DÉFINITION D'UN LANGAGE FORMEL DE REQUÊTES BASÉ SUR UN MODÈLE ENTITÉ-ASSOCIATION

par

Imad Yassine

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
8 juillet 2009

Your file Votre référence
ISBN: 978-0-494-53191-4
Our file Notre référence
ISBN: 978-0-494-53191-4

Le 8 juillet 2009

le jury a accepté le mémoire de M. Imade Yassine dans sa version finale.

Membres du jury

M. Marc Frappier
Directeur
Département d'informatique

M. Richard St-Denis
Codirecteur
Département d'informatique

M. Frédéric Gervais
Membre

M. Bessam Abdulrazak
Président-rapporteur
Département d'informatique

À la mémoire de Kendil Rkiya, ma grand-mère bien aimée.

Sommaire

Dans le cadre du développement des systèmes d'information, les méthodes formelles de spécification permettent d'éliminer ou de réduire le besoin des phases de conception et d'implémentation, l'importance étant mise sur les phases de spécification des besoins et d'analyse. La méthode EB^3 est une de ces méthodes conçue pour spécifier le comportement fonctionnel des systèmes d'information. Elle est basée sur une algèbre de processus dont les requêtes de sortie sont spécifiées à l'aide de règles d'entrée-sortie.

Ce mémoire présente la définition d'un langage formel de requêtes basé sur un modèle entité-association, qui permet de spécifier les requêtes de sortie de la méthode de spécification EB^3. Des schémas de traduction sont élaborés pour réaliser la compilation de ces requêtes vers du code exécutable. Ils sont utilisés par l'outil EB^{3}QG développé et intégré dans la plateforme APIS dédiée à la génération automatique des systèmes d'information à l'aide de la méthode de spécification EB^3.

SOMMAIRE

Remerciements

Je tiens à remercier mon directeur de recherche, le professeur Marc Frappier qui m'a donné l'opportunité d'effectuer une maîtrise au sein de son groupe de recherche en ingénierie du logiciel (GRIL). Je souhaite le remercier aussi pour la qualité de l'encadrement et les conseils utiles qu'il m'a apportés tout au long de la période de mes études et qui m'ont permis de mener à bien mon projet de recherche. Je le remercie également pour m'avoir offert un cadre de travail propice et d'avoir mis à ma disposition les outils nécessaires à ma recherche.

Mes sincères remerciements vont aussi à mon codirecteur de recherche, le professeur Richard St-Denis qui m'a aidé dans la rédaction de ce mémoire et qui a partagé avec moi ses connaissances et son expertise sur la théorie des langages et la méthodologie de recherche. Je suis également très reconnaissant aux membres du laboratoire GRIL, en particulier Michel Embe Jiague, pour leur soutien continu.

Mes remerciements vont aussi au docteur Benoît Fraikin pour son apport de connaissances sur la méthode EB^3 et les algèbres de processus, ses conseils ainsi que son aide précieuse.

Je tiens à remercier également mes chers parents, ma soeur et mon frère pour m'avoir soutenu et encouragé durant toute la période de mes études de maîtrise.

Enfin, je remercie l'Université de Sherbrooke, son personnel administratif et ses professeurs pour leur accueil et leur encadrement.

REMERCIEMENTS

Abréviations

ANTLR ANother Tool for langage Recognition

APIS Automated Production of Information Systems

AST Abstract Syntax Tree

EA Entité-association

EAE Entité-association étendu

EB^3 Entity-Based Black-Box

EB^3GG EB^3 Guard Generator

EB^3QG EB^3 Query Generator

EB^3PAI EB^3 Process Algebra Interpreter

EB^3TG EB^3 Transactions Generator

EER Enhanced Entity-Relationship

ER Entity-Relationship

ES Entrée-sortie

GRIL Groupe de recherche en ingénierie du logiciel

GUI Graphical User Interface

IO Input Output

IS Information System

JDBC Java DataBase Connectivity

PE Process Expression

SI Système d'information

SQL Structured Query langage

UML Unified Modeling langage

XML Extensible Markup langage

Table des matières

Liste des figures

Introduction

Contexte

Un système d'information (SI) coordonne grâce à l'information les activités d'une organisation et lui permet ainsi d'atteindre ses objectifs. Il est le véhicule de la communication dans une organisation. De plus, le SI représente l'ensemble des ressources humaines, matérielles et logicielles organisées pour collecter, stocker, traiter et communiquer des informations. L'utilisation de tels systèmes permet d'automatiser le traitement de l'information. Bien que leur utilisation soit largement répandue, la réalisation des SI nécessite beaucoup de ressources humaines et logicielles et demande un important effort de développement.

Une approche possible pour le développement de SI consiste à utiliser les méthodes formelles qui permettent de réduire l'effort de développement aux seules phases de spécification des besoins et d'analyse [6]. Ainsi le développement peut être obtenu à l'aide d'outils qui génèrent un SI automatisé en ne se basant que sur les résultats de ces deux phases. Les méthodes formelles incluent des techniques qui permettent de raisonner rigoureusement, à l'aide de la logique mathématique, sur des applications logicielles afin de démontrer leur validité par rapport à une spécification. Ces méthodes sont très appréciées dans le développement des applications critiques, car elles permettent d'obtenir une très forte assurance de leur bon fonctionnement par rapport aux attentes et aux besoins des utilisateurs. La méthode EB^3 est une de ces méthodes de spécification formelle qui permet l'automatisation du développement des systèmes d'information.

Problématique

Dans le cadre du projet APIS dédié à la génération automatique des systèmes d'information à l'aide de la méthode de spécification EB^3, les règles d'ES font partie intégrante de toute spécification EB^3. De plus, ces règles sont utilisées par l'interpréteur des expressions de processus EB^{3}PAI. Après l'étude de l'existant, il s'est avéré que ces règles ne sont pas prises en charge par la plateforme APIS. Pour ces raisons, le problème d'écriture des requêtes de sortie à l'aide d'un langage formel de requêtes a été envisagé.

Résultats

Nous avons conçu un langage formel de requêtes basé sur un modèle entité-association (EA) qui permet de spécifier les requêtes de sortie des règles d'ES d'une spécification EB^3. De plus, des schémas de traduction ont été définis pour traduire les requêtes de sortie en requêtes SQL. Un outil appelé EB^{3}QG a été développé afin de prendre en charge ces schémas de traduction et de générer une implémentation des requêtes de sortie en requêtes SQL.

Méthodologie

La réalisation de notre projet est rendue possible grâce à une démarche précise. Premièrement, nous avons effectué une recherche bibliographique dans le but de trouver un ou des langages de requêtes sur lesquels nous avons appuyé notre travail. Deuxièmement, nous avons étudié la méthode de spécification EB^3 afin de déterminer comment intégrer des requêtes de sortie. Troisièmement, nous avons emprunté le concept d'expressions de chemins dans la définition d'un langage formel de requêtes basé sur un modèle EA. Ce concept a été introduit dans le cadre des bases de données orientées objet. Nous avons utilisé une grammaire hors contexte pour décrire la syntaxe de notre langage de requêtes. Quatrièmement, nous avons établi des schémas de traduction pour passer des requêtes d'expressions de chemin vers des requêtes SQL. Enfin, nous avons produit un outil nommé EB^{3}QG, effectué des tests fonctionnels et intégré ce nouvel outil dans la plateforme APIS.

Structure du mémoire

Le chapitre 1 introduit le projet APIS et la méthode de spécification EB^3. Le chapitre 2 présente un article intitulé « Definition of a formal query langage for entity-relationship models ». Cet article décrit un nouveau langage utile pour la spécification des requêtes de sortie contenues dans les règles d'ES de la méthode EB^3. Le chapitre 3 décrit l'implémentation de l'outil EB^{3}QG et son intégration au sein de la plateforme APIS. Enfin, le chapitre 4 présente une étude de cas, un système de gestion d'une compagnie, qui illustre l'utilisation de notre langage.

Chapitre 1

Projet APIS

Ce chapitre présente le projet APIS dédié au développement rapide de systèmes d'information (SI) à partir de spécifications formelles ; le processus est appelé « *information system synthesis* ». La méthode de spécification EB^3 (Section 1.1) propose une notation formelle pour décrire les différents composants nécessaires à la génération d'un SI. La plateforme APIS regroupe les outils qui permettent l'exécution d'un SI par une interprétation efficiente du code généré à partir des différents composants de la spécification EB^3.

1.1 La méthode EB^3

Les SI sont spécifiés à l'aide de la méthode EB^3 [7] qui comporte un langage formel à base de traces. La séquence des événements acceptés par le système est décrite par une algèbre de processus. Les entités et les associations sont décrites à l'aide d'un diagramme de classes, les attributs sont calculés au moyen de fonctions récursives définies sur les traces valides du système.

1.1.1 Le diagramme EA

Le diagramme EA décrit les entités, les associations et les attributs. Il est basé sur le concept du modèle entité-association [4] et utilise un sous-ensemble de la notation graphique UML pour les diagrammes de classes.

1.1.2 L'interface utilisateur

La spécification d'une interface utilisateur permet de déterminer l'agencement des éléments issus d'un SI ainsi que leur intégration dans une interface de type Web. Cela comprend les formulaires nécessaires à l'entrée de données et de paramètres des actions, les zones d'affichages des résultats ainsi que la possibilité de signaler des erreurs à l'utilisateur.

1.1.3 Les fonctions d'attributs

Les fonctions d'attributs définies sur les traces valides de l'expression de processus « *main* », affectent des valeurs aux attributs des entités ou des associations. Ces fonctions sont totales et sont représentées dans un style fonctionnel, comme en CAML [13] avec un filtrage sur le dernier événement de la trace. Elles retournent les valeurs d'attributs qui sont valides pour l'état courant du système après avoir exécuté les événements de la trace.

1.1.4 Les règles d'ES

Les règles d'ES associent une sortie à chaque événement valide à l'entrée du système. Ces règles sont spécifiées par un langage formel de requêtes décrit dans le chapitre 2 et permettent d'extraire les données à afficher relatifs aux actions de l'utilisateur.

1.1.5 Les expressions de processus

En EB^3 l'algèbre de processus permet de définir l'ordonnancement des différents événements. La notation EB^3 pour les expressions de processus est similaire à la notation CSP d'Hoare [9]. Des expressions de processus complexes peuvent être construites à partir des expressions de processus élémentaires au moyen des opérateurs suivants : l'opérateur de séquence (dénoté par « . »), le choix (« | »), la fermeture de Kleene (« * »), l'entrelacement noté (« ||| »), la composition parallèle (« || »), la garde (« $\Longrightarrow$ »), le choix quantifié (« | $x : T : ...$ ») et l'entrelacement quantifié (« || $x : T : ...$ »).

1.2 La plateforme APIS

Dans le projet APIS, plutôt que d'utiliser les techniques de raffinement utilisées dans l'implémentation des systèmes à base d'états, le SI est obtenu par l'interprétation et la génération de code à partir des différents éléments de la spécification EB^3. La Figure 1.1 présente l'organisation entre les différents modules de la plateforme APIS ainsi que les relations entre eux et entre les différentes composantes de la méthodes EB^3.

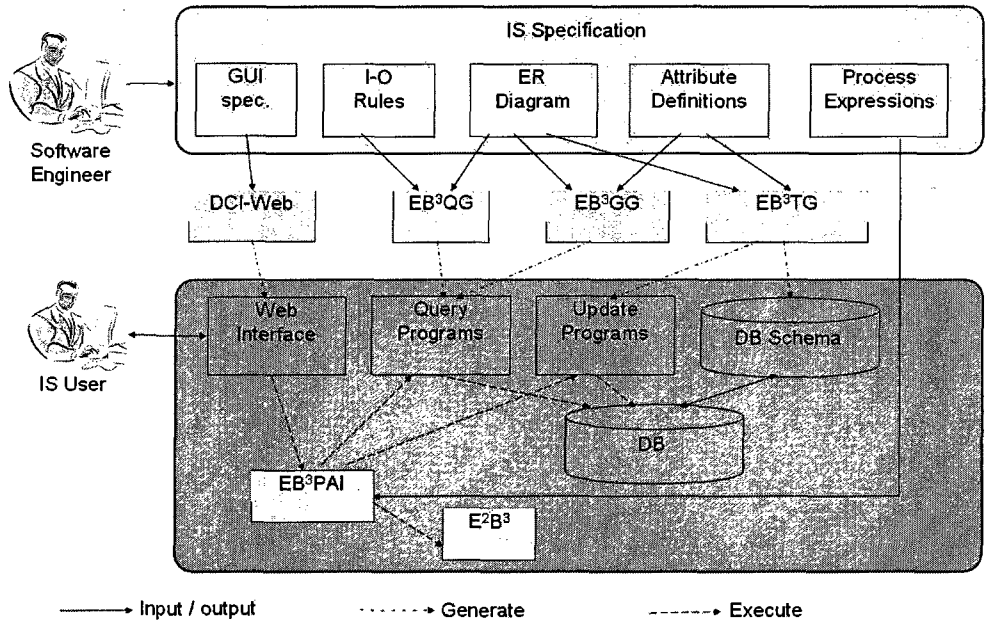


Figure 1.1 – Structure du projet APIS

1.2.1 DCI-WEB

Ce module permet de générer automatiquement des interfaces Web à partir des spécifications formelles de la GUI. Il peut être connecté à l'interpréteur d'expressions de processus EB^{3}PAI ou à tout autre système qui peut fournir des informations pour le contenu dynamique de la page ; par conséquent, DCI-WEB est indépendant de EB^3. Il génère à partir

d'une spécification les pages JSP, les formulaires d'actions *Struts* et les *Servlets* requis pour implémenter l'interface Web utilisateur.

1.2.2 EB^{3}TG

Une transaction dans une base de données relationnelle est générée par l'outil EB^{3}TG [1] pour chaque action du diagramme EA. La transaction est exécutée à chaque fois que l'événement correspondant est considéré comme valide par EB^{3}PAI, mais elle n'est générée qu'une seule fois par EB^{3}TG. C'est l'interpréteur EB^{3}PAI qui fait appel aux transactions quand c'est nécessaire. EB^{3}TG supporte la génération d'une base de données relationnelle à partir de la représentation XML du diagramme EA. Il analyse la définition d'attributs et génère un ensemble de classes qui implémentent les transactions en utilisant JDBC.

1.2.3 EB^{3}GG

EB^{3}GG [11] est un outil qui s'interface avec l'interpréteur EB^{3}PAI pour vérifier l'exécution des expressions de processus gardées. C'est un module entièrement dédié aux gardes dans EB^3.

1.2.4 EB^{3}PAI

L'interpréteur d'expressions de processus EB^{3}PAI développé par Benoît Fraikin [5] dans le cadre d'un doctorat en informatique à l'Université de Sherbrooke est le cœur de la plateforme APIS. Il peut interpréter efficacement toutes les expressions de processus d'un SI en utilisant la sémantique opérationnelle de l'algèbre de processus EB^3.

Chapitre 2

Définition d'un langage formel de requêtes

Résumé

Cet article présente la définition d'un langage formel de requêtes basé sur un modèle entité-association. Le principal concept du langage est les expressions de chemins. L'article expose la syntaxe et la sémantique de ce nouveau langage ainsi que les schémas de traduction en requêtes SQL des expressions de chemins.

Commentaires

Ma contribution au sein de cet article consiste en la définition du langage exposé, la réalisation des schémas de traduction vers des requêtes SQL ainsi que la rédaction du contenu. L'implémentation d'un outil appelé EB^{3}QG qui prend en charge ces schémas de traduction est aussi une de mes contributions.

Definition of a formal query language for entity-relationship models

Imad Yassine, Marc Frappier and Richard St-Denis
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
{Imade.Yassine,Marc.Frappier,Richard.St-Denis}@USherbrooke.ca

Abstract

This paper describes a formal query language based on the entity-relationship model, which provides users with high level, relational-like view of their data. This language allows one to express queries from path expressions, a concept that was introduced in the context of object-oriented databases. Translation rules are proposed to translate these queries into executable code. They are implemented by the tool EB^{3}QG, developed and integrated in the APIS platform dedicated to the automatic generation of information systems specified using EB^3 method.

2.1 Introduction

An information system (IS) coordinates information through the organization's activities and enables it to achieve its objectives. It is a major vehicle of communication in the organization. An IS is generally characterized by large persistent data structures which are modified or queried by several users in concurrency. An IS can be decomposed into three parts : the user interface, the business logic and the database [5]. The database is definitely the most mastered part.

It is commonly accepted today that conceptual design step plays a central role in process of designing a database of an IS [8]. To describe the requirements of database users in a formal and a complete manner, so called data models are needed. The widely accepted entity-relationship (ER) model [3] is often considered the most appropriate data model, since it allows one to capture most of the important phenomena of the real world and express them in natural and easily understandable way. Query languages based on the ER approach have been proposed, like ERQL [12], which is claimed to be more natural than relational query languages. Nevertheless it lacks a formal semantics.

Our intended contribution in this paper is to present a new query language based on path expressions for ER models. Similar path expressions have previously been used for navigating in object-oriented database [2, 10]. Our language is characterized by the facts that it only uses abstract concepts that appear in the ER model, that is does not depend on any implicit relational representation of the ER schema, and that it has a simple syntax and semantics. We also define rules that translate the ER queries defined by our formal language into SQL queries, following a predefined (conventional) representation of the ER model into a relational model. These rules are implemented in the tool called EB^{3}QG.

2.2 State of the Art

2.2.1 Relational Algebra

The relational algebra is often considered to be an integral part of the relational data model [4]. Its operations can be divided into two groups. One group includes set operations from mathematical set theory. Set operations include union, intersection and difference. The

other group consists of operations specific to relation algebra, namely Cartesian product, projection and join. These operations permit user to specify queries. The result of these queries is a new relation formed from one or more relations [4].

We see that the relational algebra is very important for several reasons. The first one, it provides a formal basis for relational model operation and some of its concepts are incorporated into SQL. The second reason, it is used as a basis for implementing queries in relational database management systems (DBMS) [4].

2.2.2 Relational Calculus

The relational calculus consists of two calculi, the tuple relational calculus and the domain relational calculus. It provides a declarative way to specify database queries in contrast to the relational algebra which provides a more procedural way for specifying queries. The main distinguishing feature between relational algebra and relational calculus, is in a calculus expression, there is *no order of operations* to specify how to retrieve the query result ; it specifies only what information the result should contain [4]. Thus, it takes its foundation from mathematical logic. The most popular query language SQL has some of its foundations in the tuple relational calculus.[1]

2.3 The ER model

The ER model [3] describes data as *entities*, *relationships* and *attributes*. It is an abstract data model which is widely used for the analysis of user requirements and the conceptual design of databases. It is supported by several industrial tools. Nowadays, an ER model is often expressed using the class diagram notation of UML. Figure 2.1, taken from [4] and written using Chen's original notation, illustrates all the concepts of an ER model.

The ER model differs from the relational model in several ways. The former distinguishes between entities and relationships, whereas the latter considers only relations. Connections between entities and relationships are represented by the join operator and foreign keys in a relational model. ER attributes (of both entities and relationships) can be

[1] SQL is based on the tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions [4].

of structured types like aggregation, set and list, whereas the relational model only considers atomic types. There are several ways of representing an ER model into a relational model [4]. In any case, the corresponding relational model constitutes a flattened representation, closer to an implementation, since structured attributes are represented by relations containing only atomic types, and entities and relationships are represented using relations, loosing the visual structure of the associations and the distinctions between entities and relationships.

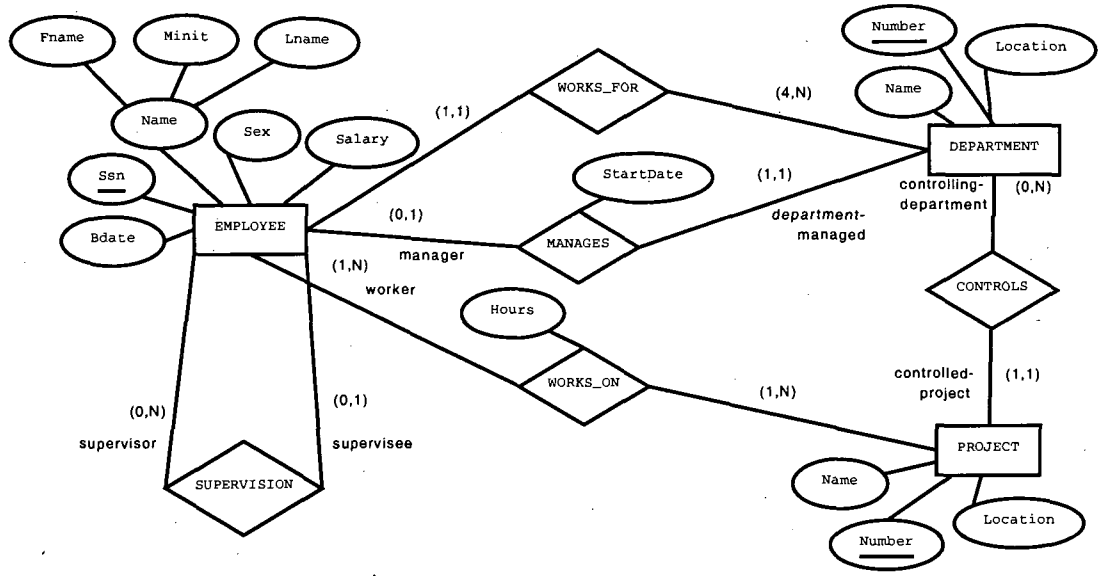


Figure 2.1 – ER model for a company

2.3.1 Entities and Attributes

The ER model represents an **entity** as a basic object, which may be defined as a thing recognized as being capable of an independent existence and being uniquely identified. An entity may be a physical object such as a house or a car, or a concept such as a transaction or an order. Each entity has **attributes** that describe his properties. The attribute values become a part of the data stored in the database [4].

2.3.2 Relationships and Roles

A relationship illustrates how two or more entities are related one to each other. It describes the associations among entities. According to the example in Figure 2.1, let see the relationship WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which he works. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity to one DEPARTMENT entity [4].

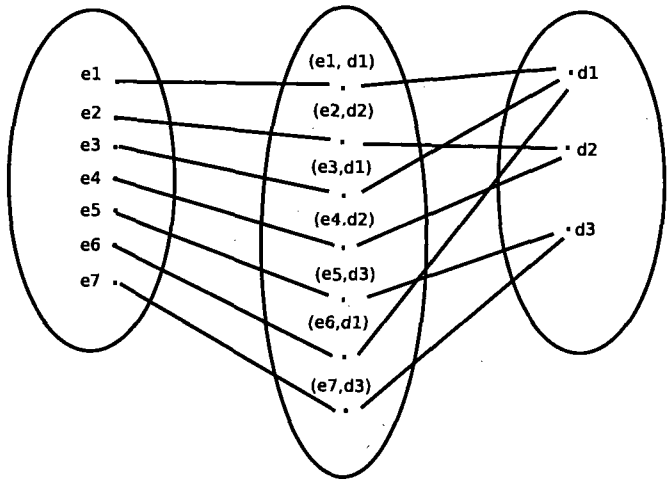


Figure 2.2 – Some instances in the WORKS_FOR relationship

When an entity participates in a relationship it may play a particular role. The *role name* become more important when the entity participates more than once in a relationship, because it describes what the relationship means [4] as illustrated in Figure 2.2.

2.4 Formal Language Description

Our query language is a simple, powerful query language for the ER model. It simplifies the task of expressing queries to a database by using only the abstract concepts defined in the ER model. It hides the implementation details and provides the user with clearer understanding. Our query language uses path expressions for navigation in ER models. We

start by illustrating the language using a small example, then we provide the syntax and semantics.

2.4.1 Example

The following is a simple example of a query on the ER diagram of Figure 2.1. It returns the employees that work for some department x earning more than 50 K$.

```
listEmployeeDept(x : int) ::=
    d.number, d.name, employee.lname, employee.salary
  where
    department[d]{d.number = #x}.
    works_for.
    employee{employee.salary > 50000}
```

The first line denotes the declaration of the query name with its parameters. The second line gives the attributes returned by the query. Symbol `d` is an alias for entity type **department**; it is declared in the `where` clause using `[d]`. The `where` clause is a path in the ER model. A path is constructed using operator "." for concatenation of path elements. A path element can be an entity type name or a relationship type name. A role name can also be used instead of a relationship type name; this is especially useful when an entity type occurs more than once in a relationship (*e.g.* in recursive relationships like SUPERVISION in Figure 2.1), in order to determine the direction. Path elements can be decorated with an alias name enclosed between brackets and a selection condition enclosed between braces. A condition can refer to any attribute of any path element and to parameters decorated with #. The semantics of a query is a set of tuples with the requested attributes. The tuples are constructed from the entities related by relationship instances and satisfying the selection conditions. We shall make this precise using a relational semantics.

2.4.2 Syntax

Query Grammar

The following provides the BNF syntax of a query. As BNF conventions, "|" separates alternatives, ^{*b} and $^{+b}$ respectively denote 0 or more, and one or more occurrences separated

by ♭, "?" denotes an optional expression, and parentheses can be used to group syntactic expressions. Terminal symbols are either `keywords`, other symbols enclosed between simple quotes (*e.g.* '['), or *identifiers*.

QueryDecl	::=	*queryName* Parameters ':==' Query
Parameters	::=	'(' (*parameterName* : *type*)*, ')'
Query	::=	Attributes `where` Path
	|	Query (`union` | `intersection` | `minus`) Query
	|	'(' Query ')'
Attributes	::=	(*ERNameOrAlias* '.' *attribute*)$^{+,}$
Path	::=	PathElement$^{+\cdot}$
	|	Path '+'
	|	'(' Path ')'
PathElement	::=	ERName Alias ? Selection ?
ERName	::=	*entityType* | *relationshipType* | *role* | '<' *role* ',' *role* '>'
Alias	::=	'[' *alias* ']'
Selection	::=	'{' Condition '}'

The definition of Condition is omitted, for the sake of concision. It follows the usual SQL syntax for conditions.

Path expressions

We now provide more details on the definition of paths. Path expressions are inductively defined. An elementary path expression is defined by the BNF production rule with the rhs PathElement, where *ERName* is either an entity type, a relationship type or a role name or a pair of role names. Compound path expressions are constructed using concatenation (".") and transitive closure ("+"). In the sequel, we use $e^\sharp$, $r^\sharp$ and $rol^\sharp$ to respectively denote entity, relationship and role elementary path expression. An ER diagram can be seen as a bipartite graph where nodes are partitioned into entity types and relationship types. A path expression is essentially a path $\cdots e_1.a_1.e_2.a_2.\cdots$ of that graph, where e_i denotes an entity type and a_i denotes a relationship type. A path can start with either an entity type or a relationship type. When an entity type participates more than once in a relationship type (*e.g.* entity type EMPLOYEE for relationship type SUPERVISION in

Figure 2.1), then the role name must be used instead of the relationship type to disambiguate the navigation direction : for instance, instead of writing $e_1.a.e_2$, one must write $e_1.r.e_2$;if e_2 also participates more than once in a, then one must write $e_1.<r_1, r_2>.e_2$. In the sequel, to simplify our definitions, we assume that each relationship type of a path is rewritten as a normal form $e_1.<r_1, a, r_2>.e_2$.

Let $last(c)$ denotes the last element of a path such that $last(\cdots .x^\sharp) = x$ and $last(c^+) = last(c)$. Dually, $first(c)$ denotes the first element of a path such that $first(x^\sharp.\cdots) = x$ and $first(c^+) = first(c)$.

Definition 2.4.1 ***Valid Concatenations*** *Let $e_1 - r_1 - a - r_2 - e_2$ denote two entity types e_1 and e_2 associated by a relationship type a with roles r_1 and r_2, as illustrated in Figure 2.3.*

1. *If $last(c) = e_1$ and e_1 participates only once in a, then $c.a^\sharp$ and $c.r_2^\sharp$ are path expressions.*
2. *If $last(c) = r_2$ then $c.e_2^\sharp$ is a path expression.*
3. *If $last(c) = a$ and e_2 participates only once in a, then $c.e_2^\sharp$ is a path expression.*

For instance, the following are valid concatenations.

```
 1: supervisor
 2: supervision
 3: supervisee
 4: employee
 5: supervisor.employee
 6: employee.supervisor
 7: employee.supervisor.employee[b].works_for
 8: employee.supervisee.employee[e].works_for
 9: employee.works_on.project.controls.department
10: employee.supervisor+.employee[e]
```

As we shall see in the semantics, the first three paths are equivalent in content. Path 4 returns all the employees and Path 5 returns the supervisor of an employee ssn, while path 6 returns an employee and his supervisor ssn. The distinction between the two is the list of attributes made available by the path expressions. The attributes included in a path

are only the attributes of the path elements. Path 5 returns all the attributes of supervisors from entity type EMPLOYEE and the ssn of the supervised employees, while path 6 returns all the attributes of supervised employees from entity type EMPLOYEE and the ssn of the supervisor. Thus, they both return the same set of employees and supervisors, but with different sets of attributes. Paths 7 and 8 are "oriented" differently, so that path 7 returns the department id of the supervisor of an employee, while path 8 returns the department ids of the supervised employees of a supervisor. Attribute `b.name` denotes the name of a supervisor, while `e.name` denotes the name of a supervised employee. Path 10 illustrates a recursive query which returns the employees with their supervisors, transitively. Hence, the company CEO appears once for each employee, since the CEO transitively supervises each employee. However, the CEO does not appear as an employee, since he/she has no supervisor.

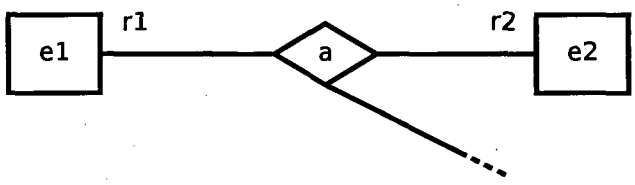


Figure 2.3 – Graphical representation of the relationship $e_1 - r_1 - a - r_2 - e_2$

A transitive closure can only be applied to a path that starts from a relationship type or a role and ends with a relationship type or a role, and these start and end elements must be connected to the same entity. For instance, `supervisor+` denotes all the hierarchical supervisors of an employee in Figure 2.1, *i.e.* his immediate supervisor, the supervisor of his supervisor, and so on, up to the organization CEO.

Definition 2.4.2 ***Valid Transitive Closure*** *If $e.c.e$ is a path expression such that $first(c)$ and $last(c)$ are role names, then c^+ is a path expression.*

Note that e is not included in the scope of the transitive closure operator; entity type e is simply used to ensure that a closure is applied to a path that originates and ends with relationship elements connected to the same entity type. Note also that, as a syntactic sugar, a relationship type could also be used as a starting element or an ending element of a

closure, as long as there is a unique e such that, respectively, $e.c$ or $c.e$, are valid path expressions. The relationship type can be rewritten with its corresponding role name for the entity in the relationship type. A closure c^+ only offers the key attributes of e, renamed with the corresponding starting and ending role names.

Finally, if an entity type, a relationship type or a role name occurs more than once in a path expression, then aliases must be used to distinguish these occurrences and ensure uniqueness of path element names.

2.4.3 Semantics

Preliminary Definitions

Let μ stand for the interpretation function, which maps a syntactic element to a mathematical structure. It is inductively defined. Path expressions are interpreted as relations of a relational algebra. The basis of the semantics is the interpretation of entity types, relationship types and roles, which are mapped to relations. We use $\mu(e)$ to denote the set of tuples representing the entities of entity type e. In essence, $\mu(e)$ denotes the state of entity type e, *i.e.* the set of entities that exists at some given point in time. A tuple t is noted $\langle v_1, \cdots, v_m \rangle$, where each v_i corresponds to the value of attribute b_i of e in the ER diagram. Each entity type e has a primary key, noted $key(e)$, which is a list of attributes of e. The value of a list of attributes k for a given tuple t is noted $k(t)$, which is a sub tuple of t. Similarly, a relationship type a between entity types $e_1, \cdots, e_n$ with roles $r_1, \cdots, r_n$ is interpreted by a relation. Its attributes are the relationship attributes and the keys of the participating entity types $e_1, \cdots, e_n$. Hence, $\mu(a)$ has attributes $key(e_1), \cdots, key(e_n), b_1, \cdots, b_n$, where b_i denote an attribute of a in the ER diagram. The sublist of attributes $key(e_i)$ of a under role r_i is noted $key(r_i)$. The parameterized join $\bowtie(u_1, u_2, k_1, k_2)$ between two relations u_1 and u_2 over attribute lists k_1 and k_2 is defined as follows.

$$\bowtie(u_1, u_2, k_1, k_2) = \{\langle t_1, t_2 \rangle \mid t_1 \in u_1 \wedge t_2 \in u_2 \wedge k_1(t_1) = k_2(t_2)\}$$

For the sake of simplicity, tuples are flattened, so that

$$\langle \langle v_1, \cdots, v_m \rangle, \langle w_1, \cdots, w_n \rangle \rangle = \langle v_1, \cdots, v_m, w_1, \cdots, w_n \rangle \ .$$

The projection of a relation r on a list of attributes k is noted $\pi(r, k)$. The concatenation of two lists k_1 and k_2 is simply noted (k_1, k_2). The transitive closure of a relation r with attributes (k_1, k_2) such that k_1 and k_2 are of the same type, is noted r^+ and defined as follows

$$\begin{aligned} r^+ &= \bigcup_{n\geq 1} r^n \\ r^1 &= r \\ r^{n+1} &= r \circ r^n \\ r_1 \circ r_2 &= \{\langle k_1(t_1), k_2(t_2)\rangle \mid t_1 \in r_1 \wedge t_2 \in r_2 \wedge k_2(t_1) = k_1(t_2)\} \end{aligned}$$

Note that each r_i in the above definition has attributes (k_1, k_2).

Semantics of Path Expressions

Let $e_1{-}r_1{-}a{-}r_2{-}e_2$ denote two entity types e_1 and e_2 associated by a relationship type a with roles r_1 and r_2, as illustrated in Figure 2.3. We assume that roles and relationship types are rewritten in a normal form $<r_1, a, r_2>$. We assume that a path element $x[\ldots]$ without selection condition is rewritten as $x[\ldots]\{\textbf{true}\}$. Moreover, we let $\phi(t)$ denote the condition ϕ evaluated with the attribute values of t. We suppose that each path element is uniquely identified using either its name or alias. The semantics of a path expression is defined as follows.

$$\begin{array}{rcll} & & \mu(x[\nu]\{\phi\}) & = & \{t \mid t \in \mu(x) \wedge \phi(t)\} \\ & & \mu(c^+) & = & \mu(c)^+ \\ last(c) = <r_1, a, r_2> & \Rightarrow & \mu(c.e) & = & \bowtie(\mu(c), \mu(e), key(r_2), key(e)) \\ & & \mu(c.<r_1, a, r_2>) & = & \bowtie(\mu(c), \mu(a), key(last(c)), key(r_1)) \\ first(c_2) = <r_1, a, r_2> & \Rightarrow & \mu(c_1.c_2^+) & = & \bowtie(\mu(c_1), \mu(c_2^+), key(last(c_1)), key(r_1)) \end{array}$$

Semantics of Queries

The semantics of a query is defined as follows.

$$\begin{aligned}
\mu(k \texttt{ where } c) &= \pi(\mu(c), k) \\
\mu(q_1 \texttt{ union } q_2) &= \mu(q_1) \cup \mu(q_2) \\
\mu(q_1 \texttt{ intersection } q_2) &= \mu(q_1) \cap \mu(q_2) \\
\mu(q_1 \texttt{ minus } q_2) &= \mu(q_1) - \mu(q_2)
\end{aligned}$$

2.5 Generating Queries

2.5.1 General Principle

We propose rules to translate our ER queries into Java and SQL code. These rules are implemented in a tool called EB^{3}QG. It takes as input the ER model (represented in XML) and the ER queries. It first performs a lexical and syntactic analyses of ER queries and constructs an abstract syntax tree (AST) representing them. This AST is then transformed into an object representation, after performing static and semantical analyses. An SQL query object is then constructed to implement an EB^{3}QG query, following the translation rules described in this section. The ER model is translated into a relational model using EB^{3}TG, another tool from the APIS framework. The generated relational model conforms to the semantics described in Section 2.4.3.

2.5.2 Translation Rules

This section describes the generation of a SELECT statement corresponding to an ER query. Let S denote a select statement. We use S_s, S_f and S_w to respectively denote the SELECT, FROM and WHERE clauses of this select statement. The algorithm is structured according to the syntax of ER queries. The main algorithm, called *transSetOP*, deals with the set-theoretic operators `union`, `intersection` and `minus`, which can be directly implemented in SQL using the same operators. It calls *transElemQuery* to translate an elementary query into an SQL select statement. We use a simple intuitive pseudo syntax for SQL statements which should be clear from the context.

Algorithm 2.5.1 $transSetOP(q)$

Description : *Return an SQL statement evaluating query q*

Input q **:** *An ER query.*

Output : *The SELECT statement that computes q.*

let $transSetOP(q)$ = ***match*** q ***with***

q' `union` q''	$\rightarrow$	$transSetOP(q')$ ***union*** $transSetOP(q'')$
q' `intersection` q''	$\rightarrow$	$transSetOP(q')$ ***intersection*** $transSetOP(q'')$
q' `minus` q''	$\rightarrow$	$transSetOP(q')$ ***minus*** $transSetOP(q'')$
q'	$\rightarrow$	$transElemQuery(q')$

Algorithm $transElemQuery$ is responsible for constructing a SELECT statement. It is divided into three parts : SELECT clause, FROM clause and WHERE clause. A FROM clause is represented by a list of table names ; we use the function *concat* to build this list ; similarly, a SELECT clause is represented by a list of attributes.

Algorithm 2.5.2 $transElemQuery(q)$

Description : *Generate SELECT statement for an elementary ER query.*

Input q **:** *An elementary ER Query.*

Output : *The SELECT statement computing q.*

$transElemQuery(q)$
begin
  let R *and* P *be such that* $q = R$ `where` P
  let S *be a SELECT Statement*
  $S.s := \varepsilon$
  foreach b ***in*** R ***do***
    $S_s := concat(S_s, b)$
  done
  $S_f := transFromClause(P)$
  $S_w := transWhereClause(P)$
  return S
end

Algorithm $transFromClause$ generates the FROM clause and also generates temporary tables for the computation of transitive closures. Variable o denotes an elementary path. Variables c and c' denote paths. Function $table(o)$ returns the table name and alias of an elementary path element. Function $transClosure(c)$ generates a temporary table that holds the result of the evaluation of the transitive closure of path c. Note that joins between tables will be expressed in the WHERE clause.

Algorithm 2.5.3 $transFromClause(p)$

Description : *Generate the FROM clause of a select statement for a path.*

Input p **:** *A path of an ER query.*

Output : *The FROM clause for computing the path* p.

let $transFromClause(p)$ = ***match*** p ***with***

$o \quad \rightarrow \quad table(o)$

$c^+ \quad \rightarrow \quad table(transClosure(c))$

$c'.c^+ \quad \rightarrow \quad concat(transFromClause(c'), table(transClosure(c))$

$c.o \quad \rightarrow \quad concat(transFromClause(c), table(o))$

Algorithm $transWhereClause$ constructs the WHERE clause of a select statement. The WHERE is represented by a condition. Here, q, q' denote either an elementary path or a closure. Function $join(table(q), table(q'))$ is used to denote the SQL condition of the parameterized join $\bowtie(\ldots)$, following the semantics given in Section 2.4.3. Hence, if o is an entity with key $b_1, \ldots, b_n$ and o' is a role with attributes $d_1, \ldots, d_n$, then the join condition is simply $b_1 = d_1$ AND ... AND $b_n = d_n$.

Algorithm 2.5.4 $transWhereClause(p)$

Description : *Generate WHERE clause of a select statement for a path.*

Input p **:** *A path of an ER query.*

Output : *The WHERE clause computing the path* p.

let $transWhereClause(p)$ = ***match*** p ***with***

$q.q' \quad \rightarrow \quad join(table(q), table(q'))$

$c.q \quad \rightarrow \quad transWhereClause(c)$ *AND* $join(table(last(c)), o)$

Algorithm $transClosure(p)$ generates a *program* to compute the transitive closure of a path. As such, it does not return a single select statement. It updates the global variable $tempTable$, which is a map that associates to a transitive closure path a temporary SQL table. This temporary table is initialized with the result of executing the query corresponding to p. Then, the program iterates to compute each power of p, until a fix point is reached. It uses an insert statement over a join between the temporary table and the query of p. When no record has been inserted, the loop stops ; the local variable i holds the number of records inserted in the temporary table.

Algorithm 2.5.5 $transClosure(p)$

Description : *Generate a temporary table to hold the result of a SELECT statement for transitive closure of path expression.*

Input p **:** *A Path of an ER query that represents a transitive closure.*

Output : *The sequential program to compute a temporary table of the closure of* p.

The following program is returned.

$tempTable[p] := transElemQuery(p)$;
repeat
  $i :=$ *insert into* $tempTable[p]$
    select $t1.leftRole(tempTable[p])$, $t2.rightRole(transElemQuery(p))$
    from $tempTable[p]$ *as t1,* $(transElemQuery(p))$ *as t2*
    where $join(tempTable[p], transElemQuery(p))$ *AND*
      *not exists (select * from* $tempTable[p]$ *as t3 where*
      $t1.leftRole(tempTable[p])$ = *t3.*$leftRole$ *AND*
      $t2.rightRole(transElemQuery(p))$ = *t3.*$rightRole$*)*
until i = 0

2.6 Case Study

In this section, we present a case study to illustrate the use of our language to define the user queries from path expressions. We also present the SELECT statements generated from these path expressions, by the tool EB^{3}QG, based on the translation schema described

in the Section 2.5. For this example, we set in the case of a system for managing a company. The ER model on which we base our example is described in the Section 2.3.

The first query (Query1) illustrates the use of logic predicates. It returns the information of male employees working in the department "Research" or "Administration" and having a salary between 20 000 and 40 000 dollars. After the application of translation schema, we obtain the SELECT statement of Query1 illustrated in Figure 2.4.

```
Query1() ::=
e.*
WHERE
employee[e]{((e.salary >= 20000 OR e.salary <= 40000)
AND (e.sex = "M"))}.works_for.departement[d]
{((d.name = "Research") OR (d.name = "Administration"))};
```

```
1  SELECT e.*
2  FROM   employee AS e,works_for,departement AS d
3  WHERE  e.ssn=works_for.ssn
4    AND  works_for.dnumber=d.dnumber
5    AND  ((e.salary >= 20000 OR e.salary <= 40000)
6    AND  e.sex = 'M')
7    AND  (d.name = 'Research' OR d.name = 'Administration') ;
```

Figure 2.4 – The SQL statements of Query1

The second query (Query2) accepts a parameter which is the employee ssn. It allows us to find the name and address of all employees involved in projects controlled by the department managed by the employee whose ssn is pssn. This query illustrates a path that passes twice through the same entity, the use of aliases and the use of roles in path expressions. The Figure 2.5 shows the SELECT statement corresponding to Query2.

```
Query2(pssn : numeric) ::=
employee.fname,employee.lname,employee.ssn
WHERE
employee.works_on.project.controlling_department
department.manages.employee[e]{e.ssn = #pssn};
```

```
1   SELECT employee.fname,employee.lname,employee.ssn
2   FROM   employee,works_on,project,controls,department,
3          manages,employee AS e
4   WHERE  employee.ssn=works_on.ssn
5     AND  works_on.pnumber=project.pnumber
6     AND  project.pnumber=controls.pnumber
7     AND  controls.controlling_department=department.dnumber
8     AND  department.dnumber=manages.department_managed
9     AND  manages.ssn=e.ssn
10    AND  e.ssn = ?
```

Figure 2.5 – The SQL statements of `Query2`

The third query `(Query3)` illustrates the use of set operations, particularly the operation of intersection. It allows to find employees who work in both projects « Alpha » and « Beta ». The Figure 2.6 shows the SELECT statement corresponding to `Query3`.

```
Query3() ::=
(e.*
WHERE
project[p]{p.pname = "Alpha"}.works_on.employee[e])
intersection
(e.*
WHERE
project[p]{p.pname = "Beta"}.works_on.employee[e]);
```

The last query `(Query4)` illustrates the use of a transitive closure in a path expression. This query returns the direct and indirect supervisors of the employee with the name 'Franklin Wong'. The Figure 2.7 shows the pseudo code of the SELECT statement corresponding to `Query4`.

```
1   SELECT e.*
2   FROM   project AS p,works_on,employee AS e
3   WHERE  p.pnumber=works_on.pnumber
4     AND  works_on.ssn=e.ssn
5     AND  p.pname = 'Alpha'
6   INTERSECT
7   SELECT e.*
8   FROM   project AS p,works_on,employee AS e
9   WHERE  p.pnumber=works_on.pnumber
10    AND  works_on.ssn=e.ssn
11    AND  p.pname = 'Beta';
```

Figure 2.6 – The SQL statements of `Query3`

```
Query4() ::=
employee.*
WHERE
employee[e]{e.lname = "Wong" AND e.fname="Franklin"}
.supervisor+.employee;
```

```
S_RT : SELECTStatement corresponding to a no recursive term.
S_WT : SELECTStatement corresponding to a recursive term.
RT   : an intermediate table.
WT   : a working table.

RT :=   INSERT INTO RT
        SELECT supervision.*
        FROM   employee AS e,supervision
        WHERE  e.ssn=supervision.supervisee
        AND  (e.fname = 'Franklin' AND e.lname = 'Wong');
        WT := RT
        while !empty(WT) do
         WT :=  INSERT INTO WT
                SELECT supervision.*
                FROM   WT,supervision
                WHERE  WT.supervisee=supervision.supervisor;
         RT := WT UNION  RT
         done
SELECT employee.*
FROM   RT,employee
WHERE  RT.supervisor=employee.ssn;
```

Figure 2.7 – The pseudo code of `Query4`

2.7 Conclusion

We have defined through this paper the syntax and semantics of a formal query language based on the ER model. It is based on the concept of navigation in the ER model by path expressions. We also proceeded to the definition of translation schema of path expressions into SQL queries. These translation schema construct the SELECT statement corresponding to a user query. The tool EB^{3}QG that automates the implementation of path expressions from translation schema was developed and integrated in the APIS platform.

Similar query language for EER schema has been defined by other researchers [12], but their semantics is more complex. In contrast, the semantics of our query language is given in a recursive manner in terms of path expressions.

Although our query language provides a good balance between simplicity and expressiveness, it represents a subset of the SQL language since it does not have all the features provided in SQL. Thus, it does not express any possible SQL query. Another limitation of our language is that it is only based on the concepts defined in the ER model [3] and does not take into account the new additions of the extended entity-relationship model (EER).

Among the future work, a first step would be to incorporate in our query language all the functions and operations supported by the SQL language. Another application of such a language is to be used as a query language for the EER model. Its extension to an updating language and its inclusion in a host application is another area of future research.

Chapitre 3

Implémentation

Dans ce chapitre, nous nous intéressons au fonctionnement de l'outil EB^{3}QG qui implémente les schémas de traduction des expressions de chemins. Nous présentons aussi son architecture et son intégration au sein de la plateforme APIS.

3.1 L'outil EB^{3}QG

L'outil EB^{3}QG implémente les schémas de traduction des expressions de chemins en requêtes SQL. Il utilise tout d'abord un analyseur syntaxique, créé à l'aide de l'outil ANTLR, pour réaliser une analyse des expressions de chemins et obtenir un arbre syntaxique abstrait. Puis, il réalise la phase d'analyse sémantique et obtient une représentation objet des expressions de chemins. Ensuite, EB^{3}QG réalise la transformation de la représentation objet obtenue vers du code exécutable composé de Java et d'énoncés SQL. Le principe de fonctionnement du module EB^{3}QG est schématisé à la Figure 3.1.

L'outil prend en entrée un fichier (Figure 3.2) qui contient la définition des requêtes de sortie à partir d'expressions de chemins et fournit en sortie un fichier (Figure 3.3) qui contient une classe Java avec une méthode pour chaque requête. Le nom de la méthode correspond au nom de la requête dans le fichier d'entrée. De plus, les signatures des méthodes Java précisent que leur type de sortie est de type *ResultSet*.

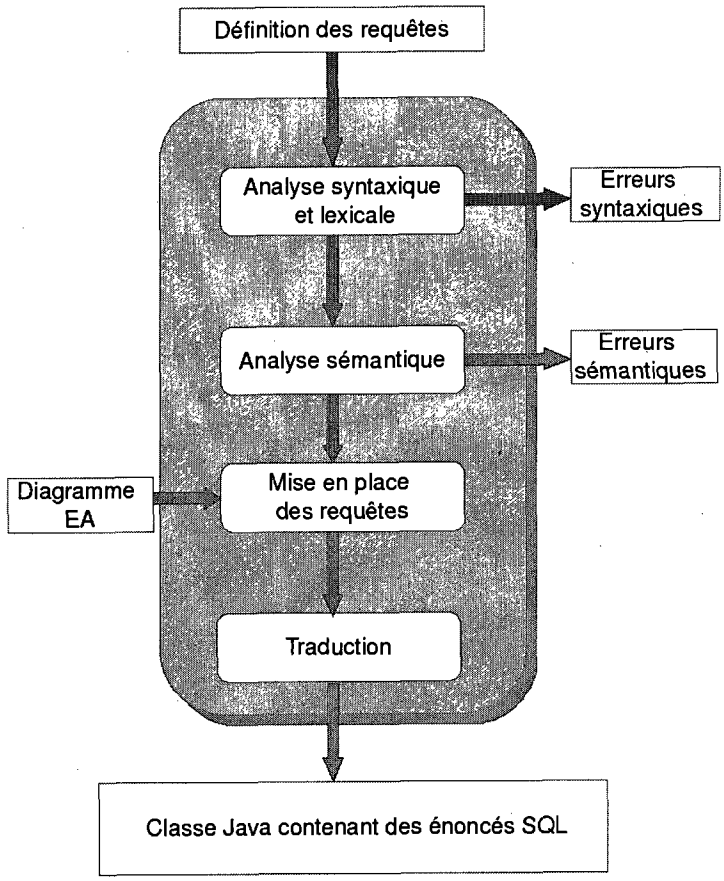


Figure 3.1 – Fonctionnement de l'outil EB^{3}QG

3.2 Architecture de EB^{3}QG

L'outil est composé d'une classe principale. Elle assure de faire transiter la représentation objet de la requête d'une classe à l'autre. Chacune des étapes de traduction est implémentée par une classe spécifique. La classe qui réalise l'analyse syntaxique a été créée grâce à l'outil ANTLR [14] qui accepte la grammaire hors contexte d'une définition de requêtes. La classe qui réalise la génération des requêtes SQL s'appuie sur le modèle EA [3]

```
1 query_1(var_1:type_1,...,var_n:type_n) :
2         SELECT_Statement ;
3
4 query_2(var_1:type_1,...,var_n:type_n) :
5         SELECT_Statement ;
6  ...
7
8 query_n(var_1:type_1,...,var_n:type_n) :
9         SELECT_Statement ;
```

Figure 3.2 – Exemple de fichier contenant des requêtes

de la spécification EB^3. Pour réaliser ce travail, une partie de l'outil EB^{3}TG a été réutilisée ; il s'agit du module ER2SQL. Ce module réalise la traduction du modèle EA vers la base de données. Il a donc servi de base pour l'interrogation du modèle EA afin de réaliser la mise en place des requêtes SQL.

3.3 Intégration dans la plateforme APIS

L'outil EB^{3}QG permet de créer un fichier Java (Figure 3.3) qui contient les méthodes correspondantes aux requêtes de sortie associées aux règles d'ES. En effet, lorsque l'interpréteur EB^{3}PAI valide une action, il analyse le fichier de spécification des règles d'ES et vérifie si une requête de sortie est associée à cette action, si c'est le cas, il appelle la méthode correspondante de la requête.

```
1  import java.sql.*;
2  import org.apache.log4j.Logger;
3
4  public class Queries {
5     SimpleDateFormat dateFormat = new SimpleDateFormat
6                                   ("dd/MM/yy HH:mm:ss");
7     private Connexion connection;
8     static Logger logger = Logger.getLogger(Queries.class);
9     public void connect(String serverAddress,
10          int port, String dbName,
11          String user, String pass)
12    {
13       try
14       {
15          connection = new Connexion(serverAddress,
16                port, dbName, user, pass);
17       }
18       catch (SQLException e)
19       {
20          logger.error(e.getMessage());
21       }
22    }
23
24    public Queries(String serverAddress, int port, String dbName
25                ,String user, String pass)
26 {
27    super();
28    connect(serverAddress,port,dbName,user,pass);
29    try
30    {
31    ...
32    }
```

Figure 3.3 – Exemple de fichier contenant l'implémentation des requêtes

```
1  catch(SQLException e)
2      {
3          logger.error(e.getMessage());
4      }
5  }
6
7  public ResultSet query1( double mId)
8  {
9      ResultSet result;
10     try
11     {
12     ...
13     }
14     catch (Exception e)
15     {
16         logger.error(e.getMessage());
17     }
18     return result ;
19 }
20
21 public ResultSet query2( )
22 {
23     ResultSet result;
24     try
25     {
26     ...
27     }
28     catch (Exception e)
29     {
30         logger.error(e.getMessage());
31     }
32     return result;
33 }
34
35 }
```

Figure 3.3 – Exemple de fichier contenant l'implémentation des requêtes (suite)

Chapitre 4

Étude de cas

Dans ce chapitre, nous présentons une étude de cas qui permet d'illustrer la génération de requêtes SQL par l'outil EB^{3}QG en s'appuyant sur les schémas de traduction décrits dans le chapitre 2. Pour cette exemple, nous nous plaçons dans le cas d'un système de gestion d'une compagnie. Le modèle EA sur lequel nous basons notre exemple figure dans le livre [4]. Ce modèle est illustré dans la Figure 4.1.

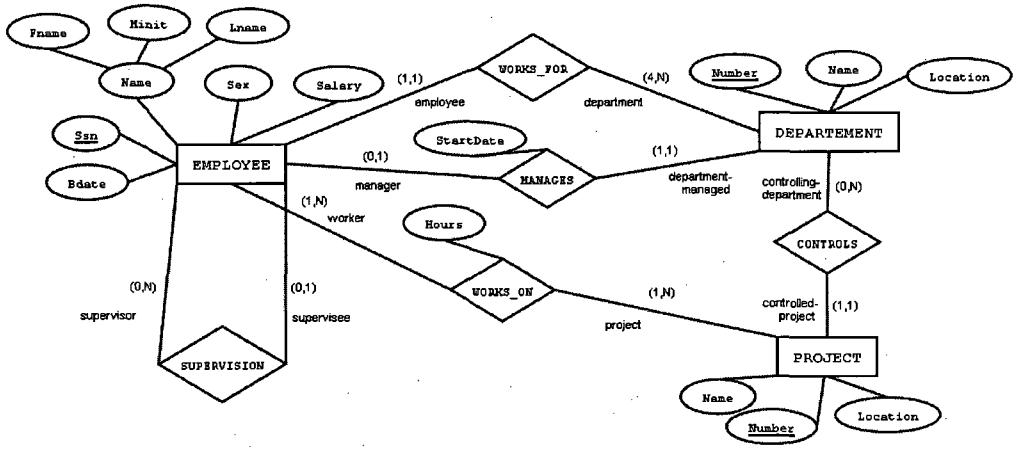
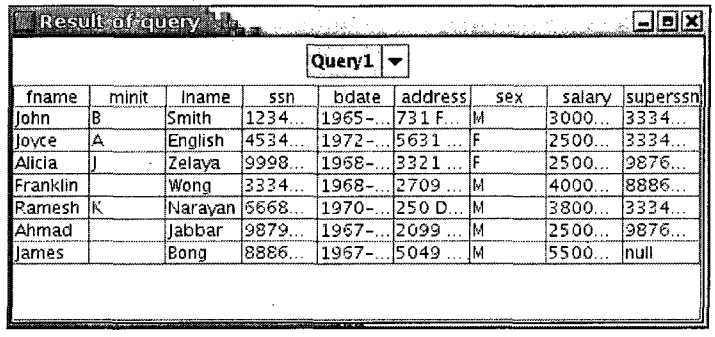


Figure 4.1 – Diagramme ER de la compagnie

Dans ce chapitre, nous allons nous intéresser aux requêtes suivantes.

- La première requête (`Query1`) retourne toutes les informations des employés travaillant dans le département « Research » ou « Administration » et ayant un salaire de plus que 20 000 dollars. Le résultat de cette requête est illustré dans la Figure 4.2.

```
Query1() ::=
e.*
WHERE
employee[e]{e.salary > 20000}.works_for.departement[d]
{d.name = "Research" OR d.name = "Administration"};
```



fname	minit	lname	ssn	bdate	address	sex	salary	superssn
John	B	Smith	1234...	1965-...	731 F...	M	3000...	3334...
Joyce	A	English	4534...	1972-...	5631 ...	F	2500...	3334...
Alicia	J	Zelaya	9998...	1968-...	3321 ...	F	2500...	9876...
Franklin		Wong	3334...	1968-...	2709 ...	M	4000...	8886...
Ramesh	K	Narayan	6668...	1970-...	250 D...	M	3800...	3334...
Ahmad		Jabbar	9879...	1967-...	2099 ...	M	2500...	9876...
James		Bong	8886...	1967-...	5049 ...	M	5500...	null

Figure 4.2 – Résultat de la requête Query1

- La deuxième requête (`Query2`) illustre l'utilisation des prédicats logiques. Elle permet de retourner les projets sur lesquels les employés de sexe masculin travaillent plus de 40 heures et gagnent entre 20 000 et 45 000 dollars. Le résultat de cette requête est illustré dans la Figure 4.3.

```
Query2() ::=
p.*
WHERE
project[p].works_on[w]{w.hours > 40}.employee[e]
{((e.salary >= 20000 OR e.salary <= 45000) AND (e.sex = "M"))};
```

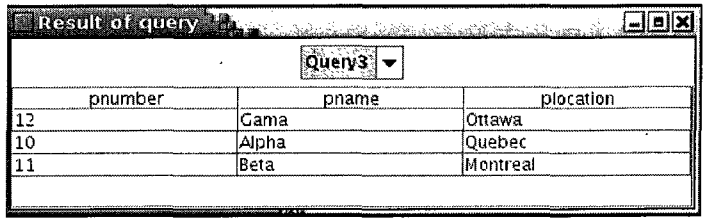


Figure 4.3 – Résultat de la requête Query2

- La troisième requête (`Query3`) accepte un paramètre qui est le numéro d'employé (`pssn`). Elle permet de retrouver le nom et l'adresse de tous les employés qui participent aux projets contrôlés par le département géré par l'employé ayant comme numéro d'employé (`pssn`). Cette requête illustre un chemin qui passe deux fois par la même entité et l'utilisation d'alias. Le résultat de cette requête est illustré dans la Figure 4.4.

```
Query3(pssn : numeric) ::=
employee.fname,employee.lname,employee.ssn
WHERE
employee.works_on.project.controls.
departement.manages.employee[e]{e.ssn = #pssn};
```

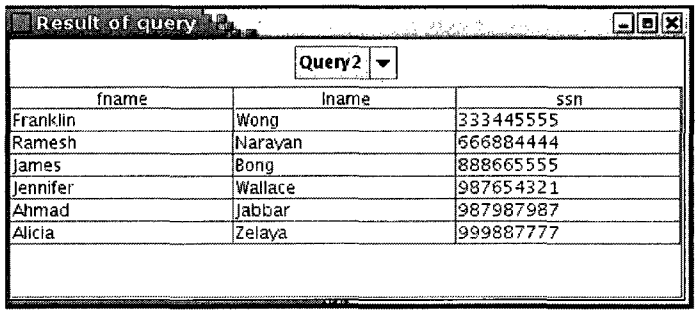
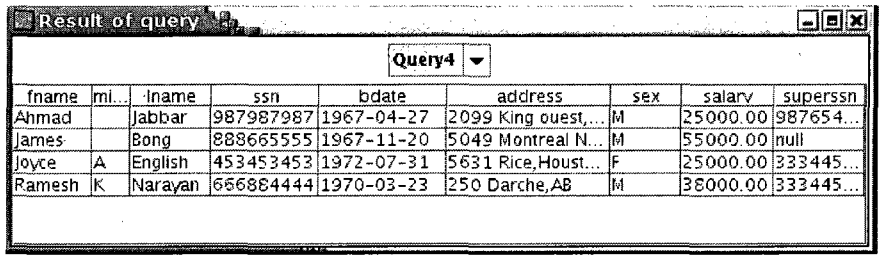


Figure 4.4 – Résultat de la requête Query3

- La quatrième requête (`Query4`) illustre l'utilisation des opérations ensemblistes, particulièrement l'opération d'intersection. Cette requête permet de retracer les em-

ployés qui travaillent dans les deux projets « Alpha » et « Beta ». Le résultat de cette requête est illustré dans la Figure 4.5.

```
Query4() ::=
(e.*
WHERE
project[p]{p.pname = "Alpha"}.works_on.employee[e])
intersection
(e.*
WHERE
project[p]{p.pname = "Beta"}.works_on.employee[e]);
```



fname	mi...	lname	ssn	bdate	address	sex	salary	superssn
Ahmad		Jabbar	987987987	1967-04-27	2099 King ouest,...	M	25000.00	987654...
James		Bong	888665555	1967-11-20	5049 Montreal N...	M	55000.00	null
Joyce	A	English	453453453	1972-07-31	5631 Rice,Houst...	F	25000.00	333445...
Ramesh	K	Narayan	666884444	1970-03-23	250 Darche,AB	M	38000.00	333445...

Figure 4.5 – Résultat de la requête Query4

Suite à l'application des schémas de traduction, nous obtenons une implémentation en Java des requêtes illustrée à la Figure 4.6.

```
 package eb3io;

import java.sql.*;
import org.apache.log4j.Logger;
/**
*
* @author imade
*/
public class Queries {

static Logger logger = Logger.getLogger(Queries.class);
private Connexion cx;

private ResultSetMetaData rsmd;

public void connect(String sgbd, String serverAddress,
               String dbName, String user, String pass) {
  try
  {
    cx = new Connexion(sgbd, serverAddress,
                                  dbName, user, pass);
  }
  catch (SQLException e)
  {
    logger.error(e.getMessage());
  }
}
public PreparedStatement temp0;
public PreparedStatement temp1;
public PreparedStatement temp2;
public PreparedStatement temp3;
```

Figure 4.6 – Fichier de sortie de l'étude de cas

```
1  public Queries(String sgbd, String serverAddress,
2                 String dbName,String user, String pass) {
3  super();
4  connect(sgbd,serverAddress,dbName,user,pass);
5  try
6  {
7  temp0 = cx.getConnection().prepareStatement(
8    "SELECT e.*"+
9        "FROM departement AS d,works_for,employee AS e"+
10       "WHERE d.dnumber=works_for.dnumber"+
11       "AND works_for.ssn=e.ssn"+
12       "AND (d.dname = 'Research' OR d.dname = 'Administration')"+
13       "AND e.salary > 20000"+
14       ""
15 );
16 temp1 = cx.getConnection().prepareStatement(
17   "SELECT employee.fname,employee.lname,employee.ssn"+
18       "FROM employee,works_on,project,controls,departement,
19        manages,employee AS e "+
20       "WHERE employee.ssn=works_on.ssn"+
21       "AND works_on.pnumber=project.pnumber"+
22       "AND project.pnumber=controls.pnumber"+
23       "AND controls.dnumber=departement.dnumber"+
24       "AND departement.dnumber=manages.dnumber"+
25       "AND manages.ssn=e.ssn"+
26       "AND e.ssn = ?"+
27       ""
28 );
29 temp2 = cx.getConnection().prepareStatement(
30   "SELECT p.* "+
31       "FROM project AS p,works_on AS w,employee AS e"+
32       "WHERE p.pnumber=w.pnumber"+
33       "AND w.ssn=e.ssn"+
34       "AND w.hours > 40"+
35       "AND ((e.salary <= 20000 OR e.salary >= 45000)
36        AND e.sex = 'M')"+
37       ""
38 );
```

Figure 4.6 – Fichier de sortie de l'étude de cas (suite)

```
 temp3 = cx.getConnection().prepareStatement(
  " SELECT e.* "+
       " FROM project AS p,works_on,employee AS e "+
       " WHERE p.pnumber=works_on.pnumber"+
       " AND works_on.ssn=e.ssn"+
       " AND p.pname = 'Alpha'"+
       "" +
 " INTERSECT "+
 " SELECT e.* "+
       " FROM project AS p,works_on,employee AS e "+
       " WHERE p.pnumber=works_on.pnumber"+
       " AND works_on.ssn=e.ssn"+
       " AND p.pname = 'Beta'"+
       "" );
}catch(SQLException e)
{
 logger.error(e.getMessage()) ;
}
}

 public ResultSet Query1( )
     throws SQLException {

ResultSet result = null;
  try
  {
result = temp0.executeQuery();
  }
  catch (SQLException e)
  {
    logger.error(e.getMessage());
  }
return result ;
}
```

Figure 4.6 – Fichier de sortie de l'étude de cas (suite)

```
1  public ResultSet Query2( double ppssn)
2       throws SQLException {
3  ResultSet result = null;
4    try {
5  double var0= ppssn;
6  Object var1= var0;
7  templ.setObject(1,var1);
8  result = templ.executeQuery();
9    }
10   catch (SQLException e){
11     logger.error(e.getMessage());
12   }
13 return result ;
14 }
15 public ResultSet Query3( )
16      throws SQLException {
17 ResultSet result = null;
18   try {
19 result = temp2.executeQuery();
20   }
21   catch (SQLException e) {
22     logger.error(e.getMessage());
23   }
24 return result ;
25 }
26 public ResultSet Query4( )
27      throws SQLException {
28 ResultSet result = null;
29   try {
30 result = temp3.executeQuery();
31   }
32   catch (SQLException e) {
33     logger.error(e.getMessage());
34   }
35 return result ;
36  }
37 }
```

Figure 4.6 – Fichier de sortie de l'étude de cas (suite)

Conclusion

Nous avons défini un langage formel de requêtes basé sur le modèle entité-association qui permet d'écrire les requêtes de sortie associées aux règles d'ES définies dans une spécification EB^3. Nous avons procédé aussi à la définition des schémas de traduction des expressions de chemins vers des requêtes SQL. L'outil EB^{3}QG qui automatise l'implémentation des requêtes à partir de ces schémas de traduction a été développé et intégré dans la plateforme APIS.

Bien que notre langage de requêtes fournit un bon équilibre entre simplicité et expressivité, il représente un sous-ensemble du langage SQL puisque qu'il n'intègre pas toutes les fonctions que propose SQL. Ainsi, il ne permet pas d'exprimer toutes les requêtes imaginables du langage SQL. Une autre limitation de notre langage est qu'il est basé seulement sur les concepts définis dans le modèle EA et ne prend pas en considération les nouveaux ajouts du modèle entité-association étendu (EAE).

Parmi les travaux futurs, une première étape serait d'incorporer dans notre langage de requêtes toutes les fonctions et les opérations supportées par le langage SQL. Une autre application d'un tel langage est de s'en servir comme un langage de requête pour le modèle de données EAE. Son extension à un langage de mise à jour et son inclusion dans un langage d'application hôte est un autre volet de recherches futures.

Conclusion

Annexe A

Grammaire hors contexte du langage de requêtes

```
grammar query;

options {
  output = AST; // uses CommonAST by default
  k=1;
}
tokens {
  UNION =  'union';
  INTER = 'intersection';
  PLUS  = '+';
  MOINS = '-';
  ETOILE = '*';
  ET = 'AND';
  OU = 'OR';
  NOT = 'NOT';
  SEQUENCE =  '.';
  DIVISION =   '/';
  COMMA = ',';
  COLON = ':';
```

```
  DEFINITION = '::=';
  SEMI = ';';
  LPAREN = '(';
  RPAREN = ')';
  LACCO = '{';
  RACCO = '}';
  LCRO ='[';
  LANG ='<';
  RCRO =']';
  RANG ='>';
  INFEGAL = '<=';
  SUPEGAL =  '>=';
  EGAL = '=';
  NOTEGAL = '!=';
  MINUS = 'minus' ;
  CARTESIEN = 'cartsien';
}

/* ------------Begin Grammar --------- */
/*--------------------------------------
* PARSER RULES
*--------------------------------------*/
spec
  : 'Queries'^ (listeRequetes)+ EOF!
  ;
listeRequetes
  : ID  listeParametresFormels DEFINITION definition (s=SEMI->s)
  -> ^(s  ^(ID   listeParametresFormels definition) )
  ;
definition
  : (a=requete->a)                  // set result
  (
    b=operationEnsembliste
    c=requete -> ^(b definition c) // use previous rule result
```

```
  )*
  ;
listeParametresFormels
  : LPAREN^ (parametreFormel (COMMA! parametreFormel)*
  )? RPAREN!
  ;
parametreFormel
  : ID COLON^ ID
  ;
requete
  : resultat 'WHERE'^ condition
  | LPAREN! requete RPAREN!
  ;
resultat
  : (attribut (COMMA^ attribut)*)?
  ;
attribut
  : ID^ SEQUENCE ID
  ;
condition
  : chemin  (operationLogique^ chemin)*
  ;
chemin
  : cheminBase (ETOILE^ | PLUS^)?
  ;
cheminBase
  : relation (SEQUENCE^  relation)*
  | LPAREN! chemin  RPAREN!
  ;
relation
  : ID^ alias? predicat?
  ;
alias
  : LCRO^ variable RCRO
```

```
  ;
variable
  : ID
  ;
operationLogique
  : ET^ | OU^ | NOT^
  ;
operationEnsembliste
  : UNION^ | INTER^ | MINUS^
  ;
predicat
  : LACCO^ predicatOU RACCO!
  ;
predicatOU
  : predicatET  (OU^ predicatET)*
  ;
predicatET
  : predicatNot (ET^ predicatNot)*
  ;
predicatNot
  : NOT^ predicatAtomique
  | predicatAtomique
  ;
predicatAtomique
  : terme (( LANG^ | INFEGAL^ | RANG^ | SUPEGAL^
  | EGAL^|NOTEGAL^) terme)*
  ;
terme
  : termeSommeDifference
  ;
termeSommeDifference
  : termeProduitDivision (( PLUS^ | MOINS^ ) termeProduitDivision)*
  ;
termeProduitDivision
```

```
  : termeAtomique (( ETOILE^ | DIVISION^) termeAtomique)*
  ;
termeAtomique
  : attribut
  | litteral
  | vrai
  | faux
  | LPAREN! predicatOU  RPAREN!
  ;
litteral
  : STRING
  | NOMBRE
  | DATE
  ;
vrai
  : 'true'
  ;
faux
  : 'false'
  ;
/*--------------------------------
* LEXER RULES
* --------------------------------*/
DATE
  : DATENUM '/' DATENUM '/' YEARNUM
  ;
fragment
DATENUM
  : ('0'..'9') ('0'..'9')?
  ;
fragment
YEARNUM
  : ('19'|'20') ('0'..'9')('0'..'9')
  ;
```

```
WS
  : (' '|'\r'|'\t'|'\u000C'|'\n') {channel=HIDDEN;}
  ;
DIGIT
  :   '0'..'9'
  ;
NOMBRE
  :  (DIGIT)+ ('.' (DIGIT)+ ('E' ('-')? (DIGIT)+)?)?
  ;
ID
  :    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
  ;
STRING
  :   '"' ID  '"'
  ;
SL_COMMENT
  :   '/*' .* '*/' {channel=HIDDEN;}
  ;
// multiple-line comments
ML_COMMENT
 : '///' ~('\n'|'\r')* '\r'? '\n' {channel=HIDDEN;}
 ;

/*-------------- END GRAMMAR ---------------*/
```

Annexe B

Diagramme de classes de la représentation objet

Cette annexe présente le diagramme de classes de la représentation objet des requêtes. Ces classes sont utilisées dans les phases d'analyse lexicale, d'analyse syntaxique et de traduction des requêtes.

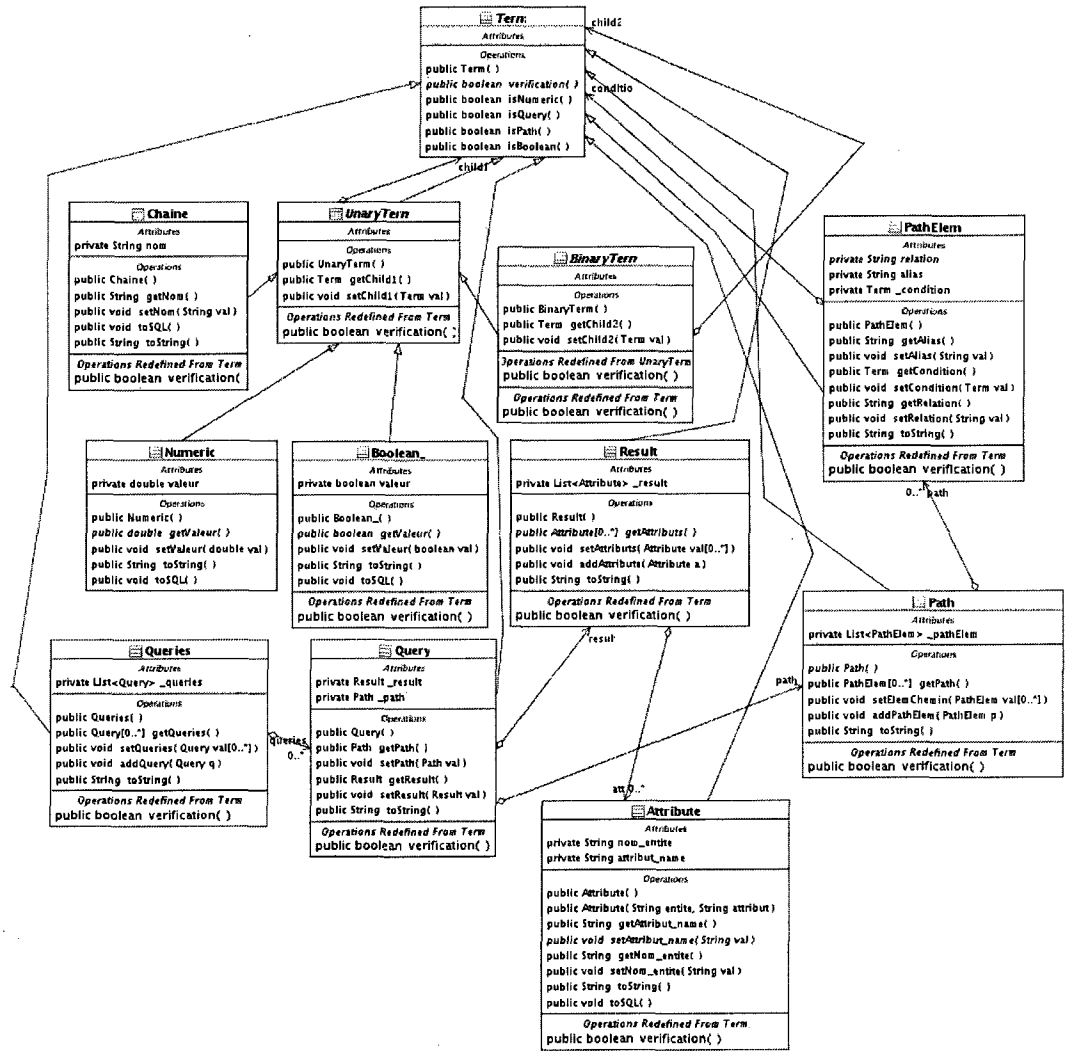


Figure B.1 – Diagramme de classes de la représentation objet

Bibliographie

[1] Panawé BATANADO. « Synthèse de transactions de base de données relationnelle à partir de définitions d'attributs EB^3 ». Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, Québec, Canada, juin 2005.

[2] Elisa BERTINO, Mauro NEGRI, Giuseppe PELAGATTI et Licia SBATTELLA. « Object-Oriented Query Languages : The Notion and the Issues ». IEEE Trans. Knowl. Data Eng., 4(3) :223–237, 1992.

[3] Peter P. CHEN. « The Entity-Relationship Model - Toward a Unified View of Data ». ACM Trans. Database Syst., 1(1) :9–36, 1976.

[4] Ramez ELMASRI et Shamkant B. NAVATHE. Fundamentals of Database Systems, 5th Edition. Benjamin/Cummings, 2007.

[5] Benoît FRAIKIN. « Interprétation efficace d'expression de processus EB^3 ». Thèse de doctorat, Université de Sherbrooke, Sherbrooke, Québec, Canada, avril 2006.

[6] Benoît FRAIKIN, Frédéric GERVAIS, Marc FRAPPIER, Régine LALEAU et Mario RICHARD. « Synthesizing Information Systems : the APIS Project ». Dans Colette ROLLAND, Oscar PASTOR et Jean-Louis CAVARERO, éditeurs, First International Conference on Research Challenges in Information Science (RCIS), Ouarzazate, Morocco, avril 2007.

[7] Marc FRAPPIER et Richard ST-DENIS. « EB^3 : An Entity-Based Black-Box Specification Method for Information Systems ». Software and Systems Modeling, 2(2) :134–149, 2003.

[8] Martin GOGOLLA et Uwe HOHENSTEIN. « Towards a Semantic View of an Extended Entity-Relationship Model ». ACM Trans. Database Syst., 16(3) :369–416, 1991.

[9] Charles Antony Richard HOARE. CSP–Communicating Sequential Processes. Prentice Hall, 1985.

[10] Michael KIFER, Won KIM et Yehoshua SAGIV. « Querying Object-Oriented Databases ». Dans Michael STONEBRAKER, éditeur, SIGMOD Conference, pages 393–402. ACM Press, 1992.

[11] Pierre KONOPACKI. « Synthèse automatique de gardes EB^3 ». Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, Québec, Canada, février 2008.

[12] Michael LAWLEY et Rodney W. TOPOR. « A Query Language for EER Schemas ». Dans Australasian Database Conference, pages 292–304, 1994.

[13] Xavier LEROY et Pierre WEIS. Manuel de référence du langage Caml. InterEditions, 1993.

[14] R. Mark VOLKMANN. « ANTLR 3 ». Java News Brief, 2008.