont le statut est supérieur à dix pour cent de la quantité totale des pièces qu'ils approvisionnent;	OUI

Ferrailleur

Tableau 5-21 : Exactitude de la traduction de l'exemple Ferrailleur

requêtes	exactitude
Création	
création de la table/relation : Metal	OUI
création de la table/relation : TypeProd	OUI
création de la table/relation : TypeOrg	OUI
création de la table/relation : ProduitM	OUI
création de la table/relation : Organisation	OUI
création de la table/relation : Achat	OUI
création de la table/relation : Vol	OUI
Insertion	
données pour la table/relation : Metal	OUI
données pour la table/relation : TypeProd	OUI
données pour la table/relation : TypeOrg	OUI
données pour la table/relation : ProduitM	OUI
données pour la table/relation : Organisation	OUI
données pour la table/relation : Achat	OUI
données pour la table/relation : Vol	OUI
requêtes	
a : La liste des produits en cuivre dont le prix est inférieur au prix du produit en étain le moins coûteux	OUI
b : Les organisations victimes de vol	OUI
c : Les organisations à la fois victimes et coupables de vol	OUI
d 1 : création de vue	OUI
d 2 : Le plus important vendeur d'étain	OUI
d 3 : suppression	OUI

e : Augmenter le prix des produits de 30 %	OUI
f : Définir un double de tout produit dont le métal est 'Cuivre' en suffixant son nom par '-Étain', en changeant son métal par 'Étain' et en augmentant son prix de 35 %.	OUI
g : Retirer toutes les organisations dont le type est 'extracteur'	OUI

Patinage

Tableau 5-22 : Exactitude de la traduction de l'exemple Patinage

requêtes	exactitude
Création	
création de la table/relation : Gala	OUI
création de la table/relation : Compétition	OUI
création de la table/relation : Adresse	OUI
création de la table/relation : Équipe	OUI
création de la table/relation : Personne	OUI
création de la table/relation : Participant	OUI
création de la table/relation : Juge	OUI
création de la table/relation : JuryMembre	OUI
création de la table/relation : Président	OUI
création de la table/relation : Lieu	OUI
création de la table/relation : seqComp	OUI
création de la table/relation : JugeSequence	OUI
création de la table/relation : Figure	OUI
création de la table/relation : Exercice	OUI
création de la table/relation : Inscription	OUI
création de la table/relation : Participe	OUI
création de la table/relation : SeqPar	OUI
création de la table/relation : Pointage	OUI
création de la table/relation : NoComptabilise	OUI
Insertion	
données pour la table/relation : Gala	OUI
données pour la table/relation : Compétition	OUI
données pour la table/relation : Adresse	OUI
données pour la table/relation : Équipe	OUI
données pour la table/relation : Personne	OUI
données pour la table/relation : Participant	OUI
données pour la table/relation : Juge	OUI
données pour la table/relation : JuryMembre	OUI
données pour la table/relation : President	OUI
données pour la table/relation : Lieu	OUI
données pour la table/relation : seqComp	OUI
données pour la table/relation : JugeSequence	OUI

données pour la table/relation : Figure	OUI
données pour la table/relation : Exercice	OUI
données pour la table/relation : Inscription	OUI
données pour la table/relation : Participe	OUI
données pour la table/relation : SeqPar	OUI
données pour la table/relation : Pointage	OUI
données pour la table/relation : NoComptabilise	OUI
requêtes	
a : Donner la liste des membres d'une équipe	OUI
b : Donner la liste des membres d'un jury	OUI
c : Donner la liste des figures qui font partie de plus d'un gala au cours de l'année. (F2, F6)	OUI
d : Donner la liste des figures qui font partie de tous les galas au cours de l'année.	OUI
e : Donner la liste des figures qui ne font pas partie d'aucun gala au cours de l'année. (F3, F5)	OUI
f : Donner la liste de tous les participants au cours de l'année.	OUI
g : Donner la liste des participants ayant participé à tous les galas.	OUI
h : Déterminer le participant ayant obtenu la meilleure note pour une figure donnée au cours de l'année.	OUI
i : Déterminer la moyenne annuelle par figure d'un participant.	OUI
j : Déterminer le rendement annuel moyen par compétition d'un participant (le rendement est défini comme le rapport des points obtenus lors d'une compétition sur le maximum possible lors de cette compétition).	OUI
k : Déterminer le juge ayant coté en moyenne le plus bas pour une figure donnée au cours de l'année	OUI
l : Donner la liste des participants ayant abandonné une compétition au cours de l'année.	OUI

Annexe I
Présentation de NullException

Un des objectifs du langage Discipulus est la fiabilité. En particulier, chaque fois qu'une valeur absente est utilisée dans un calcul le système d'exceptions doit entrer en jeu. L'utilisation dans un calcul d'une valeur nulle est toujours considérée comme une exception. En conséquence, Discipulus comprend une classe d'exception prédéfinie pour la représenter, la classe NullException :

```
class
        NullException
inherit
        EXCEPTIONS
create
        default_create
feature
        relvar_name : STRING
        attribute_name : STRING
        null_type : STRING
end
```

La classe ci-dessus représente la première version de la classe NullException, ce type d'exception permettra de récupérer dans un premier temps le nom de la variable, l'attribut source de l'exception et le type de l'exception (la cause de l'absence de la valeur).

Une transaction peut traiter une exception par une clause **rescue** et définir une stratégie pour corriger le problème ou le propager :

- Si la stratégie fonctionne, l'exécution de la transaction est reprise depuis le début depuis le début (**retry**).
- Sinon, la transaction échoue et l'exception est propagée.

Exemple :

```
with service test_1 do
 with schema Distribution do
      transaction requete_9 : resultats_9
      do
            from P, A
            select quantite_vis : sum(A.quantite)
            where P.nomP = "vis" and P.noP = A.noP
      rescue
            inspect exception
                  when NullException then
                        if NullException.relvar_name = "A"
                        and NullException.attribute_name = "quantite" then
                              update A set quantite := 0 where IS_NULL(quantite)
                              retry
                        else
                              -- l'exception ne peut être traitée, elle sera propagée
                              print
                                    "NULL intempestif : " +
                                    NullException.relvar_name + " ," +
                                    NullException.attribute_name + " ," +
                                    NullException.null_type
                        end
            else
                  -- toute autre exeception sera propagée
                  print "XXX" exception. developer_exception_name
            end -- inspect
      end -- transaction requete_9
 end -- schema Distribution
end -- service test_1
```

Annexe J
Exemples algorithmiques

Cette annexe présente des exemples de traduction d'éléments du langage algorithmique plus particulièrement ceux présentés à la section 4.3.

La procédure utilisée pour vérifier les antécédents et les conséquents des fonctions et des procédures est définie ainsi :

```
CREATE OR REPLACE PROCEDURE dpl_assertion
      (assertion IN BOOLEAN, message IN VARCHAR2)
IS
      Voilation_contrat EXCEPTION;
BEGIN
 IF assertion = FALSE THEN
      dbms_output.put_line(message);
      RAISE Voilation_contrat;
 END IF;
END;
```

Exemple 1

Discipulus :

```
class
      ACCOUNT

create
      make

feature
      balance: INTEGER -- Amount on the account
      owner: STRING -- Account holder
      minimum_balance: INTEGER -- Lowest permitted balance

      open (who: STRING)
      -- Assign the account to owner who.
      do
            owner := who
```

```
    end

    deposit (sum: INTEGER)
    -- Deposit sum into the account.
    require
        positive: sum > 0
    do
        add (sum)
    ensure
        deposited: balance = old balance + sum
    end

    withdraw (sum: INTEGER)
    -- Withdraw sum from the account.
    require
        positive: sum > 0
        sufficient_funds: sum <= balance – minimum_balance
    do
        add (–sum)
    ensure
        withdrawn: balance = old balance – sum end

    may_withdraw (sum: INTEGER): BOOLEAN
    -- Is there enough money to withdraw sum?
    do
        Result := (balance >= sum + minimum_balance)
    end

feature {NONE}
    add (sum: INTEGER)
    -- Add sum to the balance.
    do
        balance := balance + sum
    end

    make (initial, min: INTEGER)
    -- Initialize account with balance initial and minimum balance min.
    require
        not_under_minimum: initial >= min
    do
        minimum_balance := min
        balance := initial
    ensure
        balance_initialized: balance = initial
        minimum_initialized: minimum_balance = min
```

```
    end

invariant
    sufficient_balance: balance >= minimum_balance

end
```

Résultat de la traduction :

```
CREATE OR REPLACE TYPE dpl_ACCOUNT AS OBJECT (
-- Declaring a constant :
-- Declaring a variables :
    dpl_balance NUMBER(10),
    dpl_minimum_balance NUMBER(10),
    dpl_owner NVARCHAR2(3000),
-- Declaring a procedure :
    MEMBER PROCEDURE dpl_open ( dpl_who IN NVARCHAR2 ),
    MEMBER PROCEDURE dpl_deposit ( dpl_sum IN NUMBER ),
    MEMBER PROCEDURE dpl_withdraw ( dpl_sum IN NUMBER ),
    MEMBER PROCEDURE dpl_add ( dpl_sum IN NUMBER ),
    MEMBER PROCEDURE dpl_make ( dpl_initial IN NUMBER, dpl_min IN NUMBER ),
-- Declaring a function :
    MEMBER FUNCTION dpl_may_withdraw ( dpl_sum IN NUMBER ) RETURN
BOOLEAN,
-- Invariant
    MEMBER PROCEDURE dpl_invariant
);

CREATE OR REPLACE TYPE BODY dpl_ACCOUNT AS
-- Body of procedure :
MEMBER PROCEDURE dpl_open ( dpl_who IN NVARCHAR2 ) IS
-- local variable
BEGIN
    dpl_invariant;
    -- pre-condition
    -- Body
     dpl_owner := dpl_who ;
    -- Post-Condition
    dpl_invariant;
END dpl_open;

MEMBER PROCEDURE dpl_deposit ( dpl_sum IN NUMBER ) IS
-- local variable
    dpl_balance_old NUMBER(10);
BEGIN
```

```
        dpl_balance_old := dpl_balance;
        dpl_invariant;
        -- pre-condition
        dpl_assertion (dpl_sum >  0 ,'#####');
        -- Body
        dpl_add (dpl_sum );
        -- Post-Condition
        dpl_assertion (dpl_balance = dpl_balance_old  +  dpl_sum ,'#####');
        dpl_invariant;
END dpl_deposit;

MEMBER PROCEDURE dpl_withdraw (  dpl_sum IN NUMBER )  IS
-- local variable
        dpl_balance_old NUMBER(10);
 BEGIN
        dpl_balance_old := dpl_balance;
        dpl_invariant;
        -- pre-condition
        dpl_assertion (dpl_sum >  0 ,'#####');
        dpl_assertion (dpl_sum <= dpl_balance  - dpl_minimum_balance ,'#####');
        -- Body
        dpl_add (-dpl_sum );
        -- Post-Condition
        dpl_assertion (dpl_balance = dpl_balance_old  -  dpl_sum ,'#####');
        dpl_invariant;
END dpl_withdraw;

MEMBER PROCEDURE dpl_add (  dpl_sum IN NUMBER )  IS
-- local variable
 BEGIN
dpl_invariant;
-- pre-condition
-- Body
dpl_balance := dpl_balance  +  dpl_sum  ;
-- Post-Condition
dpl_invariant;
END dpl_add;

MEMBER PROCEDURE dpl_make (  dpl_initial IN NUMBER,  dpl_min IN NUMBER )  IS
-- local variable
 BEGIN
        dpl_invariant;
        -- pre-condition
        dpl_assertion (dpl_initial >= dpl_min ,'#####');
        -- Body
```

```
        dpl_minimum_balance := dpl_min ;
        dpl_balance := dpl_initial ;
        -- Post-Condition
        dpl_assertion (dpl_balance = dpl_initial ,'#####');
        dpl_assertion (dpl_minimum_balance = dpl_min ,'#####');
        dpl_invariant;
END dpl_make;

-- Body of function :
MEMBER FUNCTION dpl_may_withdraw ( dpl_sum IN NUMBER ) RETURN BOOLEAN AS
-- local variable
 BEGIN
        -- pre-condition
        -- Body
         RETURN   dpl_balance >= dpl_sum  + dpl_minimum_balance ;
        -- Post-Condition
END may_withdraw;

-- Body of invariant :
MEMBER PROCEDURE dpl_invariant IS
BEGIN
--sufficient_balance
 dpl_assertion (dpl_balance >= dpl_minimum_balance , 'sufficient_balance ');
END dpl_invariant ;
END;
```

Exemple 2

Discipulus

```
class
        ACCOUNT_BONUS
inherit
        ACCOUNT
feature
        add_bonus
        require
                positive: balance > 10000
        do
                add (100)
        end
end
```

Résultat de la traduction

```
CREATE OR REPLACE TYPE dpl_ ACCOUNT_BONUS  UNDER dpl_ACCOUNT(
-- Declaring a constant :
-- Declaring a variables :
-- Declaring a procedure :
      MEMBER PROCEDURE dpl_add_bonus,
-- Declaring a function :
-- Invariant
      MEMBER PROCEDURE dpl_invariant
);

CREATE OR REPLACE TYPE BODY dpl_ACCOUNT_BONUS AS
-- Body of procedure :
      MEMBER PROCEDURE dpl_add_bonus  IS
      -- local variable
       BEGIN
      dpl_invariant;
      -- pre-condition
      dpl_assertion (dpl_balance >  10000 ,'#####');
      -- Body
      dpl_add ( 100 );
      -- Post-Condition
      dpl_invariant;
      END dpl_add_bonus;

-- Body of function :
-- Body of invariant :
      MEMBER PROCEDURE dpl_invariant IS
      BEGIN
      -- @@ dpl_invariant @@
      END dpl_invariant ;
END;
```

Annexe K
Exemple Distribution

Implémentation Discipulus

La relation Approvisionnement

```
with service test_01 do
    with schema Distribution do
        relation Approvisionnement
            noP : STRING;
            oF : STRING ;
            quantite : nullable INTEGER ;
        invariant
            key (noP, noF);
            quantite_positif : quantite > 0;
        end -- relation Approvisionnement
    end -- schema Distribution
end -- service test_01
```

La relation Fournisseur

```
with service test_01 do
    with schema Distribution do
        relation Fournisseur
            noF : STRING;
            nomF : STRING ;
            statut : nullable INTEGER ;
            ville : nullable STRING ;
        invariant
            key noF;
            key (nomF, ville);
            satut_defini : 0 <= statut and statut <= 9999;
        end -- relation Fournisseur
    end -- schema Distribution
end -- service test_01
```

La relation pièce

```
with service test_01 do
    with schema Distribution do
```

```
            relation Piece
                noP : STRING;
                nomP : STRING ;
                couleur : nullable STRING ;
                poids : nullable INTEGER ;
                ville : nullable STRING ;
            invariant
                key noP;
                poids_positif : poids > 0;
            end -- Piece
        end -- schema Distribution
end -- service test_01
```

La variable A de type Approvisionnement

```
with service test_01 do
        with schema Distribution do
            variable A : Approvisionnement
        end -- schema Distribution
end -- service test_01
```

La variable F de type Fournisseur

```
with service test_01 do
        with schema Distribution do
            variable F : Fournisseur
        end -- schema Distribution
end -- service test_01
```

La variable P de type pièce

```
with service test_01 do
        with schema Distribution do
            variable P : Piece
        end -- schema Distribution
end -- service test_01
```

Insertion pour la A de type Approvisionnement

```
with service test_01 do
        with schema Distribution do
            transaction Approvisionnement_ini do
                into A insert
                    [noF : "F1", noP : "P1", quantite : 300],
                    [noF : "F1", noP : "P2", quantite : 200],
                    [noF : "F1", noP : "P3", quantite : 400],
```

```
                    [noF : "F1", noP : "P4", quantite : 200],
                    [noF : "F1", noP : "P5", quantite : 100],
                    [noF : "F1", noP : "P6", quantite : 100],
                    [noF : "F2", noP : "P1", quantite : 300],
                    [noF : "F2", noP : "P2", quantite : 400],
                    [noF : "F3", noP : "P2", quantite : 200],
                    [noF : "F4", noP : "P2", quantite : 200],
                    [noF : "F4", noP : "P4", quantite : 300],
                    [noF : "F4", noP : "P5", quantite : 400];
        end -- Approvisionnement_ini
    end -- schema Distribution
end -- service test_01
```

Insertion pour la F de type Fournisseur

```
with service test_01 do
    with schema Distribution do
        transaction Fournisseur_ini do
            into F insert
                [noF : 'F1', nomF : 'Smith', statut : 20, ville : 'Londres'],
                [noF : 'F2', nomF : 'Dupont', statut : 10, ville : 'Paris'],
                [noF : 'F3', nomF : 'Durand', statut : 30, ville : 'Paris'],
                [noF : 'F4', nomF : 'Clark', statut : 20, ville : 'Londres'],
                [noF : 'F5', nomF : 'El Khoury', statut : 20, ville : 'Beyrouth'];
        end -- Approvisionnement_ini
    end -- schema Distribution
end -- service test_01
```

Insertion pour P de type pièce

```
with service test_01 do
    with schema Distribution do
        transaction Piece_ini do
            into P insert
                [noP : 'P1', nomP : 'ecrou', couleur : 'rouge', poids : 12, ville : 'Londres'] ,
                [noP : 'P2', nomP : 'boulon', couleur : 'vert', poids : 17, ville : 'Paris'],
                [noP : 'P3', nomP : 'vis', couleur : 'bleu', poids : 10, ville : 'Tripoli'] ,
                [noP : 'P4', nomP : 'vis', couleur : 'rouge', poids : 14, ville : 'Londres'] ,
                [noP : 'P5', nomP : 'came', couleur : 'bleu', poids : 12, ville : 'Paris'] ,
                [noP : 'P6', nomP : 'tirette', couleur : 'rouge', poids : 19, ville : 'Londres'];
        end -- Piece_ini
    end -- schema Distribution
end -- service test_01
```

Requête (a)

```
with service test_1 do
    with schema Distribution do
        transaction requete_a : resultats_a
        do
            from F
            select statut , ville
            where statut > 10 and nomF /= "El Khoury"
        rescue
            inspect exception
                when NullException then
                    print
                        "NULL intempestif : " +
                        NullException.relvar_name + " ," +
                        NullException.attribute_name + " ," +
                        NullException.null_type
            end -- inspect
        end -- transaction requete_a
    end -- schema Distribution
end -- service test_01
```

Requête (b)

```
with service test_1 do
    with schema Distribution do
        transaction requete_b : resultats_b
        do
            from
                (
                from A
                group noP
                adding (noP)
                ) as S
            select reponse : count(S.noP)
        end -- transaction requete_b
    end -- schema Distribution
end -- service test_01
```

Requête (c)

```
with service test_1 do
    with schema Distribution do
        transaction requete_c : resultats_c
        do
            from F , P, A
```

```
                select F.nomF , P.nomP, A.quantite
                where F.noF = A.noF and P.noP = A.noP and P.poids > 15
        end -- transaction requete_c
    end -- schema Distribution
end -- service test_01
```

Requête (d)

```
with service test_1 do
    with schema Distribution do
        transaction requete_d : resultats_d
        do
            from
                (
                from A, P
                select P.nomP, A.quantite, P.noP
                where p.noP = A.noP and P.poids > 12
                ) as temp_1
            group temp_1.noP, temp_1.nomP
            adding (nomP, count(quantite))
        end -- transaction requete_d
    end -- schema Distribution
end -- service test_01
```

Requête (e)

```
with service test_1 do
    with schema Distribution do
        transaction requete_e : resultats_e
        do
            from
                (
                from F,
                    (
                    (from F select noF, quantite : null)
                    union
                    (from A select noF, quantite)
                    ) as T
                select F.noF, F.nomF, T.quantite
                where F.noF = T.noF
                ) as S
            group noF, nomF
            adding (nomF , count(quantite))
        end -- transaction requete_e
    end -- schema Distribution
```

```
end -- service test_01
```

Requête (f)

```
with service test_1 do
    with schema Distribution do
        transaction requete_b : res_b
        do
            from (
                from A
                group noP
                adding (noP)) as S
            select reponse : count(S.noP)
        end
    end
end
```

Requête (g)

```
with service test_1 do
    with schema Distribution do
        transaction requete_g : resultats_g
        do
            from F
            select max(statut)
        end
    end
end
```

Requête (h)

```
with service test_1 do
    with schema Distribution do
        transaction requete_h : resultats_h
        do
            from
                (from P
                group nomP
                adding (nomP, nb_ville : count(ville))) as T
            select nomP
            where nb_ville >= 2
        end
    end
end
```

Requête (i)

```
with service test_1 do
    with schema Distribution do
        transaction requete_i : resultats_i
        do
            from
                (
                from
                    ( from P,A,F
                    select F.nomF, A.quantite, P.poids
                    where P.noP = A.noP
                    and F.noF = A.noF
                    ) as T
                group nomF
                adding (nomF , sum_poid_q : sum(poids*quantite))
                ) as S
            select nomF
            where sum_poid_q >= 5000

        end
    end
end
```

Requête (j)

```
with service test_1 do
    with schema Distribution do
        transaction requete_j : resultats_j
        do
            from
                (from
                    (
                    from
                        (
                        from A, F, P
                        select nomP, F.ville
                        where P.noP=A.noP
                            and F.noF=A.noF
                        ) as T
                    group nomP, ville
                    adding (nomP, ville)
                    ) as N
                group nomP
                adding (nomP, v_f: count(ville))) as S,

                (from
```

```
                    (
                    from
                        (
                        from A, F, P
                        select nomP, P.ville
                        where P.noP=A.noP
                            and F.noF=A.noF
                        ) as T
                    group nomP, ville
                    adding (nomP, ville)
                    ) as N
                group nomP
                adding (nomP, v_p: count(ville))) as G

            select S.nomP, S.v_f, G.v_p
            where S.nomP=G.nomP and G.v_p = 1 and S.v_f=1
        end
    end
end
```

Requête (k)

```
with service test_1 do
    with schema Distribution do
        transaction requete_b : resultats_b
        do
                from
                    (
                    from
                        F, (
                        (from F
                        select noF, noP : null)
                        union
                        (from A
                        select noF, noP
                        )) as T
                        select F.noF, F.nomF, T.noP
                        where F.noF = T.noF
                    ) as S
                group noF, nomF
                adding (nomF , count(noP))
        end
    end
end
```

Requête (l)

```
with service test_1 do
    with schema Distribution do
        transaction requete_l : resultats_l
        do
            from F
            select nomF
            where ville /= "Londres"
        end
    end
end
```

Requête (m)

```
with service test_1 do
    with schema Distribution do
        transaction requete_m : resultats_m
        do
            from P,A
            select quantite_de_vis : sum(quantite)
            where P.noP = A.noP
            and P.nomP='vis'
        end
    end
end
```

Requête (n)

```
with service test_1 do
    with schema Distribution do
        transaction requete_n : resultats_n
        do
            from (
                from
                    (
                    from
                        F, (
                        (from F
                        select noF, quantite : null)
                        union
                        (from A
                        select noF, quantite
                        )) as T
                        select F.noF, F.nomF, T.quantite
                        where F.noF = T.noF
```

```
                    ) as S
                    group noF, nomF
                    adding (nomF , q_p : count(quantite))) as N
            select N.nomF
            where q_p = 0
        end
    end
end
```

Requête (o)

```
with service test_1 do
    with schema Distribution do
        transaction requete_o : resultats_o
        do
            from P as PX
            select PX.nomP
            where PX.poids > all (
                                from P as PY
                                select PY.poids
                                where PY.couleur = 'bleu' )
        end
    end
end
```

Requête (p)

```
with service test_1 do
    with schema Distribution do
        transaction requete_p : resultats_p
        do
            from
            (
                from
                    (
                    from
                        F, (
                        (from F
                        select noF, nop : null)
                        union
                        (from A
                        select noF, nop
                        )) as T
                        select F.noF, F.nomF, T.nop
                        where F.noF = T.noF
                    ) as S
```

```
                    group noF, nomF
                    adding (nomF , c_nop : count(noP)) ) as S
                select nomF
                where c_nop = 0
            end
        end
end
```

Résultats de la traduction en SQL

Traduction de la variable A (Approvisionnement)

```
CREATE OR REPLACE PROCEDURE dpl_A IS
BEGIN
 EXECUTE IMMEDIATE '
CREATE TABLE dpl_A_t(
      dpl_noF NVARCHAR2(255) NOT NULL,
      dpl_noP NVARCHAR2(255) NOT NULL,
      dpl_quantite NUMBER(10) ,
      dpl_quantite_n INT,
      CONSTRAINT quantite_positif CHECK (dpl_quantite >  0 ),
      CONSTRAINT ck_noP_noF UNIQUE (dpl_noP, dpl_noF)
)';
END dpl_A;
```

Traduction de la variable F (Fournisseur)

```
CREATE OR REPLACE PROCEDURE dpl_F IS
BEGIN
 EXECUTE IMMEDIATE '
CREATE TABLE dpl_F_t(
      dpl_ville NVARCHAR2(255) ,
      dpl_ville_n INT,
      dpl_statut NUMBER(10) ,
      dpl_statut_n INT,
      dpl_noF NVARCHAR2(255) NOT NULL,
      dpl_nomF NVARCHAR2(255) NOT NULL,
      CONSTRAINT satut_defini CHECK ( 0 <= dpl_statut and dpl_statut <= 9999 ),
      CONSTRAINT ck_noF UNIQUE (dpl_noF),
      CONSTRAINT ck_nomF_ville UNIQUE (dpl_nomF, dpl_ville)
)';
END dpl_F;
```

Traduction de la variable P (Pièce)

```
CREATE OR REPLACE PROCEDURE dpl_P IS
BEGIN
 EXECUTE IMMEDIATE '
CREATE TABLE dpl_P_t(
      dpl_ville NVARCHAR2(255),
      dpl_ville_n INT,
      dpl_nomP NVARCHAR2(255) NOT NULL,
      dpl_couleur NVARCHAR2(255),
      dpl_couleur_n INT,
      dpl_noP NVARCHAR2(255) NOT NULL,
      dpl_poids NUMBER(10),
      dpl_poids_n INT,
      CONSTRAINT poids_positif CHECK (dpl_poids >  0 ),
      CONSTRAINT ck_noP UNIQUE (dpl_noP)
)';
END dpl_P;
```

Traduction de l'insertion des données dans la variable A

```
CREATE OR REPLACE PROCEDURE dpl_Approvisionnement_ini IS
BEGIN
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P1', 300, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P2', 200, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P3', 400, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P4', 200, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P5', 100, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F1', 'P6', 100, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F2', 'P1', 300, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F2', 'P2', 400, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F3', 'P2', 200, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F4', 'P2', 200, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F4', 'P4', 300, 0);
      INSERT INTO dpl_A_t (dpl_noF, dpl_noP, dpl_quantite, dpl_quantite_n) VALUES ('F4', 'P5', 400, 0);
END dpl_Approvisionnement_ini;
```

Traduction de l'insertion des données dans la variable F

```
CREATE OR REPLACE PROCEDURE dpl_Fournisseur_ini IS
BEGIN
      INSERT INTO dpl_F_t ( dpl_noF, dpl_nomF, dpl_statut, dpl_ville, dpl_statut_n, dpl_ville_n)
VALUES ( 'F1', 'Smith', 20, 'Londres', 0, 0);
      INSERT INTO dpl_F_t ( dpl_noF, dpl_nomF, dpl_statut, dpl_ville, dpl_statut_n, dpl_ville_n)
VALUES ( 'F2', 'Dupont', 10, 'Paris', 0, 0);
      INSERT INTO dpl_F_t ( dpl_noF, dpl_nomF, dpl_statut, dpl_ville, dpl_statut_n, dpl_ville_n)
VALUES ( 'F3', 'Durand', 30, 'Paris', 0, 0);
```

```
    INSERT INTO dpl_F_t ( dpl_noF, dpl_nomF, dpl_statut, dpl_ville, dpl_statut_n, dpl_ville_n)
VALUES ( 'F4', 'Clark', 20, 'Londres', 0, 0);
    INSERT INTO dpl_F_t ( dpl_noF, dpl_nomF, dpl_statut, dpl_ville, dpl_statut_n, dpl_ville_n)
VALUES ( 'F5', 'El Khoury', 20, 'Beyrouth', 0, 0);
END dpl_Fournisseur_ini;
```

Traduction de l'insertion des données dans la variable P

```
CREATE OR REPLACE PROCEDURE dpl_Piece_ini IS
BEGIN
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P1', 'ecrou', 'rouge', 12, 'Londres', 0, 0, 0);
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P2', 'boulon', 'vert', 17, 'Paris', 0, 0, 0);
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P3', 'vis', 'bleu', 10, 'Tripoli', 0, 0, 0);
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P4', 'vis', 'rouge', 14, 'Londres', 0, 0, 0);
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P5', 'came', 'bleu', 12, 'Paris', 0, 0, 0);
    INSERT INTO dpl_P_t ( dpl_noP, dpl_nomP, dpl_couleur, dpl_poids, dpl_ville, dpl_couleur_n,
dpl_poids_n, dpl_ville_n) VALUES ( 'P6', 'tirette', 'rouge', 19, 'Londres', 0, 0, 0);
END dpl_Piece_ini;
-- Tests
```

Traduction de la requête (a) : Statut et ville des fournisseurs dont le statut est supérieur à 10 (a l'exception du fournisseur El Khoury).

```
CREATE OR REPLACE PROCEDURE dpl_requete_a IS-- mes declarations de variables
    Attribut_Name  VARCHAR2;
    Relvar_Name    VARCHAR2;
    Null_Exception EXCEPTION;
    Number_null    INTEGER;
BEGIN
    SELECT count(*) INTO Number_null FROM dpl_F WHERE dpl_statut_n > 0 ;
        IF Number_null > 0 THEN
            Attribut_Name := 'dpl_statut';
            Relvar_Name := 'dpl_F';
            Raise Null_Exception;
        END IF;
    EXECUTE IMMEDIATE
        'CREATE TABLE dpl_requete_a AS(
        SELECT DISTINCT
        dpl_statut , dpl_ville
        FROM dpl_F_t
        WHERE dpl_statut >  10  and  dpl_nomF  <> "El Khoury" )';
    EXCEPTION -- mon traitement des exceptions
```

```
            WHEN Null_Exception THEN
            -- strategie pour Les NULL
             dbms_output.put_line( ' OOoops valeur NULL detecte !!!' );
             dbms_output.put_line( ' detail :' );
             dbms_output.put_line( ' Relvar :' + Relvar_Name);
             dbms_output.put_line( ' Attribut :' + Attribut_Name);
      -- la fin du bloc executable
END dpl_requete_a;
```

Traduction de la requête (b) : Nombre de pièces distinctes approvisionnées.

```
EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_b AS(
SELECT DISTINCT  count(dpl_S_t.dpl_noP ) AS dpl_reponse
FROM (SELECT dpl_noP
            FROM dpl_A_t
            GROUP BY dpl_noP ) dpl_S_t;
)';
END dpl_requete_b;
```

Traduction de la requête (c) : Pour toutes les pièces de poids supérieur à 15, le nom de la pièce, le nom du fournisseur et la quantité approvisionnée.

```
CREATE OR REPLACE PROCEDURE dpl_requete_c IS-- mes declarations de variables
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_c AS(
      SELECT DISTINCT  dpl_F_t.dpl_nomF , dpl_P_t.dpl_nomP , dpl_A_t.dpl_quantite
      FROM  dpl_F_t , dpl_P_t , dpl_A_t
      WHERE  dpl_F_t.dpl_noF = dpl_A_t.dpl_noF
      AND dpl_P_t.dpl_noP = dpl_A_t.dpl_noP
      AND dpl_P_t.dpl_poids > 15;)';
END dpl_requete_c;
```

Traduction de la requête (d) : Pour chacune des pièces dont le poids est supérieur a 12,le nombre de fournisseurs potentiels et le nombre de fournisseurs effectifs.

```
CREATE OR REPLACE PROCEDURE dpl_requete_d IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_d AS(
      SELECT dpl_nomP , count(dpl_quantite)
      FROM (SELECT dpl_P_t.dpl_nomP , dpl_A_t.dpl_quantite , dpl_P_t.dpl_noP
                  FROM dpl_A_t , dpl_P_t
                  WHERE dpl_p_t.dpl_noP = dpl_A_t.dpl_noP and dpl_P_t.dpl_poids > 12 )
dpl_temp_1
      GROUP BY dpl_noP , dpl_nomP ;
      )';
```

```
END dpl_requete_d;
```

Traduction de la requête (e) : Pour chacun des fournisseurs, le nombre pièces offertes et la quantité totale de pièces approvisionnées.

```
CREATE OR REPLACE PROCEDURE dpl_requete_e IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_e AS(
      SELECT dpl_nomF , count(dpl_quantite)
      FROM (SELECT dpl_F_t.dpl_noF , dpl_F_t.dpl_nomF , dpl_T_t.dpl_quantite
                  FROM dpl_F_t , ( (SELECT dpl_noF , null AS dpl_quantite
                                    FROM dpl_F_t
                                    ) union (
                                    SELECT dpl_noF , dpl_quantite
                                    FROM dpl_A_t
                                    ) ) dpl_T_t
      WHERE dpl_F_t.dpl_noF = dpl_T_t.dpl_noF ) dpl_S
      GROUP BY dpl_noF , dpl_nomF ;
      )';
END dpl_requete_e;
```

Traduction de la requête (f) : Décupler son statut de chacun des fournisseurs dont la ville est la même que celle d'au moins une pièce.

```
CREATE OR REPLACE PROCEDURE dpl_requete_f IS
BEGIN
      UPDATE dpl_F_t
      SET   dpl_statut = dpl_statut *  10
      WHERE dpl_ville in   (SELECT DISTINCT dpl_ville  FROM dpl_P_t  );
END dpl_requete_f;
```

Traduction de la requête (g) : Le plus grand statut d'un fournisseur.

```
CREATE OR REPLACE PROCEDURE dpl_requete_g IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_g AS(
      SELECT DISTINCT max( dpl_statut)
      FROM dpl_F_t
)';
END dpl_requete_g;
```

Traduction de la requête (h) : Les pièces fabriquées dans au moins deux villes.

```
CREATE OR REPLACE PROCEDURE dpl_requete_h IS
```

```
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_h AS(
      SELECT DISTINCT  dpl_nomP
      FROM (SELECT dpl_nomP , (count(dpl_ville)) AS dpl_nb_ville
                  FROM dpl_P_t
                  GROUP BY dpl_nomP
            ) dpl_T_t
      WHERE dpl_nb_ville >= 2;
       )';
END dpl_requete_h;
```

Traduction de la requête (i) : Les fournisseurs ayant fourni pour plus de 5 000 kg de pièces.

```
CREATE OR REPLACE PROCEDURE dpl_requete_i IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_i AS(
      SELECT DISTINCT  dpl_nomF
      FROM (SELECT dpl_nomF , (sum(dpl_poids * dpl_quantite )) AS dpl_sum_poid_q
                  FROM (SELECT dpl_F_t.dpl_nomF , dpl_A_t.dpl_quantite , dpl_P_t.dpl_poids
                              FROM dpl_P_t , dpl_A_t , dpl_F_t
                              WHERE dpl_P_t.dpl_noP = dpl_A_t.dpl_noP and dpl_F_t.dpl_noF =
dpl_A_t.dpl_noF ) dpl_T_t
            GROUP BY dpl_nomF ) dpl_S_t
      WHERE dpl_sum_poid_q >= 5000 ;
       )';
END dpl_requete_i;
```

Traduction de la requête (j) : Les piéces distribuées par un seul fournisseur et produites dans une seule ville.

```
CREATE OR REPLACE PROCEDURE dpl_requete_j IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_j AS(
       SELECT DISTINCT dpl_S_t.dpl_nomP , dpl_S_t.dpl_v_f , dpl_G_t.dpl_v_p
      FROM (SELECT dpl_nomP , (count(dpl_ville)) AS dpl_v_f
                  FROM (SELECT dpl_nomP , dpl_ville
                        FROM (SELECT dpl_nomP , dpl_F_t.dpl_ville
                                    FROM dpl_A_t , dpl_F_t , dpl_P_t
                                    WHERE dpl_P_t.dpl_noP = dpl_A_t.dpl_noP
                                    AND dpl_F_t.dpl_noF = dpl_A_t.dpl_noF ) dpl_T_t
                        GROUP BY dpl_nomP , dpl_ville
                  ) dpl_N_t
                  GROUP BY dpl_nomP
                  ) dpl_S_t ,
                  (SELECT dpl_nomP , (count(dpl_ville)) AS dpl_v_p
```

```
                    FROM (SELECT dpl_nomP , dpl_ville
                            FROM (SELECT dpl_nomP , dpl_P_t.dpl_ville
                                    FROM dpl_A_t , dpl_F_t , dpl_P_t
                                    WHERE dpl_P_t.dpl_noP = dpl_A_t.dpl_noP and
dpl_F_t.dpl_noF = dpl_A_t.dpl_noF ) dpl_T_t
                    GROUP BY dpl_nomP , dpl_ville
                    ) dpl_N_t
                GROUP BY dpl_nomP
                ) dpl_G_t
    WHERE dpl_S_t.dpl_nomP = dpl_G_t.dpl_nomP and dpl_G_t.dpl_v_p = 1 and
dpl_S_t.dpl_v_f = 1;
     )';
END dpl_requete_j;
```

Traduction de la requête (k) : Le nombre de pièces différentes distribuées par chaque fournisseur (si un fournisseur n'en distribue aucune, indiquer zéro).

```
CREATE OR REPLACE PROCEDURE dpl_requete_k IS
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_k AS(
    SELECT dpl_nomF , count(dpl_noP)
    FROM (SELECT dpl_F_t.dpl_noF , dpl_F_t.dpl_nomF , dpl_T_t.dpl_noP
            FROM dpl_F_t , ( (SELECT dpl_noF , null AS dpl_noP
                                FROM dpl_F_t
                            ) union
                            (SELECT dpl_noF , dpl_noP
                                FROM dpl_A_t
                            ) ) dpl_T_t
    WHERE dpl_F_t.dpl_noF = dpl_T_t.dpl_noF ) dpl_S
    GROUP BY dpl_noF , dpl_nomF ;
    )';
END dpl_requete_k;
```

Traduction de la requête (l) : La liste des noms des fournisseurs situe ailleurs qu'à Londres.

```
CREATE OR REPLACE PROCEDURE dpl_requete_l IS-- mes declarations de variables
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_l AS(
    SELECT DISTINCT  dpl_nomF
    FROM dpl_F_t
    WHERE dpl_ville <> "Londres";
     )';
END dpl_requete_l;
```

Traduction de la requête (m) : La quantité de vis approvisionnées.

```
CREATE OR REPLACE PROCEDURE dpl_requete_m IS-- mes declarations de variables
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_m AS(
      SELECT DISTINCT sum(dpl_quantite) AS dpl_quantite_de_vis
      FROM dpl_P_t , dpl_A_t
      WHERE dpl_P_t.dpl_noP = dpl_A_t.dpl_noP
      AND dpl_P_t.dpl_nomP = "vis";
       )';
END dpl_requete_m;
```

Traduction de la requête (n) : Les fournisseurs qui n'ont pas d'approvisionnement

```
CREATE OR REPLACE PROCEDURE dpl_requete_n IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_n AS(
      SELECT DISTINCT  dpl_N_t.dpl_nomF
      FROM (SELECT dpl_nomF , (count(dpl_quantite)) AS dpl_q_p
                  FROM (SELECT dpl_F_t.dpl_noF , dpl_F_t.dpl_nomF , dpl_T_t.dpl_quantite
                              FROM dpl_F_t , ( (SELECT dpl_noF , null AS dpl_quantite
                                                FROM dpl_F_t
                                                ) union
                                                (SELECT dpl_noF , dpl_quantite
                                                FROM dpl_A_t
                                                ) ) dpl_T_t
                  WHERE dpl_F_t.dpl_noF = dpl_T_t.dpl_noF ) dpl_S_t
                  GROUP BY dpl_noF , dpl_nomF
            ) dpl_N_t
      WHERE dpl_q_p = 0;
      )';
END dpl_requete_n;
```

Traduction de la requête (o) : Les pièces donc le poids est supérieur à celui de toute pièce de couleur bleu

```
CREATE OR REPLACE PROCEDURE dpl_requete_o IS-- mes declarations de variables
BEGIN
EXECUTE IMMEDIATE 'SELECT DISTINCT  dpl_PX_t.dpl_nomP
FROM dpl_P_t dpl_PX_t
WHERE dpl_PX_t.dpl_poids > all (SELECT dpl_PY_t.dpl_poids
                                FROM dpl_P_t dpl_PY_t
                                WHERE dpl_PY_t.dpl_couleur = "bleu" );
)';
END dpl_requete_o;
```

Traduction de la requête (p) : Les fournisseurs qui n'approvisionnent pas au moins une pièce.

```
CREATE OR REPLACE PROCEDURE dpl_requete_p IS
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_p AS(
    SELECT DISTINCT dpl_nomF
    FROM (SELECT dpl_nomF , (count(dpl_noP)) AS dpl_c_nop
            FROM (SELECT dpl_F_t.dpl_noF , dpl_F_t.dpl_nomF , dpl_T_t.dpl_nop
                    FROM dpl_F_t , ( (SELECT dpl_noF , null AS dpl_nop
                                        FROM dpl_F_t
                                    ) union
                                    (SELECT dpl_noF , dpl_nop
                                    FROM dpl_A_t
                                    ) ) dpl_T_t
            WHERE dpl_F_t.dpl_noF = dpl_T_t.dpl_noF ) dpl_S_t
            GROUP BY dpl_noF , dpl_nomF
        ) dpl_S_t
    WHERE dpl_c_nop = 0;
    )';
END dpl_requete_p;
```

Traduction de la requête (r, s) : Les fournisseurs dont le statut est supérieur à 10 % de la quantité totale des piéces qu'ils approvisionnent.

```
CREATE OR REPLACE PROCEDURE dpl_requete_r IS-- mes declarations de variables
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE dpl_requete_r AS(
    SELECT DISTINCT  dpl_F_t.dpl_nomF
    FROM dpl_F_t , dpl_A_t
    WHERE dpl_statut > (SELECT sum(dpl_quantite) * 0.1
                        FROM dpl_A_t
                        WHERE dpl_F_t.dpl_noF = dpl_A_t.dpl_noF );
    )';
END dpl_requete_r;
```

Bibliographie

Les livres

[L1] Chris J. Date. (2000). Introduction aux bases de données, 7 édition.

[L2] Chris J. Date. (2005). Database In Depth : Relational Theory for Practitioners.

[L3] Chris J. Date, Hugh Darwen. (1997). A guide to the SQL standard, 4 édition.

[L4] Jeffrey D. Ullman, Jennifer Widom. (2002). DataBase Systems, the complete book.

[L5] Shamkant Navathe, Ramez Elmasri, Daniel Serain. (2004). Conception et architecture des bases de données, 4 édition.

[L6] Ramez Elmasri. (2007). Fundamentals of database systems, 5 édition.

[L7] Jeffrey D. Ullman, Jennifer Widom. (2007). A first course in database systems, 3 édition.

[L8] David Maier. (1983). The Theory of Relational Databases.

[L9] Edgar F. Codd. (1990). The Relational Model for Database Management. 2 édition.

[L10] Chris J. Date, Hugh Darwen. (2007). Databases, Types, and the Relational Model, The Third Manifesto, 3 édition.

[L11] Shamkant Navathe, Ramez Elmasri, Daniel Serain. (2007). Fundamentals of database systems, 1 édition.

[L12] Aurélie Ottavi. (2012). Élaboration d'un banc d'essai pour la comparaison de SGBDR de modèles différents. Département d'informatique, Faculté des sciences, Université de Sherbrooke.

[L13] Bertrand Meyer. (2001). An Eiffel Tutorial, ISE Technical Report TR-EI-66/TU. Corresponds to release 5.0 of the ISE Eiffel environment.

[L14] Philip A. Bernstein, Eric Newcomer. (1997). Principles of transaction processing, Morgan Kaufmann.

[L15] Alex Osuna, Jawahar Lal, Roger Sanders, Jeremy Brumer. (2006). Using IBM DB2 UDB with IBM System Storage N series, 1 édition.

[L16] Chris J. Date. (2003). An Introduction to Database Systems, 8 édition.

[L17] Jeffrey D. Ullman. (1998). Elements of ML Programming.

[L18] Joseph E. Stoy. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.

Les articles de revue

[A1] ANSI/XJ/SPARC. (1975) Study Group on Data Base Management Systems. Interim Report. FDT, ACM SIGMOD.

[A2] Edgar F. Codd. (1983). A relational model of data for large shared data banks. New York, ACM, Volume 26 Issue 1, Jan. 1983, p. 64 - 69

[A3] Edgar F. Codd. (1974). Understanding relations. New York, ACM SIGMOD Volume 6 Issue 3, p. 40 - 42.

[A4] Edgar F. Codd. (1979). Extending the database relational model to capture more meaning. New York, ACM Transactions on Database Systems, Volume 4 Issue 4, p. 397 - 434.

[A5] Edgar F. Codd. (1986). Missing information (applicable and inapplicable) in relational databases. New York, ACM SIGMOD, Volume 15 Issue 4, p. 53 - 53.

[A6] Edgar F. Codd. (1981). Data models in database management. New York, ACM SIGMOD. Volume 11 Issue 2, p 112 - 114

[A7] Vassiliou Yannis. (1979). Null values in data base management a denotational semantics approach. New York, ACM SIGMOD. p. 162 - 169

[A8] Lipski JR Witold. (1977). On semantic issues connected with incomplete information databases. VLDB, Volume 3, p. 491 - 491.

[A9] Biskup Joachim. (1983). A foundation of Codd's relational maybe-operations. New York, ACM (TODS). Volume 8 Issue 4, p. 608 - 636.

[A10] Zongmin Ma. (1996) Using multivalued logic in relational database containing null value. Journal of Computer Science and Technology, Volume 11, Issue 4, p. 421-426,

[A11] Chris J. Date. (1990). Why duplicate rows are prohibited. In C. J. Date & A. Warden, Relational database: Writings. Addison-Wesley, Massachusetts, USA.

[A12] Chris J. Date. (1988). Why relational. In C. J. Date & A. Warden (Ed.), Relational database: Writings. Addison-Wesley, Massachusetts, USA.

[A13] Chris J. Date. (2008). A critique of Claude Rubinson's paper nulls, three - valued logic, and ambiguity in SQL: critiquing Date's critique. *SIGMOD Rec.*, volume 37, numéro 3, p. 20-22.

[A14] Chris J. Date. (1983) NULL values in database management. In C. J. Date (Ed.), Relational database: Selected writings. Addison-Wesley, Massachusetts, USA.

[A15] Chris J. Date. (1984) A critique of the SQL database language. In C. J. Date (Ed.), Relational database: Selected writings. Addison-Wesley, Massachusetts, USA.

[A16] Luca Cardelli, Peter Wegner. (1985). On Understanding Types, Data Abstraction, and Polymorphism, New York, ACM Computing Surveys. Volume 17 Issue 4, pp. 471 - 523.

[A17] Bertrand Meyer. (2001). Overloading vs Object Technology. Journal of Object-Oriented Programming (JOOP), vol. 14, no. 4.

[A18] Umeshwar Dayal, Nathan Goodman, Randy H. Katz. (1982). An Extended Relational Algebra with Control Over Duplicate Elimination. New York, ACM SIGACT-SIGMOD, pp. 117 - 123.

[A19] Alain Pirotte. (1982). A precise definition of basic relational notions and of the relational algebra. New York, ACM SIGMOD. Volume 13 Issue 1, pp. 30 - 45.

[A20] Carlo Zaniolo. (1982). Database relation with Null. New York, ACM SIGACT-SIGMOD, pp. 27 - 33.

[A21] Ken-Chih Liu, Rajshekhar Sunderraman. (1990). Indefinite and Maybe Information in Relational Databases. New York, ACM-TODDS. Volume 15 Issue 1, p. 1 - 39

[A22] Georg Gottlob, Roberto Zicari. (1988) Closed World Databases Opened Through Null Values. VLDB, pp. 50 - 61.

[A23] Chris J. Date. (2008) A Critique of Claude Rubinson's Paper: Nulls, Three - Valued Logic, and Ambiguity in SQL: Critiquing Date's Critique. New York, ACM SIGMOD, Volume 37 Issue 3, p. 20-22

[A24] John Grant. (2008). NULL Values in SQL. New York, ACM SIGMOD, Volume 37 Issue 3, p. 23-25

[A25] R. Reiter. (1978). On closed world data bases. In H. Gallaire and J. Minker, editors, Logic and Data Bases. New York. Plenum Publ. Co. pp. 119–40.

[A26] Bertrand Meyer. (2001). Overloading vs Object Technology. Journal of Object-Oriented Programming (JOOP), vol. 14, no. 4.

[A27] ISO/IEC (2003). ISO/IEC 9075-2:2003, "SQL/Foundation". ISO/IEC.

[A28] Safia Nait Bahloul, Youssef Amghar, M. Sayah. (2004), A*: Algebra for an Extended Object/Relational Model, International Journal of Computer Science & Applications, volume 1, n° 2, pp. 76 – 95.

[A29] Min Wang, Yuan-Chi, Chang, and Sriram Padmanabhan. (2002). Supporting Ecient Parametric Search of E-Commerce Data: a Loosely-Coupled Solution. London. pp. 409-426

[A30] Donald D. Chamberlin, Raymond F. Boyce. (1974). SEQUEL: A Structured English Query Language. IBM Research Laboratory San Jose. CA, USA.

[A31] Donald D. Chamberlin. (1976). SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Research Laboratory, San Jose, CA, USA. Volume : 20 , Issue : 6, pp. 560 - 575

[A32] Morton M Astrahan, Michael W Blasgen, Donald D. Chamberlin. (1976). System R: Relational Approach to Database Managementé. ACM Transactions on Database Systems (TODS), Volume 1 Issue 2, Pages 97 – 137, ACM New York, NY, USA

[A33] Jens Gößner, Philip Mayer, Friedrich Steimann. (2004).Interface utilization in the Java Development Kit. Proceedings of the 2004 ACM symposium on Applied computing. Pages 1310 – 1315, ACM New York, NY, USA

Les rapports de recherche et les notes techniques

[R1] Laïla Alami, Luc Lavoie. (2010). Proposition d'un d'un système de typage pour SBGD. Groupe Μῆτις, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Sherbrooke, Québec, CDN. RR-0001.

[R2] Luc Lavoie. (2012). Proposition d'un modèle de système ouvert pour SGBD. Groupe Μῆτις, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Sherbrooke, Québec, CDN. RR-0002.

[R3] Zouhir Abouaddaoui, Christina Khnaisser, Luc Lavoie. (2012). Définition du langage Discipulus. Groupe Μῆτις, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Sherbrooke, Québec, CDN. RR-0005.

[R4] Zouhir Abouaddaoui, Christina Khnaisser, Luc Lavoie. (2012). Construction d'un compilateur-interprète pour le langage Discipulus. Groupe Μῆτις, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Sherbrooke, Québec, CDN. RR-0007.

Les sites internet

[S1] http://fr.wikipedia.org/wiki/Structured_Query_Language (page consultée le 01 septembre 2010)

[S2] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html (page consultée le 01 juillet 2011)

[S3] http://javacc.java.net (page consultée le 01 mai 2011)

[S4] http://www.cocolab.com/cocktail.html (page consultée le 01 mai 2011)

[S5] http://www.ANTLR.org/ (page consultée le 01 mai 2011)

[S6] http://javacc.java.net/doc/JJTree.html (page consultée le 01 mai 2011)

[S7] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42924 (page consultée le 01 septembre 2010)

[S8] http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html (page consultée le 02 mai 2011)

[S9] http://www.gobosoft.com/eiffel/gobo/gec/index.html (page consultée 01avril 2011)

[S10] http://tecomp.sourceforge.net/index.php?file=doc/tecom (page consultée 01 avril 2011)

[S11] http://codecover.org/ (page consultée 01 octobre 2011)

[S12] http://dbappbuilder.sourceforge.net/Rel.php (page consultée 01 octobre 2009)

[S13] http://www.w3.org/XML/Schema (page consultée le 01 mai 2011)

[S14] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42924 (page consultée le 01 Octobre 2011)

[S15] http://www.stack.nl/~dimitri/doxygen/ (page consultée le 01 Octobre 2011)

[S16] http://www.javaworld.com/jw-12-1998/jw-12-techniques.html (pages consultée le 06 juillet 2012)