

AN OBJECT-ORIENTED STRUCTURED QUERY LANGUAGE
AND ITS TRANSLATION TO A FORMAL ALGEBRA

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

Ali Gürhan Gür

September 1997

QA
769
.D3
G87
1997

AN OBJECT-ORIENTED STRUCTURED QUERY
LANGUAGE AND ITS TRANSLATION TO A
FORMAL ALGEBRA

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Ali Gürhan Gür

Ali Gürhan Gür

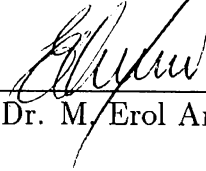
By

Ali Gürhan Gür

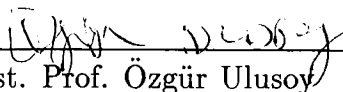
September, 1997

9A
76.3
107
687
107
8038466

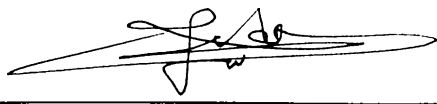
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Prof. Dr. M. Erol Arkun(Advisor)

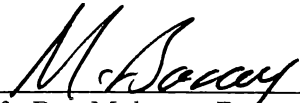
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Dr. Reda Alhajj

Approved for the Institute of Engineering and Science:


Prof. Dr. Mehmet Baray,
Director of Institute of Engineering and Science

Abstract

AN OBJECT-ORIENTED STRUCTURED QUERY LANGUAGE AND ITS TRANSLATION TO A FORMAL ALGEBRA

Ali Gürhan Gür

M.S. in Computer Engineering and Information Science

Supervisor: Prof. Dr. M. Erol Arkun

September, 1997

A declarative query capability has been accepted as a fundamental feature of any database management system. This thesis proposes an extension of the standard query language SQL, SQL/OO, designed for querying object-oriented databases. It has additional constructs to deal with the rich data model introduced by object-orientation. SQL/OO rests on a formal object-oriented query algebra that is highly expressive and open to optimization. Formal definitions of syntax and semantics are presented. The mapping of SQL/OO queries into object algebra is provided by a syntax-directed translation scheme. A prototype system that evaluates SQL/OO queries is designed. The system starts with a translator that translates an SQL/OO query into an equivalent object algebra expression. This algebra expression is parsed and an Object Algebra Tree (OAT) is generated which will be used as the internal representation. OAT Trees can be used as the input and output of a query optimizer module. The result of the query will be evaluated by traversing the tree and evaluating each node using proper functions that execute object algebra operations. A survey of existing object-oriented query languages in the literature is also provided. Their characteristics are identified, compared and contrasted, in order to present the necessary background.

Keywords: object-oriented database, query language, query algebra, SQL, translation.

Özet

NESNESEL YAPILI BİR SORGULAMA DİLİ VE BİÇİMSEL BİR CEBİRE ÇEVİRİSİ

Ali Gürhan Gür

Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Prof. Dr. M. Erol Arkun

Eylül, 1997

İfadesel bir sorgulama yeteneği, herhangi bir veri tabanı sisteminin temel bir özelliği olarak kabul edilmiştir. Bu tezde, standart sorgulama dili SQL'in bir uzantısı, SQL/OO, nesnel veri tabanlarını sorgulamak amacıyla önerilmiştir. Bu dil nesnel yaklaşımın getirdiği zengin veri modeliyle ilgilenmek için ek yapılara sahiptir. SQL/OO, ifade gücü yüksek, optimizasyona açık, nesnel bir sorgulama cebirine dayanır. Dilin sözdizimsel ve anlamsal tanımları sunulmuştur. SQL/OO sorgularının nesnel cebire eşlemesi sözdizimine dayalı bir çeviri düzeninde verilmiştir. SQL/OO sorgularını değerlendiren bir prototip sistem tasarlanmıştır. Sistem bir SQL/OO sorgusunu karşılık gelen cebirsel ifadeye çeviren bir çevirmenle başlar. Bu cebirsel ifade çözümlenerek içsel temsilci olarak kullanılacak olan nesnel cebir ağacı (OAT) oluşturulur. OAT ağaçları sorgu iyileştiren bir bölüm için girdi ve çıktı olarak kullanılabilir. Ağaç üzerinde dolaşarak her düğüm için uygun olan cebirsel işlemi gerçekleştirecek fonksiyon çalıştırılmak suretiyle sorgu sonucu hesaplanır. Ayrıca literatürde bulunan nesnel sorgu dillerinin genel bir özeti verilmiştir. Gerekli zemini sunmak amacıyla, bu dillerin özellikleri belirlenmiş, kıyaslanmış ve karşılaştırılmıştır.

Anahtar kelimeler: nesnel veri tabanı, sorgulama dili, sorgulama cebiri, SQL, çeviri.

*To my family,
who make this,
and all things,
possible...*

Acknowledgements

It is a great pleasure to acknowledge my debt to the people involved, directly or indirectly, in my study which led to the production of this thesis.

I owe a great deal to the guidance and encouragement of my supervisor, Prof. Dr. M. Erol Arkun. It has been a privilege to work with him during my years in Bilkent University. Without his clear thinking, understanding, great patience and faith in me this thesis would never have happened.

Asst. Prof. Reda Alhajj has always made himself available to answer my questions and to discuss my ideas, and his time and support is sincerely appreciated.

I would also like to thank the other member of my committee Asst. Prof. Özgür Ulusoy, who made useful comments about my work.

I owe special thanks to my colleague Çağlar Günyaktı and other friends Ümit V. Çatalyürek and Hakkı Tunç Bostancı for their endless intellectual and moral support during the development of this thesis.

To my family and friends, thank you for your help, for your support, for sharing my anxiety, for patiently urging me to finish, for sticking with me during some very difficult times and never having doubted me – even when I did myself. My parents' patience, understanding and infinite moral support have helped make the completion of this thesis a reality.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Related Work | 4 |
| 2.1 | Data Model Issues | 4 |
| 2.2 | Query Language Issues | 7 |
| 2.3 | A Survey of Existing Systems | 8 |
| 2.3.1 | Object SQL of IRIS | 8 |
| 2.3.2 | EXCESS of EXODUS | 9 |
| 2.3.3 | O ₂ SQL of O ₂ | 10 |
| 2.3.4 | ORIONQL | 11 |
| 2.3.5 | ONTOS SQL | 11 |
| 2.3.6 | CQL++ of ODE | 12 |
| 2.3.7 | OQL[C++] | 12 |
| 2.3.8 | Other Proposals | 12 |
| 2.4 | Comparison of Query Languages | 13 |
| 3 | Background | 16 |
| 3.1 | Data Model | 16 |
| 3.1.1 | Informal Description | 16 |
| 3.1.2 | Example Database | 17 |
| 3.1.3 | Basic Notations | 19 |
| 3.2 | Object Algebra | 20 |
| 3.2.1 | Informal Description | 20 |
| 3.2.2 | Object Algebra Operations | 22 |
| 4 | Query Language | 25 |
| 4.1 | Informal Description | 25 |

| | | |
|----------|---|-----------|
| 4.1.1 | Basic queries | 25 |
| 4.1.2 | Method calling | 26 |
| 4.1.3 | Complex objects | 26 |
| 4.1.4 | Object identity | 27 |
| 4.1.5 | Class hierarchy | 27 |
| 4.1.6 | Inheritance | 28 |
| 4.1.7 | Multiple domains | 28 |
| 4.1.8 | Subqueries in FROM-clauses | 29 |
| 4.1.9 | Subqueries in WHERE-clauses | 30 |
| 4.1.10 | Quantifiers | 30 |
| 4.1.11 | Aggregate functions | 30 |
| 4.1.12 | Set operations | 31 |
| 4.2 | Syntax and Semantics | 31 |
| 4.2.1 | Overview | 32 |
| 4.2.2 | FROM-clauses | 33 |
| 4.2.3 | WHERE-clauses | 34 |
| 4.2.4 | GROUP BY-clauses | 37 |
| 4.2.5 | HAVING-clauses | 38 |
| 4.2.6 | SELECT-clauses | 39 |
| 4.2.7 | Set Operations | 40 |
| 5 | Implementation | 41 |
| 5.1 | System Overview | 41 |
| 5.2 | Data Model Representation | 41 |
| 5.3 | SQL/OO to Algebra Translator | 45 |
| 5.4 | Algebra Parser and Tree Generator | 46 |
| 5.5 | Query Evaluator | 47 |
| 6 | Conclusions and Future Work | 49 |
| A | SQL/OO Syntax | 54 |
| B | SQL/OO to Algebra Translator | 57 |

List of Figures

| | | |
|-----|---|----|
| 3.1 | Example Database Schema | 18 |
| 5.1 | System Overview | 42 |
| 5.2 | Class definition of <code>AClass</code> | 43 |
| 5.3 | Class definition of <code>ACObject</code> | 44 |
| 5.4 | Class definition of <code>AMessage</code> | 45 |
| 5.5 | Class definition of <code>OATNode</code> | 47 |
| 5.6 | Class definition of <code>AOperand</code> | 48 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Data Model Features | 13 |
| 2.2 | Query Language Features | 14 |
| 2.3 | Data Modification Features | 14 |
| 2.4 | Query Domain, Range, Image Classes | 15 |

Chapter 1

Introduction

Object-Oriented Database Management Systems (OODBMSs) [6, 25, 15, 11, 21, 31] became popular in the mid-eighties, as a result of the sharply increased popularity of the object-oriented programming languages. Early efforts at OODBMSs, and query languages of them, centered around making particular object-oriented languages persistent [2]. These persistent languages provided support for user queries through the programming language itself, or simple preprocessor extensions of it. The queries written in these languages were therefore completely non-declarative. It was commonly believed that this was a sufficient querying capability for OODBMSs.

This belief no longer holds and declarative query capability, one of the most important reasons why Relational DBMS technology is so popular, has been accepted as a fundamental feature of OODBMSs. There are several proposals for declarative object-oriented query languages in the literature based on different object models and implementation schemes [10, 24, 4, 9, 18, 7]. In this thesis, a survey of existing query languages for object-oriented systems is provided. Common and different characteristics found in these systems are identified, compared and contrasted. A comparative analysis of the features, strength and weaknesses of existing languages are given in a tabular fashion.

An object-oriented query language needs to rest on an object-oriented algebra which provides an abstract execution engine with which queries can be expressed using algebraic operations. An algebraic background is important in expressive power of the query language, as well as in the optimization of queries. A formal object-oriented data model and a query algebra had been proposed in [3]. This algebra is very strong in terms of expressive power. However, for

real-world users of a database system, a formal algebra is too difficult to use. In this thesis, we present a high-level object-oriented query language, SQL/OO, which rests on this data model and query algebra. In defining SQL/OO, one of our goals was to stick as closely as possible to the standard query language SQL [19], both in syntax and semantics, and extend it naturally in order to incorporate object-oriented concepts. Hence the required effort for existing SQL users to learn SQL/OO will be minimized.

In designing our language, care was taken to obtain a language that is closed and adequate with respect to its underlying object-oriented data model. With closure we mean that the result of a query can be represented in the data model itself, i.e. as a derived class, properly placed in the class hierarchy. Closure is important in the context of query composition and view definition. With adequacy we mean that our language provides constructs to handle all features supported by the data model. In particular, we provide logical predicates and quantifiers adequate for handling complex structures, restructuring techniques and an inheritance mechanism.

The mapping of SQL/OO queries into object algebra is provided in terms of a translation scheme. The translation is syntax-directed, with translation rules associated with grammar productions. A translator which accepts SQL/OO query expressions in text form and translates them into equivalent object algebra expressions is implemented using YACC [20], a compiler compiler. The translator parses the input expression and generates the equivalent algebra expression using grammar actions that implement translation rules. In order to execute a query given in text form, it should be translated into an internal representation. With this intention a second parser is implemented. This parser accepts an object algebra query expression in text form and generates an object algebra tree (OAT), which serves as the internal representation. An OAT is an operator tree, with internal nodes representing algebra operations and leaf nodes representing input data operands. A prototype system that evaluates user queries is designed. Internal representations of the data model and the query model is defined using C++ [30] class structure. The modules which translate SQL/OO queries into internal representations are implemented.

The organization of this thesis is as follows. The related work in the literature is reviewed in Chapter 2. A survey of current user query languages for object-oriented database systems is presented, together with their characteristics and drawbacks. This survey does not include all object-oriented query languages, however the ones which mostly mentioned in the literature and well documented are selected. The reference data model and a formal object algebra which lies in the background of this thesis is presented in Chapter 3. The object-oriented user query language, which is the main contribution of this thesis is described in Chapter 4. The prototype implementation experience is discussed in Chapter 5. Finally the conclusions and future work is summarized the in Chapter 6.

Chapter 2

Related Work

In this chapter we present the necessary background knowledge about object-oriented database systems and their query languages. First we identify the common features supported by most object-oriented data models, and the variations. Then we discuss the characteristics of query languages and present a survey of various object-oriented query languages which are most referenced in the literature.

2.1 Data Model Issues

The proliferation of object-oriented systems is largely due to the data modeling power, behavior modeling power and the extensibility provided by these systems. A typical object-oriented system allows users to define new entity types, their attributes and behavior, that is the static and dynamic properties of entity types. It resembles abstract data types found in programming languages. The following features are found in many object-oriented data models and can be considered as the essential elements that should be found in every object-oriented data model [6].

Object and Object Identity Real world entities are represented by complex objects in object-oriented data models. Each object has a unique identifier called an object identifier. Object identifiers are vehicles for object sharing and recursive data structure.

Attributes and Methods Each object may have one or more attributes and one or more methods which operate on the values of the attributes. The value of an attribute may be an object or a collection of objects.

Encapsulation and Message Passing Attributes of an object can be accessed only via the defined methods. These methods form the public interface of the object and encapsulate the object attributes. A method is invoked by passing a message (i.e., the method name and the parameters if any) to the object.

Class A class is like an abstract data type in programming languages and every object is an instance of a class. Objects in the same class have the same attributes and methods.

Class Hierarchy and Inheritance A class can be defined as a subclass of another class. A subclass inherits all the attributes and methods from the superclass. Consequently, an object of a subclass can be used wherever an object of the superclass is expected. The inheritance relationships between classes form a graph called a class hierarchy.

Overloading, Overriding and Dynamic Binding Different methods can be associated with a single overloaded message name, and inherited methods can be overridden. System dynamically determines which method should be used according to the object class.

Despite the sharing of a set of common features, there are some differences that distinguish one object-oriented data model from another. The following is a list of the common differences found in existing models.

Objects and Values Some systems support values in addition to objects. Values do not have an explicit identifier and are identified by their contents. For example, in some systems *integer*, *real*, *string* and *boolean* are not considered as classes, but as base values. Some systems allow constructed values, such as tuples.

Direct Attribute Access and Message Passing Some systems do not allow object attributes to be accessed directly. Some systems allow a selected set of attributes of an object to be accessed directly. Others allow all attributes of an object to be accessed directly. In the latter case, an object is like a tuple with a set of associated methods.

Object Ownership and Sharing Support of complex objects arises the concept of object ownership. Some systems support additional constructs to identify whether an object reference in another object is owned or shared. Other systems treat complex objects as sharable, and values as non-sharable objects.

Class and Extent In some systems every class is associated with a user accessible collection called a class extent. An extent contains all instances of a class, and generally referred by the class name. Some systems do not support extents, however a user defined collection can be used as an extent.

Inclusive Extent and Exclusive Extent If the extent of a subclass is a subset of the extent of its superclass(es), it is called an inclusive extent. On the contrary, if the extent of a superclass does not include the extents of its subclasses, it is called as an exclusive extent.

Single Inheritance and Multiple Inheritance Some systems restrict all subclasses to have only one superclass. These models are said to support single inheritance. Others allow more than one superclass for a subclass. In other words, they support multiple inheritance.

Single Root and Multiple Roots For many systems, the class hierarchy is rooted at one class. There are systems that do not support a particular class of which all other classes are subclasses. Systems belonging to the first case have a single root. Those belonging to the second case have multiple roots.

Single Type or Multiple Type of Collection Class All object oriented data models support *set* as a collection class. For many systems *set* is the only collection class. However some systems have extra collection classes like *list* and *bag* (*multiset*).

2.2 Query Language Issues

This section discusses the approaches used by existing query languages in expressing complicated queries [10]. It explains the different constructs and points out the proper combinations of these constructs.

In relational context, queries are applied to relations and return relations again. In general, we can formulate a relation as a set of tuples. Hence relational query languages operate on sets and tuples. However object-orientation allow more complex structures and different collection types. Object-oriented query languages operates on collections of complex objects, but there are differences among object-oriented query languages about *range*, *image* and *domain* classes of queries. A *range* denotes the kind of the output collection of a query, such as a set, a bag, or a list. The element type of the output collection is referred as the *image*. A *domain* is a collection of objects on which a query will be evaluated. These specifications are closely related with the data models underlying the query languages. A *domain* can be a class extent, a "real" collection, or both depending on the data model. The *image* of a query can be an object or a value. While some languages can return new objects and values, others can return only existing ones.

Most of the query languages support multiple domains. One advantage of having multiple domains is to minimize complicated nesting of queries. In object-oriented models one domain can depend on another domain. This happens when one domain refers to a collection and another domain refers to a nested collection of each object in the first collection (e.g. the first domain could be a set of *Student* objects and the second domain referred to a set of *Course* objects belonging to the current *Student* object being processed). This dependence relationship is referred to as dependent domains. For an extent-based model, multiple domains can be useful even without the support of dependent domains, because dependent domains can be expressed using a membership test in the selection specification. For a non-extent-based model, multiple domains should come with dependent domain, otherwise nested collections cannot be handled easily.

In relational query languages (e.g. SQL and QUEL), nested subqueries are allowed only in the WHERE-clause and cannot appear in the SELECT or the FROM-clauses. This restriction does not create any serious problems.

because the relational model only supports atomic valued attributes which requires minimal structuring power. In object-oriented query languages, nesting subqueries in the selection specification of a query serves the same purpose as in the relational context. Nested queries can be correlated or independent. Many object-oriented query languages allow query composition in the image specification of a query so that results of a complex structure can be returned. Nesting of queries in the domain specification depends whether the result of a query can be used as a domain of another query.

Object-oriented data models subsume the relational model, so an object-oriented query language should similarly subsume relational completeness. Most object oriented languages support five basic relational algebra operation with additional semantics, therefore they can express all queries that can be expressed in the relational algebra. However *union* and *difference* operations are not supported by all languages.

Existential and universal quantifiers can simplify complex queries. Aggregate functions return a value from a collection and have been shown very useful in earlier data models.

While some languages are designed only for querying a database, some others can be used for data definition and data modification purposes as well. However data modification facilities are very limited. Dynamic schema evolution is supported by ORION only.

2.3 A Survey of Existing Systems

2.3.1 Object SQL of IRIS

The IRIS DBMS [22, 23] has been developed at Hewlett-Packard Laboratories. It is intended to meet the needs of office information and knowledge-based systems, engineering test and measurement, as well as hardware and software design. It supports persistence, concurrency, recovery, rules, inference, novel data types, clustering, long transactions and version control. The database can be accessed through two programming language interfaces, C-IRIS and Lisp-IRIS, and a query language Object SQL (OSQL) that can be used as a stand-alone interactive interface and a language extension.

The IRIS data model is a functional data model. Relationships between

classes can be modeled using independent functions not associated with any class. They are presented by values similar to tuples where retrieval is not done by field names but pattern matching. IRIS functions may be defined intensionally (i.e. using formula) or extensionally (i.e. without any formula but solely by explicit setting and updating of their values for particular arguments). The result of a function can be a base value or a set. Composition of functions is not limited to single-valued functions. Multi-valued functions can be composed together with the result being the union of the individual results. The data model also supports inverse attribute relationships.

Object SQL (OSQL) [9] is not a simple query language but a vehicle language for the database model. It is largely influenced by the relational paradigm and its standard SQL. As SQL, OSQL serves as a data description, data manipulation and query language. In the initial version of OSQL, some complex problems have to be solved by a call to external functions which have to be written in a foreign programming language. Thus, it has the traditional impedance mismatch problem. In later versions [5], OSQL has evolved to include general computational primitives and is now a computationally complete, extensible database language.

OSQL has an SQL-like syntax. An OSQL query is a sequence of function declarations followed by a select-from-where block. Alternative keywords are offered where the existing keyword would be misleading. As in SQL, a select-from-where block implicitly constructs and returns a set of tuples. However, the constructed set does not belong to the type hierarchy. Functions are the major modeling constructs of OSQL. User-defined functions can be invoked in select-from-where blocks.

Quantification and aggregate functions are not supported but can be augmented using functions. Structuring power is good except that it does not support new objects as the result of a query. Computational power is high. Usability can be improved if some unnecessary restrictions are removed.

2.3.2 EXCESS of EXODUS

EXCESS [14] is the query language of EXODUS database system. It is based on the EXTRA data model. EXTRA model supports not only objects but also values which are entities that do not respect the object encapsulation

and identity principles. Values are typed, objects belong to classes. EXTRA supports complex structures obtained by tuple, set and array constructors. The type system is based on a type lattice with multiple inheritance.

The EXCESS query language has a QUEL syntax. It allows modification of a database as well as querying. It also provides facilities to define new functions, these functions cannot be associated with classes but only to types. Both data and functions can be queried. Objects are queried through appropriate methods, values are queried according to their structure and appropriate functions. Hence encapsulation principle is not violated. EXCESS also supports aggregates and aggregate functions. The output of a query is a set of tuples.

The characteristics of EXCESS is very similar to OSQL since both use a relational query processor. Structuring power of EXCESS is good but new objects cannot be created as the result of a query. Computational power is high. Usability is acceptable.

2.3.3 O₂SQL of O₂

O₂ [8] is an object-oriented database system developed in Altair and has been later turned into a commercial product marketed by O₂ Technology. O₂ is designed to be a general purpose database system for various kinds of applications. It supports a database programming language called O₂C and a query language O₂SQL [7]. It supports persistence, concurrency, transactions, version control and distribution.

The O₂ data model features object identity, abstract typing, encapsulation, multiple inheritance and late binding. It supports all three kinds of collections and class extents as an option.

O₂SQL works in two modes: the interactive mode is for ad hoc queries, and the programming mode is for embedding into programming languages. Encapsulation can be violated in the interactive mode but not in the programming mode. The query is functional in nature. The syntax is SQL-like. It supports both multiple and dependent domains. A query returns a set of objects or values. Returned objects are that already exist in the database, while new values can be built.

O₂SQL is designed as a retrieval oriented language, thus its computational

power is poor. The query language is good in terms of functionality and usability.

2.3.4 ORIONQL

ORION [24] has been developed at MCC for CAD/CAM, artificial intelligence, multimedia and office information applications. ORION supports persistence, concurrency, transactions, recovery, composite objects, version control, dynamic schema evolution and multimedia data management.

The ORION Query Language (ORIONQL) proposal [25] has the basic SQL select-from-where structure. Unlike many systems, ORION data model supports exclusive extents. Hence querying a family of classes requires a class hierarchy operator ‘*’, which extends the evaluation to the extents of all subclasses.

Attributes can be directly accessed from the query language and hence encapsulation is not supported. Multiple and dependent domains are supported in the FROM-clause using ‘is-in’ operator, which can also be used in the WHERE-clause to specify a membership relationship.

Structuring power of the language is not good. Computational power is unsatisfactory except the schema evolution feature. Usability is the strong point of the language.

2.3.5 ONTOS SQL

ONTOS [4] is a commercial product from Ontologic. It aims to provide storage and retrieval mechanism for advanced applications. ONTOS operates in a multi-user, distributed homogeneous environment where transaction management and concurrency control are supported.

ONTOS SQL is an embedded query language for C++. ONTOS SQL is designed as a simple retrieval-based query language. The result of queries are limited to two forms: a list of strings, a list of lists of objects. It supports multiple domains but not dependent domains. Since class extents are not supported by the data model, the lack of dependent domains decreases the expressive power.

Both structuring and computational powers of the language are insufficient. Usability is acceptable in a limited scope.

2.3.6 CQL++ of ODE

ODE [2] is an object-oriented database system developed in AT&T Bell Labs. The object model of ODE is based on C++ [30]. The ODE Database is defined, queried and manipulated in the database language O++ [1] which is an extension of C++.

CQL++ [18] is proposed as a declarative front end to the ODE system. It uses an SQL-like syntax for defining classes, querying, displaying, and updating objects. Only tuples can be returned as the result of a query. Thus structuring power is poor. Computational power is good, however usability can be improved.

2.3.7 OQL[C++]

OQL[C++] [12] is an effort to extend C++ with an object query capability. It uses the C++ [30] type system as an object data model. The standard select-from-where structure of SQL is used for embedding queries in C++ programs. Certain C++ expressions can be used in the formulation of queries.

OQL[C++] supports the creation of new objects as the result of a query. New objects are created by calling a constructor in the select-clause, hence the structure of new objects should be predefined. Structuring and computational power are the strong points, however usability is poor.

2.3.8 Other Proposals

Bussche and Heuer [13] propose an extension of relational SQL to query object-oriented database systems. They present a mapping from an object-oriented data model to the nested relational data model, and define the semantics of their object-oriented extension of SQL by translating into the nested relational algebra. In this mapping of the object-oriented model to the nested relational model, object identifiers are stored in relations as an *identifying attribute*. Initially the uniqueness of object identifiers are assured, however restructuring of nested relations through *unnest* operation causes duplication of top level attributes including the *identifying attribute*.

The object-oriented data model and query language proposed by Sarkar and Reiss [29] is an elegant proposal. It is a rule based language translated into

an algebra that support operations on different collection types. This OQL avoids the dichotomy of values versus objects, is proved closed and complete, supports recursive queries, has support for complex ownership and sharing. However it is proposed as yet another query language with its own syntax completely different from any other known language.

Object comprehensions [17] is a new query notation which is developed from list comprehensions. Comprehensions are constructs based on the standard mathematical notation for sets, and widely used in functional programming languages. It is a clear and expressive query language.

2.4 Comparison of Query Languages

In previous sections, the features of various object-oriented query languages are presented. The purpose of this section is to summarize the study in a tabular fashion.

Table 2.1 gives a summary of the data model characteristics of the object-oriented query languages surveyed in this chapter. ‘I’ and ‘E’ in the ‘Class Extents’ row stand for inclusive extent and exclusive extent respectively, where ‘O’ means that class extents are supported as an option. ‘M’ and ‘S’ in the ‘Inheritance’ row stand for multiple inheritance and single inheritance. While ‘M’ and ‘S’ in the ‘Root Class’ stand for multiple roots and single root. ‘S’, ‘B’ and ‘L’ in the ‘Collection Classes’ row stand for *set*, *bag* and *list* respectively. A ‘+’ means that the data model supports other collection classes.

| | ORION | ONTOS | IRIS | O ₂ | EXTRA | ODE | Ref |
|-----------------|-------|-------|------|----------------|-------|-----|-----|
| Objects | Y | Y | Y | Y | Y | Y | Y |
| Base Values | Y | Y | Y | Y | Y | Y | Y |
| Tuples | N | N | Y | Y | Y | Y | N |
| Complex Objects | Y | Y | Y | Y | Y | Y | Y |
| Message Passing | Y | Y | Y | Y | Y | Y | Y |
| Encapsulation | N | Y | N | N | Y | Y | Y |
| Classes | Y | Y | Y | Y | Y | Y | Y |
| Class Extents | E | O I | I | O I | O I | I | I |
| Inheritance | M | S | M | M | M | M | M |
| Root Class | S | S | M | S | M | S | S |
| Collections | S | S L + | S B | S B L | S A | S | S |

Table 2.1: Data Model Features

Table 2.2 summarizes the features supported by the query languages. Multiple domains are supported by all languages. OSQL does not support dependent domains, since class extents are supported by its data model, this does not cause a problem. However the absence of dependent domains is a problem for ONTOS SQL where class extents are optional. None of the surveyed languages, except OQL[C++], can return new objects as the result of a query.

| | ORION | ONTOS SQL | OSQL | O ₂ SQL | EXCESS | OQL[C++] | CQL++ | SQL/OO |
|------------------------|-------|-----------|------|--------------------|--------|----------|-------|--------|
| Multiple Domains | Y | Y | Y | Y | Y | Y | Y | Y |
| Dependent Domains | Y | N | N | Y | Y | Y | Y | N |
| Returning New Objects | N | N | N | N | N | Y | N | Y |
| Nested Queries | N | N | Y | Y | Y | N | Y | Y |
| Existential Quantifier | Y | N | N | Y | N | Y | Y | Y |
| Universal Quantifier | Y | N | N | Y | Y | Y | N | Y |
| Aggregate Functions | Y | N | N | N | N | N | N | Y |
| Selection | Y | Y | Y | Y | Y | Y | Y | Y |
| Projection | Y | Y | Y | Y | Y | Y | Y | Y |
| Cartesian Product | Y | Y | Y | Y | Y | Y | Y | Y |
| Union | Y | N | N | Y | Y | N | Y | Y |
| Difference | Y | N | N | Y | Y | N | Y | Y |
| Recursion | Y | N | N | N | N | N | N | N |

Table 2.2: Query Language Features

Table 2.3 shows the modification features of the query languages. ONTOS SQL and O₂SQL are developed for retrieval purposes only and do not support any modification on the database. OSQL, EXCESS, OQL[C++] and CQL++ support features for insertion, deletion and update of objects. However only ORION allows schema evolution with existing data.

| | ORION | ONTOS SQL | OSQL | O ₂ SQL | EXCESS | OQL[C++] | CQL++ | SQL/OO |
|------------------|-------|-----------|------|--------------------|--------|----------|-------|--------|
| Insertion | N | N | Y | N | Y | Y | Y | N |
| Update | N | N | Y | N | Y | Y | Y | N |
| Deletion | N | N | Y | N | Y | Y | Y | N |
| Schema Evolution | Y | N | N | N | N | N | N | Y |

Table 2.3: Data Modification Features

Table 2.4 presents the supported collection classes for domain, range and image specifications of query. The collection class set is supported by all languages. List is supported by some languages but bag is generally not supported. When different collection classes are supported, they can be used in domain, range and image specification. ONTOS being an exception, restricts the image class to string and the range class to list. In CQL++ only constructed values out existing values can be returned. OQL[C++] is the only language that can return newly created objects.

| | | ORION | ONTOS SQL | OSQL | O ₂ SQL | EXCESS | OQL[C++] | CQL++ | SQL/OO |
|-----------------|-----------------|-------|--------------|------|--------------------|--------|----------|-------|--------|
| Domain Class | Set | Y | Y | Y | Y | Y | Y | Y | Y |
| | Bag | N | Y | N | Y | N | N | N | N |
| | List | N | Y | N | Y | N | N | N | N |
| Range Class | Set | Y | N | Y | Y | Y | Y | Y | Y |
| | Bag | N | N | N | Y | N | N | N | N |
| | List | N | Y | N | Y | N | N | N | N |
| Image Class | New Object | N | N | N | N | N | Y | N | Y |
| | Existing Object | Y | N | Y | Y | Y | Y | N | Y |
| | New Value | N | Y | Y | Y | Y | N | Y | N |
| | Existing Value | Y | N | Y | Y | Y | N | N | N |

Table 2.4: Query Domain, Range, Image Classes

Chapter 3

Background

In this chapter we present the underlying data model and object algebra [3] related to our query language proposal. In Section 3.1 we describe the reference object-oriented data model that we will use throughout the thesis. The reference data model presented here includes the significant features found in most object-oriented data models, some of which are summarized in Chapter 2. Next in this chapter, we give an informal description of the model in Section 3.1.1. An example database is defined in Section 3.1.2 that will be used to illustrate the introduced concepts. The basic notations about the data model is enumerated in Section 3.1.3.

The query model and object algebra is explained in Section 3.2. An informal description of the algebra operations given in Section 3.2.1 is followed by their formal definitions presented in Section 3.2.2.

3.1 Data Model

3.1.1 Informal Description

The reference data model supports objects, classes and methods. An object has an identity, a state and behavior. The identity of an object distinguishes it from other objects in the database and provides for object sharing. The state of an object is a set of values, one for each of the instance variables of its class. A value may be either a single value or a set of values drawn from a particular domain. A domain is either atomic, or non-atomic. An atomic domain may be one of the conventional domains, such as integers, strings, etc. On the other

hand, a non-atomic domain includes the set of objects of a class represented by their identities. The state of an object is reachable via the behavior, which is defined as methods applicable to the object.

Objects are collected into classes. A class has a set of instance variables, a set of superclasses, a set of methods and a set of objects as its instances. Instances of a class have the same state structure and behaviour definition. The state structure is defined by the set of instance variables, which reflects properties of objects in the class. The set of methods applicable to objects in the class denotes their behaviour. Instance variables can be accessed only via methods defined in the class, hence encapsulation and information hiding is supported. The set of superclasses provides reusability through inheritance. If a class appears in the superclasses set of another class, the latter class is called a subclass of the former one. A class can have more than one superclass and should inherit and support both state structure and behavior of its superclasses. As a consequence of inheritance, an object can be used wherever an object of its superclass is expected. There is a root class in the model which is a subclass of no class and a direct or indirect superclass of all other classes. The root class includes the definition common to all the classes found in the schema.

The main features supported by the reference data model can be summarized as follows:

- object identity
- class hierarchy
- multi-methods
- static type checking
- complex objects
- multiple inheritance
- method overloading
- dynamic binding
- classes
- encapsulation
- message passing
- class extents

3.1.2 Example Database

The example database is a simplified university administration system that records information about students and staff members of a university, its academic departments and courses. The relationships between classes defined in the schema are graphically shown in Figure 3.1.

The class *Person* has two subclasses: *Student* and *Staff*. *Assistant* inherits from both *Student* and *Staff* to represent students doing part-time teaching. Every person and academic department is given an address which is an object of class *Address*. Every staff member and student are associated to an academic

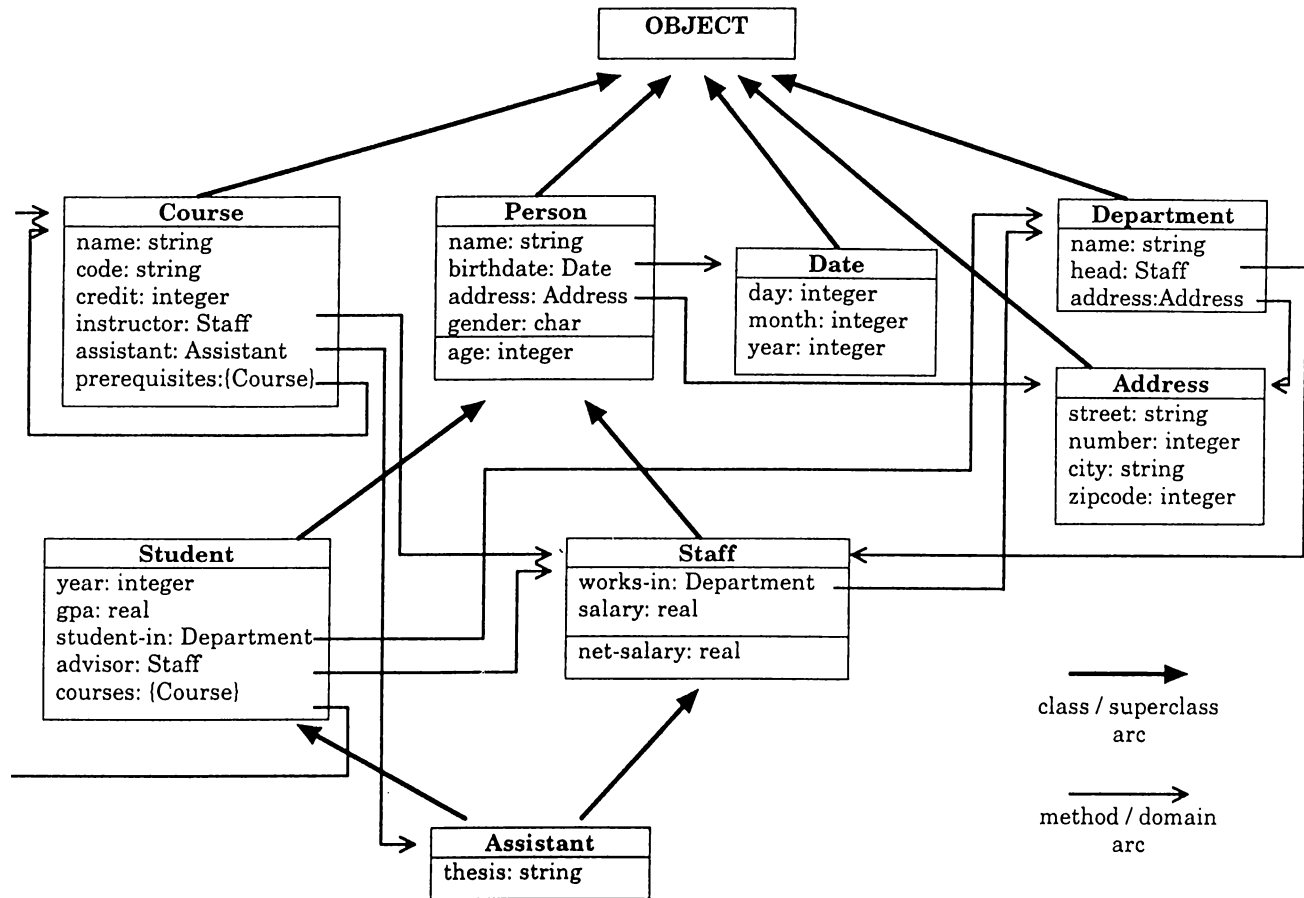


Figure 3.1: Example Database Schema

department of class *Department* via instance variables *works-in* and *student-in* respectively. A student has an *advisor* among staff members. Courses taken by each student is also recorded, by a set-valued instance variable *courses*. A course is an instance of the class *Course*. Each course is instructed by a staff member (*instructor*) and assisted by an assistant (*assistant*). A course may have a set of prerequisite courses (*prerequisites*). Each department has a head staff member associated by the instance variable *head*. The *birthdate* of each person is also stored as an object of class *Date*.

In the figure each node represents a class. *OBJECT* is the root class. A node is subdivided into three levels, the first of which contains the name of the class, the second the instance variables and the third the methods. Each instance variable and method is represented by a pair *name* : *domain* such that *name* is the instance variable or method name *domain* is the underlying domain. A domain specified between braces indicates that the corresponding

instance variable is set-valued. The nodes may be connected by two types of arc. The node which represents class C_1 may be connected to the node which represents class C_2 by means of:

- a thin arc, indicating that C_2 is the domain of an instance variable iv of C_1 , or that C_2 is the class of the result of a method m of C_1 .
- a bold arc, indicating that C_2 is a superclass of C_1 .

If the domain of an instance variable or a method is an atomic domain (for example integer, string etc.), only the name of the corresponding class is given. Atomic classes are not explicitly shown as nodes in the schema.

3.1.3 Basic Notations

The following notation adopted from [3] is used related to a class C :

- $messages(C)$ denotes the set of messages used to invoke any of the methods defined in or inherited by class C . A method is invoked via a corresponding message and retrieves either a stored value from an object, i.e. the value of an instance variable, or a derived value.
- $I_{variables}(C)$ denotes the set of all instance variables defined in or inherited by class C . For any instance variable iv , $domain(iv)$ and $value(iv)$ denote the domain and the value of instance variable iv .
- $instances(C)$ denotes the set of objects in class C but not in any of its subclasses. An object has an identity and a value. For any object o , $value(o)$ and $identity(o)$ are used to denote the value and the identity of object o , respectively.
- $T_{instances}(C)$ denotes the set of total instances of class C , which is defined to include instances of class C and all its direct and indirect subclasses. Formally speaking:

$$T_{instances}(C) = instances(C) \cup_{i=1}^{card(S)} T_{instances}(S_i)$$

where $S = \{S_1, S_2, \dots, S_{card(S)}\}$ is the set of direct subclasses of class C . i.e., $C \in supers(S_i)$.

- $supers(C)$ denotes the set of direct superclasses of class C .
- $M_e(C)$ denotes the set of message expressions of class C , which is defined as:
 - $messages(C) \subseteq M_e(C)$
 - if $x \in M_e(C)$ and x returns a value from $T_{instances}(C_1)$, then $(x \in messages(C_1)) \subseteq M_e(C)$.

The number of messages constituting a message expression x is denoted by $len(x)$. When received by an object, a message expression results in the execution of the methods underlying the constituting messages and in the same sequence. The underlying methods are executed as if they all form a single method invoked by the message expression. A message expression returns either a stored or a derived value.

3.2 Object Algebra

3.2.1 Informal Description

In the underlying object algebra, an operand consists of a pair of sets, a set of objects and a set of message expressions. Since a class has a defined set of objects and a derived set of message expressions, a class can be an operand. The result of an operation as well has a pair of sets derived in terms of the pair(s) of operand(s). Hence the result of any query operation can be placed as an operand of another operation, maintaining the closure property. The object algebra includes the five basic operators of the relational algebra in addition to nest, one-level project and aggregate function applications.

The *selection* operation has a single operand and produces an output consisting a pair, where the included objects are those satisfying a stated predicate expression. Although the set of objects in the output pair is restricted, however the message expressions of the output is the same as that of the operand.

A *predicate expression* is built using object variables, message expressions and constants, besides quantifiers may be used in a predicate. An object variable is bound by the set of objects of the operand. An object variable followed by a message expression returns either a stored or a derived value. A returned

value can be compared with another value or constant by using conventional comparison operators in addition to \subseteq , $\not\subseteq$, \in and \notin added to support set-based comparisons and \equiv , \doteq and \cong for *identical*, *shallow-equal* and *deep-equal* comparisons of objects, respectively. So predicates within an object-oriented context are more powerful than in the relational model where only atomic values are compared.

The *project* operation restricts the accessible part of each of the objects in the operand by eliminating some of the message expressions used in reaching the contents of an object. The same set of objects from the operand is maintained in the result, however only a stated set of message expressions can be used to deal with these objects. On the other hand, the *inverse project* operation extends the set of message expressions in the operand to include more message expressions applicable to objects of the operand, i.e. gives more facilities to the user.

The *one-level project* operation is used to bring values found at different levels of nesting within an object to the same level of nesting. Similarly to the *project* operation, a set of message expressions is stated with the operand. While the *project* operation does not evaluate any message expression, the *one-level project* operation evaluates the provided message expressions resulting in new objects, and a new set of message expressions is derived to be used dealing with those objects. Hence derived values are handled by this operation as well as the stored values.

Although many relationships between objects are represented within the objects themselves, an explicit operation is required to handle cases when a relationship is not present in the model. Both the *cross-product* and the *nest* operations are defined to introduce such relationships. While the *cross-product* operation is defined to be associative, the *nest* operation is not. However, the two operations are equivalent under certain conditions [3]. Associativity of the *cross-product* operation is achieved by defining the operation in four different forms depending on the domains of the instance variables of the operands.

The *cross-product* operation creates new objects, out of objects in the operands, and also a set of message expressions is derived to handle these objects. The *nest* operation also introduces missing relationships by extending the value of each object in the first operand to include a reference to object(s) in the second operand. It is a special case of *cross-product* operation, but the

difference is that here always the first operand is extended regardless of the underlying domains of the instance variables in the operands. On the contrary, the *unnest* operation is used to drop a present relationship. This operation is defined using the *project* operation.

Set operations *union*, *intersection* and *difference* are also supported. Their definitions are extended to handle both sets in an operand.

The *aggregate* operation allows to have the result of the application of an aggregate function used as an operand. A given aggregate function is evaluated on the result of a given message expression for the group of objects that return the same values for the elements of a given set of message expressions.

3.2.2 Object Algebra Operations

Let A and B be valid algebra operands, which are defined to have pairs of sets, $\langle T_{instances}(A), M_e(A) \rangle$ and $\langle T_{instances}(B), M_e(B) \rangle$, respectively. A and B may be classes or outputs from other queries. The algebra operations are defined as follows:

- *Selection:*

$$A[p] = \langle \{o \mid o \in T_{instances}(A) \wedge p(o)\}, M_e(A) \rangle \quad (1)$$

where p is a predicate expression.

- *Projection:*

$$A[M] = \langle T_{instances}(A), M \rangle \quad (2)$$

where $M \subseteq M_e(A)$.

- *One-level projection:*

$$\begin{aligned} A![M] = & \langle \{o \mid \exists o_1 \in T_{instances}(A) \wedge value(o) = (o_1 M)\}, \\ & \{x \mid \exists x_1 \in M \text{ with } x_1 \text{ returning a stored value, } x_1 = (x_2 m) \wedge \\ & len(x_1) = len(x_2) + 1 \wedge \exists x_3 \in M_e(A) \wedge x_3 = (x_2 x) \wedge x = (m x_4)\} \\ \cup & \{x \mid \exists x_1 \in M \text{ with } x_1 \text{ returning a derived value, } len(x) = 1 \wedge \\ & \forall o_1 \in T_{instances}(A) \exists o \in T_{instances}(A![M]) \text{ such that } o_1 x_1 = o x\} \rangle \quad (3) \end{aligned}$$

where $M \subseteq M_e(A)$.

- *Cross-product:*

Case 1: if $\exists x_i \in M_e(A), \text{len}(x_i) = 1 \wedge \exists x_j \in M_e(B), \text{len}(x_j) = 1$

$$\begin{aligned} A \times B &= \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge \\ &\quad \text{value}(o) = \text{identity}(o_1).\text{identity}(o_2)\}, \\ &\quad (m_1 M_e(A)) \cup (m_2 M_e(B)) \rangle \end{aligned} \quad (4)$$

Case 2: if $\forall x_i \in M_e(A), \text{len}(x_i) > 1 \wedge \exists x_j \in M_e(B), \text{len}(x_j) = 1$

$$\begin{aligned} A \times B &= \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge \\ &\quad \text{value}(o) = \text{value}(o_1).\text{identity}(o_2)\} \\ &\quad M_e(A) \cup (m_2 M_e(B)) \rangle \end{aligned} \quad (5)$$

Case 3: if $\exists x_i \in M_e(A), \text{len}(x_i) = 1 \wedge \forall x_j \in M_e(B), \text{len}(x_j) > 1$

$$\begin{aligned} A \times B &= \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge \\ &\quad \text{value}(o) = \text{identity}(o_1).\text{value}(o_2)\}, \\ &\quad (m_1 M_e(A)) \cup M_e(B) \rangle \end{aligned} \quad (6)$$

Case 4: if $\forall x_i \in M_e(A), \text{len}(x_i) > 1 \wedge \forall x_j \in M_e(B), \text{len}(x_j) > 1$

$$\begin{aligned} A \times B &= \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge \\ &\quad \text{value}(o) = \text{value}(o_1).\text{value}(o_2)\}, \\ &\quad M_e(A) \cup M_e(B) \rangle \end{aligned} \quad (7)$$

where m_1 and m_2 are messages with domains being $T_{instances}(A)$ and $T_{instances}(B)$, respectively.

- *Nest:*

$$\begin{aligned} A \gg B &= \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge \\ &\quad \text{value}(o) = \text{value}(o_1).\text{identity}(o_2)\}, \\ &\quad M_e(A) \cup (m M_e(B)) \rangle \end{aligned} \quad (8)$$

where m is a message with underlying domain $T_{instances}(B)$.

- *Unnest:*

$$A \ll B = A[M_e(A) - (m M_e(B))] \quad (9)$$

where $m \in M_e(A)$ is a message with underlying domain $T_{instances}(B)$.

- *Union:*

$$A \cup B = \langle T_{instances}(A) \cup T_{instances}(B), M_e(A) \cap M_e(B) \rangle \quad (10)$$

- *Difference:*

$$A - B = \langle \{o \mid o \in T_{instances}(A) \wedge o \notin T_{instances}(B)\}, M_e(A) - M_e(B) \rangle \quad (11)$$

- *Intersection:*

$$A \cap B = A - (A - B) \quad (12)$$

- *Aggregation:*

$$\begin{aligned} A(X, f, x_i) = & \langle \{o \mid (om_1) \subseteq T_{instances}(A) \wedge (om_3) = f(\{(o_1 x_i) \mid \\ & o_1 \in T_{instances}(A) \wedge \forall o_2 \in (om_1), (o_2 X) = (o_1 X)\})\}, \\ & (m_1 M_e(A)) \cup \{m_3\} \rangle \end{aligned} \quad (13)$$

where $X \subseteq M_e(A)$, $x_i \in M_e(A)$, f is an aggregate function and m_1 is a message with domain $T_{instances}(A)$. The operation is applied on A by evaluating the function f on the result of the message expression x_i for all objects that return the same values for elements of the set of message expressions X .

- *Inverse projection:*

$$A]M[= (A \gg B)![messages(A) \cup (m_2 messages(B))][M_e(A) \cup M] \quad (14)$$

where $M \subseteq M_e(B)$ is the set of message expressions to be added to $M_e(A)$, and m_2 is a message in the result of $A \gg B$ with domain $T_{instances}(B)$.

Chapter 4

Query Language

4.1 Informal Description

In this section we informally introduce our object-oriented database query language. The following subsections demonstrate our language using queries on the example database described in Section 3.1.2. The language's exact syntax and complete semantics in terms of its translation to the formal query algebra will be defined in Section 4.2.

4.1.1 Basic queries

Our language uses an SQL-like syntax, with extended constructs to deal with complex objects, method calls and object-orientation. Thus, typical select-project-join queries are expressed using standard SQL SELECT-FROM-WHERE clauses. The semantics of the SELECT-FROM-WHERE clauses are identical with their semantics in the relational context from user's point of view.

Query 1 *Return the names of all senior students.*

```
SELECT s.name
FROM   Student s
WHERE  s.year = 4
```

In this basic query, classes and methods are used analogous to the relations and attributes in relational SQL. The list of methods whose return values are to be output is specified in the SELECT-clause; the list of classes against whom

the query is formulated is given in the FROM-clause; and the WHERE-clause consists of a boolean combination of predicates, which specifies the selection criteria. The actual result of a query is a pair of sets as in the object algebra. However if it is a display query, a special method *display(d)* is applied on all objects in the first set in order to display the result. We call a query as a display query, if it is not a subquery nested in another query. Method *display(d)* is defined as a member of the root class *OBJECT* and inherited by all other classes. It takes an integer argument, *d*, which specifies the maximum depth of nesting to display. When it is applied to an object, it displays the values of the instance variables down to d^{th} level of nesting. By default, it is called with $d = 1$ that displays only the top-level instance variables.

4.1.2 Method calling

Encapsulation protects instance variables of an object from being accessed directly. Such an access must be made via a method. The previous example already showed the application of unary methods in queries, which returns the values of corresponding instance variables. Consider the more general method *net_salary(t)*, defined in the *Staff* class to return the net salary of a staff member after deducting taxes at the rate of *t*.

Query 2 *Return the names and net salaries of all staff members.*

```
SELECT s.name, s.net_salary(0.15)
FROM Staff s
```

Thus in our object-oriented SQL, methods returning derived values, as well as the ones returning stored values, can be used in both SELECT-clause and WHERE-clause.

4.1.3 Complex objects

Support of complex objects implies that a method call may return an object. The returned object can, in turn, receive another method call. This can go on for several method calls until, for instance an atomic value is returned. This sequential operation of methods is represented by a *path expression* that is a variable name followed by a sequence of zero or more method names separated by '.' operators.

Query 3 *Return the students who study in the department of Computer Science.*

```
SELECT s
FROM   Student s
WHERE  s.student_in.name = "Computer Science"
```

In this query, the path expression `s.student_in.name` represents the calling of method `name()` on the result returned by calling `student_in()` on a `Student` object `s`.

4.1.4 Object identity

In object-oriented data models, objects are represented by their identities which are essential for object sharing and representing cyclic relationships. Equality between objects becomes equality between object identifiers.

Query 4 *Return the assistants working and studying in the same department.*

```
SELECT s
FROM   Assistant s
WHERE  s.student_in = s.works_in
```

For example, in this query, to determine the assistants working and studying in the same department, two `Department` objects, returned by path expressions `s.student_in` and `s.works_in`, are compared using their object identifiers.

4.1.5 Class hierarchy

Recall that, our reference data model supports the class extends. The set $T_{instances}(C)$ of any class C includes instances of that class, and all instances of its direct or indirect subclasses. When we formulate a query on a class C , the query is executed on all elements of the set $T_{instances}(C)$.

Query 5 *Return all male persons in the database.*

```
SELECT p
FROM   Person p
WHERE  p.gender = "M"
```

This query returns objects, satisfying the selection criteria, from *Person*, *Student*, *Staff*, and *Assistant* classes.

4.1.6 Inheritance

The inheritance notion in object-oriented model implies that methods defined in a superclass are inherited by all its subclasses, hence can be applied to instances of these subclasses.

Query 6 *Return staff members younger than 40 years old.*

```
SELECT s
FROM   Staff s
WHERE  s.age < 40
```

Here the method *age()* defined in *Person* class is used on *Staff* objects, in order to return younger staff members.

4.1.7 Multiple domains

Although many relationships between objects are represented within the objects themselves, some queries may require relationships that are not represented in the modeling/modelling phase. Multiple domains in FROM-clause allow relationships that are not present in the database schema to be established.

Query 7 *Return the students studying in the same department as John Doe works in.*

```
SELECT x
FROM   Student x, Staff y
WHERE  y.name = "John Doe" AND x.student_in = y.works_in
```

In this example, the domain variable *x* iterates on objects in *Student* class and *y* iterates on objects in *Staff* class. The missing relationship is established applying *student_in()* and *works_in()* methods to objects *x* and *y*, respectively. Besides, multiple domains are essential to allow constructing new objects out of existing ones.

Query 8 *Return the couples living in the same address.*

```
SELECT x, y
FROM   Person x, Person y
WHERE  y.gender = "M" AND y.gender = "F" AND x.address = y.address
```

Here the domain variables x and y both iterate on the same domain, but independently from each other. One male and one female person, both living in the same address are considered as a desired couple. A new class of objects containing these couples are constructed and returned as the result.

4.1.8 Subqueries in FROM-clauses

Recall that the result of a query is a tuple of two sets: a set of objects, and a set of message expressions. As it is proved in [3], we can derive a class from any query result. Thus we can use SELECT-FROM-WHERE subqueries anywhere in query where we use a class name, especially in FROM-clause.

Query 9 *Return the high honor students who study in Management department.*

```
SELECT c
FROM   (SELECT s
        FROM   Student s
        WHERE  s.student_in.name = "Management") c
WHERE  c.gpa >= 3.5
```

In this example, range of variable c is limited by the inner subquery, which returns only the *Student* objects studying in Management department. Then high-honor students are selected from this intermediate result.

4.1.9 Subqueries in WHERE-clauses

Subqueries can be used in WHERE-clauses, where set values are expected as in the relational SQL.

Query 10 Return staff members who do not instruct any course.

```
SELECT s
FROM   Staff s
WHERE  s NOT IN (SELECT c.instructor
                FROM   Course c)
```

Here the required *Staff* objects are selected by testing their membership to the result of inner subquery, which returns the set of instructors.

4.1.10 Quantifiers

Universal and existential quantifiers can be used to answer queries involving knowing the number of elements in a collection that satisfy some particular conditions. A set-valued instance variable as well as a class name indicating a class extent can be used as the quantified collection.

Query 11 Return the courses which are not taken by any students.

```
SELECT c
FROM   Courses c
WHERE  FOR ALL s IN Student : c NOT IN s.courses
```

Query 12 Return the students who take a course instructed by his/her advisor.

```
SELECT s
FROM   Student s
WHERE  EXISTS c IN s.courses : c.instructor = s.advisor
```

4.1.11 Aggregate functions

Aggregate functions can be applied to all objects or groups of objects separated by GROUP BY-clause.

Query 13 *Return all departments together with the average salary of staff members working in that department.*

```
SELECT s.works_in, AVG(s.salary)
FROM   Staff s
GROUP BY s.works_in
```

In this example, staff members are grouped by their departments and average salary is calculated by applying AVG function to the result

4.1.12 Set operations

Binary set operations UNION, MINUS and INTERSECT can be applied on the results of queries.

Query 14 *Return all people who study or work in Computer Science department.*

```
SELECT s
FROM   Student s
WHERE  s.student_in.name = "Computer Science"
UNION
SELECT t
FROM   Staff t
WHERE  t.works_in.name = "Computer Science"
```

In this example, the *Student* objects returned by the first SELECT-FROM-WHERE subquery and the *Staff* objects returned by the second subquery are combined by the binary set operator UNION. The result of union operation involves objects of *Person* class that is the nearest common superclass of *Student* and *Staff* classes.

4.2 Syntax and Semantics

In this section we describe the syntax, and semantics in terms of translation to algebra. The syntax of the expressions are given using the BNF notation where nonterminals are written in italics, terminals are capitalized and enclosed in single quotations, optional parts are enclosed in square brackets. A complete grammar of the language is also presented as an appendix.

4.2.1 Overview

A typical query expression in our object-oriented SQL language has the general format:

$$\begin{aligned} \text{query-exp} ::= & \text{'SELECT' } \textit{projection-list} \\ & \text{'FROM' } \textit{domain-list} \\ & [\text{'WHERE' } \textit{predicate-exp}] \\ & [\text{'GROUP BY' } \textit{path-exp-list} \\ & [\text{'HAVING' } \textit{having-predicate}]] \\ & [\textit{set-operator query-exp}] \end{aligned}$$

The various clauses correspond to various steps in the query expressed by this expression, in the following order:

1. The FROM-clause expresses the *cross-product* of its arguments, which are either class names or subqueries;
2. The WHERE-clause expresses a restriction (*selection*) of the result of the preceding step, the condition of which may involve method calls and subqueries;
3. The GROUP BY-clause restructures (*aggregation*) the result of the preceding step;
4. The HAVING-clause expresses a further restriction (*selection*) on the the groups formed in the preceding step;
5. The SELECT-clause expresses the *one-level-project* of the result of the preceding step, as indicated by its arguments, which are message expressions.
6. Finally, *set-operators* (*union*, *intersection*, *difference*) are applied.

In the remainder of this section we will inductively specify all possible query expressions E , following the above order, and define their semantics by a query $Q(E)$ in the object algebra.

4.2.2 FROM-clauses

Syntax: Consider a query expression E of the form:

```
SELECT *
FROM domain-list
```

A *domain-list* is a comma list of *domain-items*. A *domain-item* is a *domain-spec* optionally followed by a domain variable. A *domain-spec* is either an identifier specifying a class name, or is a subquery that is a query expression enclosed in parenthesis.

$$\begin{aligned} \textit{domain-list} & ::= \textit{domain-item} \mid \textit{domain-list} \textit{' , ' domain-item} \\ \textit{domain-item} & ::= \textit{domain-spec} \mid \textit{domain-spec} \textit{ var-name} \\ \textit{domain-spec} & ::= \textit{class-name} \mid \textit{' (' query-exp ')} \end{aligned}$$

Semantics: For each *domain-spec*, $Q(\textit{domain-spec})$ is defined as follows:

- if *domain-spec* is an identifier, then $Q(\textit{domain-spec})$ is the class denoted by the identifier and translated as:

$$Q(\textit{domain-spec}) = \textit{class-name};$$

- if *domain-spec* is a subquery of the form (E') , then

$$Q(\textit{domain-spec}) = (Q(E')).$$

Now for each *domain-item*, $Q(\textit{domain-item})$ is defined as follows:

- if *domain-item* is a *domain-spec* followed by a range variable v , then

$$Q(\textit{domain-item}) = Q(\textit{domain-spec})\#v$$

where symbol $\#$ is used in the object algebra to bound variables to objects of an object algebra operand;

- otherwise, *domain-item* is just a *domain-spec* and

$$Q(\textit{domain-item}) = Q(\textit{domain-spec})$$

Finally $Q(E)$ is defined as follows:

- if *domain-list* consists of more than one *domain-items*, then

$$Q(E) = Q(\text{domain-item}_1) \times \cdots \times Q(\text{domain-item}_n)$$

for all *domain-items* in *domain-list*;

- if *domain-list* has only one *domain-item*, then

$$Q(E) = Q(\text{domain-item}).$$

4.2.3 WHERE-clauses

Simple Predicates

Syntax: Consider a query expression E of the form:

```
SELECT *
FROM domain-list
WHERE search-condition
```

A *search condition* In its simplest form, a *predicate expression* is a simple comparison of two *operands* using a binary *comparison operator*, which returns a boolean value. The syntax of a simple predicate expression is:

$$\text{predicate-exp} ::= \text{operand comp-operator operand}$$

An *operand* may be a *literal*, a *path expression* or a *domain-spec*.

$$\text{operand} ::= \text{literal} \mid \text{path-exp} \mid \text{domain-spec}$$

A constant *literal* is either a single literal value from any of the atomic domains, or a set literal that is a comma list of single literals enclosed in braces. The keyword 'NIL' is also a valid literal that represents the empty set.

$$\begin{aligned} \text{literal} & ::= \text{integer-literal} \mid \text{real-literal} \\ & \mid \text{string-literal} \mid \text{boolean-literal} \\ & \mid \text{set-literal} \mid \text{'NIL'} \\ \text{set-literal} & ::= \text{'\{ literal-list \}'} \\ \text{literal-list} & ::= \text{literal-list ' , ' literal} \end{aligned}$$

Finally boolean-literals and combination of predicate expressions using logical connectives AND, OR, NOT and paranthesis, are expected as valid predicate expressions.

$$\begin{aligned}
 \text{predicate-exp} & ::= \text{predicate-exp 'AND' predicate-exp} \\
 & | \text{predicate-exp 'OR' predicate-exp} \\
 & | \text{'NOT' '(' predicate-exp ')'} \\
 & | \text{'(' predicate-exp ')'} \\
 & | \text{boolean-literal}
 \end{aligned}$$

Semantics: By induction, we may assume that $Q(E^{-w})$ is already defined. In general, $Q(E)$ is defined using the *selection* operation as follows:

$$Q(E) = Q(E^{-w})[Q(\text{predicate-exp})].$$

For different forms of predicate expressions, $Q(\text{predicate-exp})$ is defined as follows:

- if *predicate-exp* is a boolean-literal:

$$\begin{aligned}
 Q(\text{TRUE}) & = T, \\
 Q(\text{FALSE}) & = F;
 \end{aligned}$$

- if *predicate-exp* is a simple comparison:

$$Q(\text{op}_1 \text{ cop } \text{op}_2) = Q(\text{op}_1) Q(\text{cop}) Q(\text{op}_2)$$

where op_1 and op_2 are operands, and *cop* is a comparison operator;

- if *predicate-exp* is a quantified predicate expression:

$$\begin{aligned}
 Q(\text{FORALL } v \text{ IN } \text{op} : p) & = \forall v \in Q(\text{op}) \wedge Q(p), \\
 Q(\text{EXISTS } v \text{ IN } \text{op} : p) & = \exists v \in Q(\text{op}) \wedge Q(p), \\
 Q(\text{op}_1 \text{ cop ALL } \text{op}_2) & = \forall x \in Q(\text{op}_2) \wedge Q(\text{op}_1) Q(\text{cop}) x, \\
 Q(\text{op}_1 \text{ cop ANY } \text{op}_2) & = \exists x \in Q(\text{op}_2) \wedge Q(\text{op}_1) Q(\text{cop}) x, \\
 Q(\text{op}_1 \text{ cop SOME } \text{op}_2) & = \exists x \in Q(\text{op}_2) \wedge Q(\text{op}_1) Q(\text{cop}) x, \\
 Q(\text{EXISTS}(\text{op})) & = \exists x \in Q(\text{op}) \wedge T
 \end{aligned}$$

where v is a variable name, op , op_1 , op_2 are operands, and *cop* is a comparison operator.

A *path expression* is a variable name optionally followed by zero or more method calls separated by dots. A method call is represented by the method name. The parameters of the method call, if any, are specified as a list enclosed in parenthesis following the message name. The syntax of a path expression is recursively defined as follows:

$$\begin{aligned} \textit{path-exp} & ::= \textit{var-name} \mid \textit{path-exp} \textit{'.'} \textit{method-call} \\ \textit{method-call} & ::= \textit{method-name} \mid \textit{method-name} \textit{'('} \textit{parameter-list} \textit{'}' \\ \textit{parameter-list} & ::= \textit{parameter} \mid \textit{parameter-list} \textit{','} \textit{parameter} \\ \textit{parameter} & ::= \textit{literal} \mid \textit{path-exp} \end{aligned}$$

A *comparison operator* is one of the usual binary relational operators, including the ones that compare set values. The negation of set comparators are also available by the use of an optional preceding keyword 'NOT'.

$$\begin{aligned} \textit{comp-operator} & ::= \textit{'='} \mid \textit{'<'} \mid \textit{'>'} \mid \textit{'<='} \mid \textit{'>='} \mid \textit{'<>'} \\ & \mid [\textit{'NOT'}] \textit{'IN'} \mid [\textit{'NOT'}] \textit{'CONTAINS'} \\ & \mid [\textit{'NOT'}] \textit{'IS SUBSET'} \mid [\textit{'NOT'}] \textit{'HAS SUBSET'} \end{aligned}$$

Now we will extend the definition of *predicate expression* to include quantified predicates. SQL/OO uses two different syntax to express quantified predicates. In the first form, an existentially or universally quantified variable that is an element of a set valued operand is declared and used in the predicate expression that follows the declaration. The keywords EXISTS and FOR ALL are used in this form of quantification.

$$\begin{aligned} \textit{predicate-exp} & ::= \textit{quantifier1} \textit{var-name} \textit{'IN'} \textit{operand} \textit{':'} \textit{predicate-exp} \\ \textit{quantifier1} & ::= \textit{'FOR ALL'} \mid \textit{'EXISTS'} \end{aligned}$$

In the second form, no additional variable is used, however the first operand is compared with elements of the second operand that has a set value.

$$\begin{aligned} \textit{predicate-exp} & ::= \textit{operand} \textit{comp-operator} \textit{quantifier2} \textit{operand} \\ \textit{quantifier2} & ::= \textit{'ALL'} \mid \textit{'SOME'} \mid \textit{'ANY'} \end{aligned}$$

The last type of predicate expressions is the traditional EXISTS predicate that tests whether the enclosed set-valued operand has non zero number of elements.

$$\textit{predicate-exp} ::= \textit{'EXISTS'} \textit{'('} \textit{operand} \textit{'}'$$

- if *predicate-exp* is a combination of several predicate expression:

$$Q(p_1 \text{ AND } p_2) = Q(p_1) \wedge Q(p_2),$$

$$Q(p_1 \text{ OR } p_2) = Q(p_1) \vee Q(p_2),$$

$$Q(\text{NOT}(p)) = \neg Q(p),$$

$$Q((p)) = (Q(p))$$

where p , p_1 and p_2 are predicate expressions.

$Q(\text{operand})$ is defined according to the following rules:

- if *operand* is a literal, then $Q(\text{operand})$ is defined as itself;
- if *operand* is a subquery of the form (E') , then

$$Q(\text{operand}) = (Q(E'));$$

- if *operand* is a *path expression*, then $Q(\text{operand})$ is the path expression itself.

$Q(\text{comp-operator})$ is defined as the corresponding operator in object algebra, may be with different syntax but the same semantics, for example:

$$Q(<=) = \leq$$

$$Q(\text{IN}) = \in$$

$$Q(\text{IS SUBSET}) = \subseteq$$

4.2.4 GROUP BY-clauses

We now turn to GROUP BY-clauses, and show how their semantics can be defined using the *aggregate* operation of the object algebra.

Syntax: Consider a query expression E of the form:

```
SELECT *
FROM domain-list
WHERE search-condition
GROUP BY grouping-list
```

A *grouping-list* is a comma list of message expressions.

Semantics: We will denote the part of E without GROUP BY-clause by E^{-g} . By induction, we may assume that $Q(E^{-g})$ is already defined. In general, a GROUP BY-clause is used together with an aggregate function call in E^{-g} . We define $Q(E)$ as the application of *aggregation* operation to $Q(E^{-g})$, such that:

- if there is a call to an aggregate function f with parameter m in SELECT-clause of E^{-g} , then

$$Q(E) = Q(E^{-g})\langle\{Q(\textit{grouping-list})\}, f, m()\rangle;$$

- if there is no aggregate function call, then

$$Q(E) = Q(E^{-g})\langle\{Q(\textit{grouping-list})\}, id, m()\rangle$$

where id is the identity function returning its argument as the result.

4.2.5 HAVING-clauses

After GROUP BY-clause there may be a HAVING-clause, which is used to denote a further restriction on the groups constructed by the GROUP BY-clause.

Syntax: Consider a query expression E of the form:

```
SELECT *
FROM domain-list
WHERE search-condition
GROUP BY grouping-list
HAVING having-condition
```

Having-condition is a predicate expression, similar to *search-condition*.

Semantics: Let the part of E without HAVING-clause be denoted by E^{-h} . By induction, we may assume that $Q(E^{-h})$ is already defined. $Q(E)$ is defined using the *selection* operation as:

$$Q(E) = Q(E^{-h})[Q(\textit{having-condition})]$$

4.2.6 SELECT-clauses

In standard SQL, the SELECT-clause essentially expresses a projection operation. However in object-oriented SQL, it will be better to use *one-level-project* operation, since it is more powerful than *project* operation in the sense that it handles both stored and derived values, and may construct new structures.

Syntax: Consider a query expression E of the form:

```
SELECT project-list
FROM domain-list
WHERE search-condition
GROUP BY grouping-list
HAVING having-condition
```

A *project-list* either the special is a comma list of *project-items*. A *project-item* is either a *path expression* or an aggregate function application. An aggregate function is specified by the function name (one of MAX, MIN, AVG, SUM,COUNT), followed by a path expression in paranthesis.

```
project-list ::= project-item | project-list , project-item
project-item ::= path-exp | function-name '(' path-exp ')'
function-name ::= 'MAX' | 'MIN' | 'AVG' | 'SUM' | 'COUNT'
```

Semantics: Let E^{-s} denote the expression obtained from E by replacing the *project-list* with '*'. By induction, $Q(E^{-s})$ is already defined in the previous sections. Here $Q(E)$ is defined using the *one-level-project* operation of the algebra as:

$$Q(E) = Q(E^{-s})! [Q(\textit{project-list})].$$

However the definition of $Q(\textit{project-list})$ depends on the number of domain items specified in the FROM-clause. Recall that the *one-level-project* operation requires a set of message expressions, and applies these message expressions to all instances of the object algebra operand it is applied. These message expressions are similar to path expressions, however they do not start with an object variable. In other words, a message expression is the concatenation of one or many message calls.

If there is only one domain specified in the FROM-clause, the domain variables are eliminated from the path expressions in the project-list, leaving only message expressions after translation.

If there is more than one domains specified in the FROM-clause, we know that these domains are joined using the *cross-product* operation, forming a complex structure with instance variables each from one domain. In other words, the level of nesting is increased and the domain variables refer to instance variables of the resulting complex structure. Since these instance variable are reachable via corresponding messages, it will be suitable to substitute proper messages instead of the domain variables in the projection-list.

If there is an aggregate function application in the SELECT-clause, it is also substituted by the message name that returns the result of the function, which is added by the *aggregate* operation.

4.2.7 Set Operations

We finally arrived at the last stage, set operations. The three basic set operations *union*, *difference* and *intersection* can be applied orthogonally to the query expressions.

Syntax: Consider a query expression E of the form:

$$\begin{array}{c} E_1 \\ \text{set-operator} \\ E_2 \end{array}$$

where E_1 and E_2 are general query expressions and *set-operator* is one of UNION, INTERSECT, MINUS.

Semantics: By induction, $Q(E_1)$ and $Q(E_2)$ are already known. $Q(E)$ is defined depending on the type of set-operator as:

$$\begin{aligned} Q(E_1 \text{ UNION } E_2) &= Q(E_1) \cup Q(E_2) \\ Q(E_1 \text{ INTERSECT } E_2) &= Q(E_1) \cap Q(E_2) \\ Q(E_1 \text{ MINUS } E_2) &= Q(E_1) - Q(E_2) \end{aligned}$$

Chapter 5

Implementation

5.1 System Overview

A prototype system that evaluates SQL/OO queries is designed. The system consists of four modules each of which operates in a sequential manner as shown in Figure 5.1. In this study, we implemented the modules contained in the dotted rectangle. A framework that can be used in the implementation of query optimizer and evaluator modules are also proposed, however their implementations are left as future work.

5.2 Data Model Representation

One of the important features of the underlying object algebra is that the algebra operations can create new classes and objects out of existing ones. Both the structure and the behavior of objects can be extended or restricted by these operations, allowing dynamic schema evolution. Objects can migrate from one class to another as a consequence of an algebraic operation, hence they are not strongly typed. Therefore our data model representation should handle these dynamic features of the algebra.

However data models of existing persistent systems do not provide us the flexibility required to implement the algebra operations. Thus we decided to define a data model representation that simulates the dynamic features of the algebra. The data model representation is designed using C++ classes. Instances of C++ classes which are defined as persistent, are assumed to be

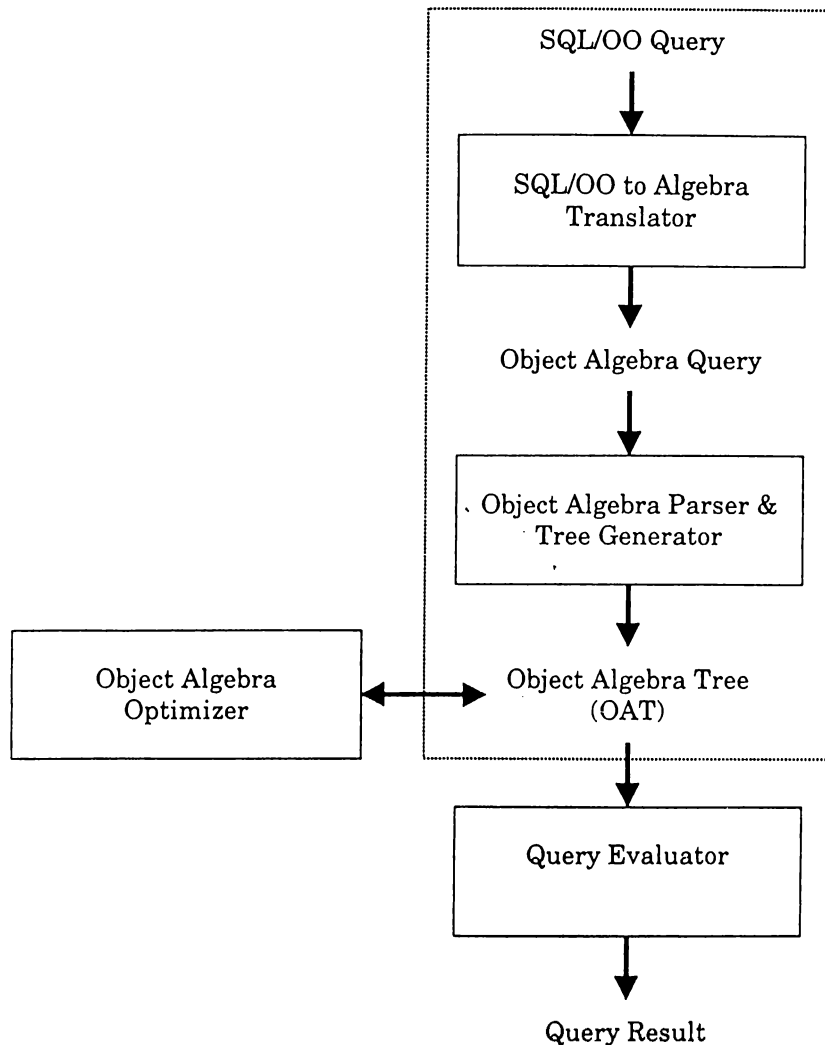


Figure 5.1: System Overview

persistent, meaning that they continue to exist after the program that created them has terminated. Existing persistent extensions to C++ have limitations about the structure of classes that are defined as persistent. They only allow fixed sized data members, which is clearly not suitable for our task at hand. One can propose to define an upper bound for variable sized data members, but in some cases sizes can inevitably grow beyond bounds.

Each persistent object is identified by a unique identifier, which is a pointer to a persistent object. Information about both the database classes and the database objects are stored as persistent C++ objects.

A persistent class, called `AClass` is defined in order to hold database classes. The class definition of `AClass` is given in Figure 5.2. `AClass` has members

that held necessary information related to a class. While the local properties are hold as data members, properties depending on inheritance and nesting hierarchies are returned by member functions that traverse these hierarchies and calculate the results. For example, local messages defined for a class is held in the data member `loc_msgs`, while all messages applicable to the instances of the class including the inherited ones can be retrieved by a call to the member function `messages()`. This member function traverses the inheritance hierarchy by following the links held in `supers` data member, and returns the union of messages held in each `loc_msgs`.

```
class AClass {
public:
    string      name;           // class name
    set<AClass*> supers;        // direct superclasses
    set<AClass*> subs;         // direct subclasses
    set<AMessage> loc_ivars;    // local instance variables
    set<AMessage> loc_msgs;    // local messages
    set<AObject*> instances;    // local instances
    set<AClass*> all_supers(); // all superclasses
    set<AClass*> all_subs();   // all subclasses
    set<AMessage> messages(); // all messages
    set<AMessage> Ivars();     // all instance variables
    set<AMessage> Tinsts();    // all instances
    AMsgExpSet  Mexps();      // all message expressions
    ...
};
```

Figure 5.2: Class definition of AClass

A database schema is stored in a dictionary that maps class names into corresponding AClass objects. Database objects are stored as instances of the AObject persistent class. This class is actually a virtual base class for AAObject and ACOBJECT classes, which represent objects from atomic and non-atomic domains respectively. Classes representing objects from atomic domains, i.e. AAStrObject, AAIntObject, AARealObject ... all derive from AAObject class. Pointers to AObject objects serves as object identifiers. An instance of AObject has two data members: domain holds a pointer to an AClass object representing the domain class of the object; value holds the value of the object. While value fields of AAObject objects are one of the types int, float, string and

boolean; value field of a `ACObject` is a vector of set of `AObject` pointers, where each element of the vector corresponds to an instance variable. The value of an instance variable is represented as a set of pointers in order to homogeneously handle both set valued and single valued instance variables. A single valued instance variable can be represented as a singleton set. The class definition of `ACObject` class is given in Figure 5.3.

```
class ACObject : AObject {
public:
    AClass          *domain; // domain class of object
    vector<set<AObject*>> value; // value of object
    virtual void    display(int d); // displays the object d levels
    ...
};
```

Figure 5.3: Class definition of `ACObject`

A message in algebra is represented as an `AMessage` object. The class definition of `AMessage` is given in Figure 5.4. The message name is hold in `name`, and the expected domain of the returned object is represented as a pointer to an `AClass` object. Expected argument types are stored as an array of `AClass` pointers in the declared order. If the message does not take any argument and returns the value of an instance variable, the index of the vector entry in the representation that corresponds to the instance variable is stored in `ivarIndx`. When a message returning the value of an instance variable is passed to an object, this index is fetched from the proper message element stored in the corresponding `AClass` object, and the value of the instance variable is accessed using this index.

A message expression in the algebra is defined as a sequence of messages. Possible cycles in the nesting hierarchy disallows the representation of message expressions simply in a list structure. A tree structure with backwards links is found to be a more proper representation for message expressions. Such a tree can be used to represent a set of message expressions as well as a single one. Thus we define another class `AMsgExpSet` that stores a set of message expressions in a tree structure. Usual set operations are also defined on `MsgExpSet` objects as member functions.


```

class AMessage {
public:
    string      name;          // message name
    AClass      *domain;      // expected return type
    int         numArgs;      // expected number of arguments
    vector<AClass*> argDomains; // expected argument types
    int         ivarIndx;     // if returns inst.var., its index
    ...
};

```

Figure 5.4: Class definition of AMessage

5.3 SQL/OO to Algebra Translator

The first module accepts the user query in the form of an SQL/OO expression and translates it to an equivalent query in the object algebra. The translation scheme is syntax-directed, with translation rules associated with grammar productions. The translator is implemented using YACC [20], a compiler compiler. A similar translation scheme had been defined in [16] from the relational SQL into the relational algebra. The source code of the translator is given in Appendix B. The translation rules are already defined in Section 4.2 while explaining the semantics of our language. Here we will present an example translation in order to demonstrate the features of our translator. Consider the following query:

Query 15 *Return the couples living in the same address.*

```

SELECT x, y
FROM   Person x, Person y
WHERE  y.gender = "M" AND y.gender = "F" AND x.address = y.address

```

The query is translated into the object algebra as follows:

```

Q(SELECT x, y
FROM Person x, Person y
WHERE y.gender = "M" AND y.gender = "F" AND x.address = y.address )

```

$$\begin{aligned}
&= Q(\text{Person } x, \text{Person } y) \\
&\quad [Q(y.\text{gender} = \text{"M"} \text{ AND } y.\text{gender} = \text{"F"} \text{ AND } x.\text{address} = y.\text{address})] \\
&\quad ![Q(x, y)] \\
\\
&= \text{Person}\#x \times \text{Person}\#y \\
&\quad [Q(y.\text{gender} = \text{"M"}) \wedge Q(y.\text{gender} = \text{"F"}) \wedge Q(x.\text{address} = y.\text{address})] \\
&\quad ![\{m_1(), m_2()\}] \\
\\
&= \text{Person}\#x \times \text{Person}\#y \\
&\quad [y.\text{gender}() = \text{"M"} \wedge y.\text{gender}() = \text{"F"} \wedge x.\text{address}() = \text{address}()] \\
&\quad ![\{m_1(), m_2()\}]
\end{aligned}$$

5.4 Algebra Parser and Tree Generator

This module accepts a query in the form of an object algebra expression as its input, parses the expression into an internal representation. The representation of a query by an operator tree is useful for manipulating the query expression since the representation is tied very closely to the algebraic query expression. Nodes in an operator tree represent query operators or access methods. Algebraic transformations can be applied on an operator tree in order to optimize the algebraic query expression. Also a query represented as an operator tree can easily be translated into an executable plan for query execution [28].

In our prototype system, a query is represented as an object algebra tree (OAT). An object algebra tree is composed of data nodes and operator nodes, connected by arcs. This representation generalizes algebraic operator tree representations by treating algebraic operators, logical operators, comparison operators and dot operator uniformly. Dot operator is used to apply message calls to objects.

Data nodes represent data that is manipulated as part of executing the query. A data node can represent a class of objects or a single object in the database. Data nodes can occur as leaf nodes in an OAT. *Operator nodes* represent actions that can be taken on data. An operation node will always have at least one child node, representing the input to the operation. If the

input child is a data node, then the class or object represented by that node serves as the input to the parent operation node. If the input child is another operation node, then the result of the child operation node serves as the input.

Nodes are labelled with syntactic query information. Operation nodes are labelled with operation names. Data nodes are labelled with a class name or a variable name for the data represented by the node. Nodes are connected by arcs representing relationships between data and operations in a query. In general, the root of an OAT is an operation node representing an algebraic operation.

OATNode representation class is defined for OAT nodes. The class definition is presented in Figure 5.5. The type of the node is held in type data member. The label field holds the appropriate label depending on the type. This information is used during query evaluation in order to identify what operations to perform with the children. Arcs are held as a list of pointers to OATNode objects representing the child nodes.

```
class OATNode {
public:
    string          type;          // node type
    string          label;        // node label
    list<OATNode*> children;      // links to children nodes
    ...
};
```

Figure 5.5: Class definition of OATNode

5.5 Query Evaluator

The query evaluator module, which is not actually implemented in the scope of thesis, is designed for executing a query passed in the form of an OAT. A direct execution plan can be generated from an OAT. The query execution represented by an OAT can be described as a top-down recursive execution. This results from the fact that evaluation of an operation node requires the data provided by the evaluation of its input child node(s).

All children of an operation node do not necessarily serve as input. Some children represent other information used in the application of the operation.

For example, *select* operation requires only one operand, however an OAT node representing a *select* operation have two children. The first child represents the actual input of the operation, while the second child represents the predicate expression used as the selection criteria. In query evaluation, input nodes are evaluated once, however the subtree representing the predicate expression is evaluated for each object in the input operand.

Evaluation of an operation node returns an object algebra operand, whose representation class definition is presented in Figure 5.6.

```
class AOperand {
public:
    set<AObject*>    Tinsts;        // set of objects
    AMsgExpSet      Mexps;        // set of message expressions
    ...
};
```

Figure 5.6: Class definition of AOperand

The result of an algebraic operation is calculated using the data members of the algebra operands returned after the evaluation of its input children. The execution of the query ends with the evaluation of the root operation node. The actual result of the whole query is returned in the *Tinsts* data member of the resultant operand object, which is a set of pointers to the representations of database objects.

The actual implementation of the query optimizer and the query evaluator is considered outside the scope of this work.

Chapter 6

Conclusions and Future Work

This thesis investigated the design and some aspects of the processing of query languages for object-oriented database systems.

A survey of existing high-level object-oriented query languages is presented. Their common and different characteristics are identified. This survey leads to the design of a high-level query language for object-oriented database systems. This query language, SQL/OO, is developed as a natural extension of the standard relational query language SQL. SQL/OO rests on a formal data model and query algebra previously proposed by Alhajj [3]. The language is mapped into the algebra using a syntax-directed translator.

Existing persistent systems were incapable of supporting all dynamic features of the query model. A prototype system that simulates algebra operations is designed using the C++ class structure. In this framework, the internal representations for data and query models are developed.

Also a prototype system that executes SQL/OO and object algebra queries is designed. Two parsers that translate user queries into internal representations are implemented.

As the future work and improvements, we can list the following:

- The data model representation can be extended in order to handle stored methods defined for database objects, as well as the instance variables.
- A query evaluator module can be implemented based on the proposed framework, using the defined data model and the query model representations. A direct execution plan can be generated given the OAT representation of the query.

- A query optimizer module, which inputs and outputs queries represented as OAT, can be implemented. Algebraic transformations can be performed on the input OAT in order to find a more efficient query, which seems to be a challenging problem itself.
- The translation scheme defined from SQL/OO to algebra can be extended to translate other query languages into the object algebra.

Bibliography

- [1] R. Agrawal, S. Dar, and N. Gehani. The O++ Database Programming Language. Technical Report 2, AT&T Bell Labs, Murray Hill, NJ, 1993.
- [2] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 25 – 40, Portland, OR, June 1989.
- [3] R. Alhajj. *A Query Model and an Object Algebra for Object-Oriented Databases*. PhD thesis, Bilkent University, Ankara, Turkey, February 1993.
- [4] T. Andrews. *The ONTOS Object Database*. Manuscript.
- [5] J. Annevelink, et.al. Object SQL – A Language for the Design and Implementation of Object databases. In [27].
- [6] M. Atkinson, et.al. The Object-Oriented Database System Manifesto. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.
- [7] F. Bancilhon, et.al. A Query Language for the O₂ Object-Oriented Database System. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, June 1989.
- [8] F. Bancilhon, C. Delobel, and P. Kanellakis, eds. *Building an Object-Oriented Database System – The Story of O₂*. Morgan Kaufmann, San Mateo, CA, 1992.

- [9] D. Beech. A Foundation for Evolution From Relational to Object Databases. In *Proceedings of the International Conference on Extending Database Technology*, Vol. 303 of *Lecture Notes in Computer Science*, pages 251–270, Springer-Verlag, New York, NY, March 1988.
- [10] E. Bertino, et.al. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, June 1992.
- [11] E. Bertino and L. Martino. *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley, Reading, MA, 1993.
- [12] J. A. Blakeley. OQL[C++]: Extending C++ with an Object Query Capability. In [26].
- [13] J. V. Bussche and A. Heuer. Using SQL with Object-Oriented Databases. *Information Systems*, 18(7):461–487, 1993.
- [14] M. J. Carey, et.al. A Data Model and Query Language for EXODUS. Technical Report CS-TR-734, University of Wisconsin, Madison, WI, 1987.
- [15] R. G. G. Catell. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, MA, 1991.
- [16] S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.
- [17] D. K. C. Chan. *Object-Oriented Query Language Design and Processing*. PhD thesis, University of Glasgow, Glasgow, U.K., September 1994.
- [18] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: A SQL for the ODE Object-Oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, Vol. 580 of *Lecture Notes in Computer Science*, pages 201–206, Springer-Verlag, New York, NY, 1992.
- [19] C. J. Date. *A Guide to the SQL Standard*. 2nd edition, Addison-Wesley, Reading, MA, 1987.
- [20] C. Donnelly and R. Stallman. *Bison - The YACC-Compatible Parser Generator version 1.20*. Free Software Foundation, December 1992.

- [21] A. Doğaç, M. T. Özsu, A. Biliris, and T Sellis, eds. *Advances in Object-Oriented Database Systems*, Vol. 130 of *NATO ASI Series F: Computer and Systems Sciences*. Springer-Verlag, New York, NY, 1994.
- [22] D. H. Fishman, et.al. IRIS: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.
- [23] D. H. Fishman. Overview of the IRIS DBMS. In [26].
- [24] W. Kim. Features of the ORION Object-Oriented Database System. In [26].
- [25] W. Kim. *Introduction to Object-Oriented Databases*. Computer Systems. The MIT Press, Cambridge, MA, 1990.
- [26] W. Kim and F. H. Lochovsky, eds. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1992.
- [27] W. Kim, ed. *Modern Database Systems: The Object Model, Interoperability and Beyond*. ACM Press, Addison-Wesley, Reading, MA, 1995.
- [28] G. A. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Brown University, Providence, RI, May 1993.
- [29] M. Sarkar and S. P. Reiss. A Data Model and a Query Language for Object-Oriented Databases. Technical Report CS-92-57, Brown University, Providence, RI, December 1992.
- [30] B. Stroustrup. *The C++ Programming Language*. 2nd edition, Addison-Wesley, 1991.
- [31] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice-Hall, New York, NY, 1997.

Appendix A

SQL/OO Syntax

query-exp ::= (*query-exp*)
| *query-exp* *set-operator* *query-exp*
| 'SELECT' *projection-list*
| 'FROM' *domain-list*
| ['WHERE' *predicate-exp*]
| ['GROUP BY' *path-exp-list* ['HAVING' *having-predicate*]]

projection-list ::= '*'
| *path-exp-list*
| *function-spec*
| *path-exp-list* ',' *function-spec*

path-exp-list ::= *path-exp*
| *path-exp-list* ',' *path-exp*

path-exp ::= *var-name*
| *path-exp* '.' *method-call*

method-call ::= *method-name*
| *method-name* '(' *parameter-list* ')'

| | | |
|-----------------------|-----|--|
| <i>parameter-list</i> | ::= | <i>parameter</i> <i>parameter-list</i> ',' <i>parameter</i> |
| <i>parameter</i> | ::= | <i>literal</i> <i>path-exp</i> |
| <i>function-spec</i> | ::= | <i>function-name</i> '(' <i>path-exp</i> ')' |
| <i>function-name</i> | ::= | 'MAX' 'MIN' 'AVG' 'SUM' 'COUNT' |
| <i>domain-list</i> | ::= | <i>domain-item</i> <i>domain-list</i> ',' <i>domain-item</i> |
| <i>domain-item</i> | ::= | <i>domain-spec</i> <i>domain-spec</i> <i>var-name</i> |
| <i>domain-spec</i> | ::= | <i>class-name</i> '(' <i>query-exp</i> ')' |
| <i>predicate-exp</i> | ::= | <i>operand</i> <i>comp-operator</i> <i>operand</i> <i>quantifier1</i> <i>var-name</i> 'IN' <i>operand</i> ':' <i>predicate-exp</i> <i>operand</i> <i>comp-operator</i> <i>quantifier2</i> <i>operand</i> 'EXISTS' '(' <i>operand</i> ')' <i>predicate-exp</i> 'AND' <i>predicate-exp</i> <i>predicate-exp</i> 'OR' <i>predicate-exp</i> 'NOT' '(' <i>predicate-exp</i> ')' '(' <i>predicate-exp</i> ')' <i>boolean-literal</i> |
| <i>operand</i> | ::= | <i>literal</i> <i>path-exp</i> <i>domain-spec</i> |
| <i>quantifier1</i> | ::= | 'FOR ALL' 'EXISTS' |
| <i>quantifier2</i> | ::= | 'ALL' 'SOME' 'ANY' |

comp-operator ::= '=' | '<' | '>' | '<=' | '>=' | '<>'
 | ['NOT'] 'IN'
 | ['NOT'] 'CONTAINS'
 | ['NOT'] 'IS SUBSET'
 | ['NOT'] 'HAS SUBSET'

set-operator ::= 'UNION' | 'INTERSECT' | 'MINUS'

having-predicate ::= *function-spec comp-operator operand*

literal ::= *integer-literal*
 | *real-literal*
 | *string-literal*
 | *set-literal*
 | *boolean-literal*
 | 'NIL'

set-literal ::= '{' *literal-list* '}'

literal-list ::= *literal*
 | *literal-list* ',' *literal*

boolean-literal ::= 'TRUE' | 'FALSE'

class-name ::= *identifier*

var-name ::= *identifier*

method-name ::= *identifier*

Appendix B

SQL/OO to Algebra Translator

```
%%
%token SELECT FROM WHERE GROUPBY HAVING
%token UNION INTERSECT MINUS
%token FORALL EXISTS ALL SOME ANY
%token TRUE FALSE
%token INTEGER REAL STRING IDENTIFIER
%left OR
%left AND
%left NOT
%left '=' '<' '>' 'LE GE NE IN CONTS ISSUB HASSUB
%left '-' '+'
%left '*' '/'
%left UMIN
%left '.'

%{
#include <iostream.h>
#include "mystring.h"
#include "sql2alg.h"
#define YYSTYPE string

string result;
%}
```

```
% start S
```

```
%%
```

```
S:      query_exp    { result = $1 };
```

```
query_exp:
```

```
    ( query_exp )
      { $$ = " (" + $2 + ") "; }
|    query_exp set_operator query_exp
      { $$ = $1 + $2 + $3; }
|    SELECT project_list
      FROM domain_list
      { $$ = " OLP[{" + $2 + "}]" + $4; }
|    SELECT project_list
      FROM domain_list
      WHERE predicate_exp
      { $$ = " OLP[{" + $2 + "}]" SEL[" + $6 +"]" + $4; }
|    SELECT '*'
      FROM domain_list
      { $$ = $4; }
|    SELECT '*'
      FROM domain_list
      WHERE predicate_exp
      { $$ = " SEL[" + $6 + "]" + $4; }

|    SELECT project_list
      FROM domain_list
      GROUPBY path_exp_list
      { $$ = " OLP[{" + $2 + "}]" AGG[{" + $6 + "}, "
        + get_funct($2) + "]" + $4; }

|    SELECT project_list
      FROM domain_list
      WHERE predicate_exp
      GROUPBY path_exp_list
```

```

        { $$ = " OLP[{" + $2 + "}]·AGG[{" + $8 + "}, "
          + get_funcnt($2) +"] SEL[" + $6 +"] " + $4; }
|  SELECT '*'
   FROM domain_list
   GROUPBY path_exp_list
       { $$ = " AGG[{" + $6 + "}, id ]" + $4; }
|  SELECT '*'
   FROM domain_list
   WHERE predicate_exp
   GROUPBY path_exp_list
       { $$ = " AGG[{" + $6 + "}, id ] SEL[" + $6 +"] " + $4; }
|  SELECT project_list
   FROM domain_list
   GROUPBY path_exp_list
   HAVING having_predicate
       { $$ = " OLP[{" + $2 + "}] SEL[" + $8 + " ] AGG[{"
         + $6 + "}, " + get_funcnt($2) + $4; }

|  SELECT project_list
   FROM domain_list
   WHERE predicate_exp
   GROUPBY path_exp_list
   HAVING having_predicate
       { $$ = " OLP[{" + $2 + "}] SEL[" + $10 + " ] AGG[{" + $8
         + "}, " + get_funcnt($2) +"] SEL[" + $6 +"] " + $4; }

|  SELECT '*'
   FROM domain_list
   GROUPBY path_exp_list
   HAVING having_predicate
       { $$ = " SEL[" + $8 + " ] AGG[{" + $6 + "}, id ]" + $4; }

|  SELECT '*'
   FROM domain_list
   WHERE predicate_exp

```

```

GROUPBY path_exp_list
HAVING having_predicate
    { $$ = " SEL[" + $10+ "]" AGG[{" +$8 + "}, id ] SEL["
      + $6 +"] " + $4; }
;

project_list:
    path_exp_list                { $$ = $1; }
  | function_spec                { $$ = $1; }
  | path_exp_list ',' function_spec { $$ = $1 + ", " + $3; }
;

path_exp_list:
    path_exp                    { $$ = $1; }
  | path_exp_list ',' path_exp  { $$ = $1 + ", " + $3; }
;

path_exp:
    var_name                    { $$ = $1 + "() "; }
  | path_exp '.' method_call    { $$ = $1 + $3; }
;

method_call:
    method_name                 { $$ = $1 + "() "; }
  | method_name '(' param_list ')' { $$ = $1 + "(" + $3 + ")" "
;

param_list:
    param_item                  { $$ = $1; }
  | param_list ',' param_item    { $$ = $1 + ", " + $3; }
;

param_item:
    literal                     { $$ = $1; }

```



```

    | path_exp                                { $$ = $1; }
;

function_spec:
    function_name '(' path_exp ')'           { $$ = $1 + "(" + $3 + " )" }
;

function_name:
    MAX                                       { $$ = "maximum"; }
    | MIN                                     { $$ = "minimum"; }
    | AVG                                     { $$ = "average"; }
    | SUM                                     { $$ = "sum"; }
    | COUNT                                  { $$ = "count"; }
;

domain_list:
    domain_item                               { $$ = $1; }
    | domain_list ',' domain_item           { $$ = $1 + " CP " + $3; }
;

domain_item:
    domain_spec                               { $$ = $1; }
    | domain_spec var_name                  { $$ = $1 + "#" $1; }
;

domain_spec:
    class_name                               { $$ = $1; }
    | '(' query_exp ')'                     { $$ = " (" + $2 + " ) " }
;

predicate_exp:
    operand comp_operator operand           { $$ = $1 + $2 + $3; }
    | quantifier1 var_name IN operand ':' predicate_exp
      { $$ = $1 + $2 + " ELT " + $4 + " & " + $6; }
    | operand comp_operator quantifier2 operand

```

```

        { $$ = $3 " x ELT " + $4 + " & " + $1 + $2 + " x ";
|   EXISTS '(' operand ')'
        { $$ = " EX x ELT " + $3 + " & T; }
|   predicate_exp AND predicate_exp    { $$ = $1 + " & " + $3; }
|   predicate_exp OR predicate_exp     { $$ = $1 + " | " + $3; }
|   NOT '(' predicate_exp ')'          { $$ = " ~" + $3; }
|   '(' predicate_exp ')'              { $$ = " (" + $1 + " ) "; }
|   boolean_literal                    { $$ = $1; }
;

operand:
    scalar_exp                          { $$ = $1; }
|   path_exp                            { $$ = $1; }
|   domain_spec                          { $$ = $1; }
;

quantifier1:
    FORALL                              { $$ = " FA "; }
|   EXISTS                              { $$ = " EX "; }
;

quantifier2:
    ALL                                  { $$ = " FA "; }
|   SOME                                { $$ = " EX "; }
|   ANY                                  { $$ = " EX "; }
;

comp_operator:
    '='                                  { $$ = " = "; }
|   '<'                                  { $$ = " < "; }
|   '>'                                  { $$ = " > "; }
|   'LE'                                 { $$ = " <= "; }
|   'GE'                                 { $$ = " >= "; }
|   'NE'                                 { $$ = " ~= "; }
|   'IN'                                 { $$ = " ELT "; }

```

```

|     NOT IN                { $$ = " NEL "; }
|     CONTS                 { $$ = " TLE "; }
|     NOT CONTS            { $$ = " NTL "; }
|     ISSUB                 { $$ = " SUB "; }
|     NOT ISSUB            { $$ = " NSB "; }
|     HASSUB                { $$ = " SUP "; }
|     NOT HASSUB           { $$ = " NSP "; }
;

set_operator:
    UNION                   { $$ = " UNI "; }
|   INTERSECT               { $$ = " INT "; }
|   MINUS                   { $$ = " DIF "; }
;

having_predicate:
    function_spec comp_operator operand { $$ = $1 + $2 + $3; }
;

scalar_exp:
    param_item '+' param_item      { $$ = $1 + " + " + $3; }
|   param_item '-' param_item      { $$ = $1 + " - " + $3; }
|   param_item '*' param_item      { $$ = $1 + " * " + $3; }
|   param_item '/' param_item      { $$ = $1 + " / " + $3; }
|   '-' param_item %prec UMIN      { $$ = " -" + $2; }
|   '(' param_item ')'             { $$ = " (" + $2 + " ) "; }
|   literal                        { $$ = $1; }
;

literal:
    INTEGER                  { $$ = yytext; }
|   REAL                     { $$ = yytext; }
|   STRING                   { $$ = yytext; }
|   boolean_literal          { $$ = $1; }
|   set_literal               { $$ = $1; }

```

```

;

set_literal:
    '{' literal_list '}'          { $$ = " {" + $2 + "} "; }
;

literal_list:
    literal                      { $$ = $1; }
|   literal_list ',' literal    { $$ = $1 + ", " $3; }
;

boolean_literal:
    TRUE                        { $$ = " T "; }
|   FALSE                      { $$ = " F "; }
;

class_name:
    IDENTIFIER                  { $$ = yytext; }
;

var_name:
    IDENTIFIER                  { $$ = yytext; }
;

name_name:
    IDENTIFIER                  { $$ = yytext; }
;
```