1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395 rue Wellington
Ottawa (Ontario)
K1A 0N4

Quality of the microform

Qualité de la microforme

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Modeling and Implementation in an Object-Oriented Database Management System

by

Juraj Petrás

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
May 13, 1993

© Copyright
1993, Juraj Petrás

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Author's Acknowledgement

Quatre-vingt-neuf

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84099-4

Canada

Name

JILL A. PETRAS

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COUNCIL OF EDITORS

SUBJECT TERM

--	--	--	--

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0376
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0247
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0474
Agronomy 0285
Animal Culture and Nutrition 0476
Animal Pathology 0476
Food Science and Technology 0352
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0617
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0370
Ecology 0320
Entomology 0351
Genetics 0369
Immunology 0323
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0785

EARTH SCIENCES

Biogeochemistry 0475
Geochemistry 0906

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleocology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

**Modeling and Implementation in
an Object-Oriented Database Management System**

submitted by

Juraj Petráš

in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

May 3, 1993

Abstract

This work addresses issues of applications and system modeling in object-oriented database systems. Research in object-oriented database systems has been driven by the increasing complexity of database applications. We present a list of requirements to be satisfied by an ideal object-oriented database management system (OODBMS). We propose a new notation for object-oriented database modeling. A new object-oriented database model, directly supporting relationships between objects, and an architecture of the OODBMS are proposed. A prototype implementation of a kernel OODBMS, written in the C++ programming language, along with its persistent object storage scheme is presented.

Acknowledgments

I would like to thank my wife Maria for her patience, understanding and family support which allowed me to finish this work, and to my supervisor Ekow Otoo for all his inspiring ideas, comments, and help, which he has always been willing to share. I would also like to thank Peter Choynowski for his software installation necessary for the implementation part of this thesis, and to my friends Ján Medved and Scott Reynolds for their careful readings of this thesis.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	ix
List Of Figures	x
1 Introduction	1
2 Background Information	5
2.1 Evolution of the Database Technology	5
2.2 Data Models	6
2.3 Requirements for the Relational DBMS	7
2.4 Object-Oriented Features of the OODBMS	8
2.4.1 Composite Objects	9
2.4.2 User-defined Types	9
2.4.3 Object Identity	9
2.4.4 Encapsulation	11
2.4.5 Types, Classes, and Categories	11
2.4.6 Type, Class, and Category Hierarchies	12
2.4.7 Overloading, Overriding, and Late Binding	13

2.4.8	Computational Completeness	13
2.5	Additional Requirements of the OODBMS	13
2.5.1	Active Data	14
2.5.2	Dynamic Schema Evolution	14
2.5.3	Advanced Composition Support	15
2.5.4	Flexible Transaction Policy	15
2.5.5	Data and Schema Versioning	16
2.6	Conceptual Overlaps	16
2.7	OODBMS Applications	17
3	Modeling	18
3.1	Related Models	18
3.1.1	ER Model	18
3.1.2	Extended ER Models	19
3.2	The ROB Modeling	20
3.2.1	Entities and Attributes	20
3.2.2	Relationships	22
3.2.3	Dependent Entities and Relationships	23
3.2.4	Relationships as Entities	24
3.2.5	Dependent Relationship as Entity	25
3.2.6	Aggregation	27
3.2.7	Generalization and Specialization	28
3.2.8	Disjoint-Partial "is-a" Hierarchy	29
3.2.9	Disjoint-Total "is-a" Hierarchy	30
3.2.10	Overlapping-Partial "is-a" Hierarchy	30
3.2.11	Overlapping-Total "is-a" Hierarchy	31
3.2.12	Complex Entity	32
3.2.13	Multiple Inheritance	33

4	ROB System	35
4.1	ROB Data Model	35
4.2	ROB Architecture	39
4.3	ROB Object Manager	42
4.3.1	Persistence	43
4.3.2	ROB Object Identity	44
4.3.3	ROB Persistent Objects	45
4.3.4	Access to ROB Persistent Objects	47
4.3.5	Categories	49
4.3.6	Storage Schemes for Category Hierarchies	50
5	Previous Work	54
5.1	ORION/ITASCA	55
5.2	GemStone	60
5.3	ONTOS	61
5.4	ObjectStore	64
5.5	VERSANT	66
5.6	Ode	69
5.7	O_2	71
5.8	POSTGRES	73
5.9	IRIS	75
6	Conclusion	78
6.1	Directions for Future Work	79
6.1.1	Long-Duration Transactions	80
6.1.2	Versioning	80
6.1.3	Active data	80
6.1.4	OODBMS Tools	80

List of Tables

5.1	The Comparison of the OODBMSs	77
5.2	The Comparison of the OODBMSs (Cont.)	77

List of Figures

2.1	Uniqueness of the OID	10
3.1	Entity and its attributes	21
3.2	Relationship, roles, and cardinalities	22
3.3	Dependent entity	24
3.4	Relationship as an entity	24
3.5	Example of relationship as an entity	25
3.6	Dependable relationship as an entity	26
3.7	Example of the dependable relationship as an entity	26
3.8	The “part-of” relationship	27
3.9	Disjoint-partial “is-a” hierarchy	30
3.10	Disjoint-total “is-a” hierarchy	30
3.11	Overlapping-partial “is-a” hierarchy	31
3.12	Overlapping-total “is-a” hierarchy	31
3.13	Complex entity	32
3.14	Multiple inheritance	33
4.1	The ROB diagram for the ROB data model	37
4.2	The ROB architecture	41
4.3	The storage scheme reflecting a logical view of a database	51
4.4	The storage scheme “store as low as possible”	52
4.5	The storage scheme “store where defined”	53

Chapter 1

Introduction

Object orientation is more than just the newest buzzword in computer science. Clearly it currently exerts considerable influence on the way people think about the design and implementation of large software systems. These software systems often access data that survive the execution of the program. The conventional Database Management System (DBMS) can effectively store such data - called persistent data - but they lack the expressiveness of Object-Oriented Programming Languages (OOPL). On the other hand, an OOPL lacks the effectiveness to work with persistent data. If designers want to use both expressiveness and persistence, they must access persistent data using a conventional DBMS language and then manipulate this data using an expressive OOPL, such as C++ [54], Smalltalk [23], Objective-C [16], Eiffel [36], CLOS [27], Trellis-Owl [43], or DSM [47]. The objective of an Object-Oriented Database Management System (OODBMS) is to provide both the effective access to persistent data and the expressibility.

The OODBMS research started in the early eighties and was initiated by increasing complexity of database applications. The database community has been looking for a database model richer than just the set of tables which can store only primitive data types.

The OODBMS research encompasses the following research areas:

- Object-oriented programming languages
- Complex data management
- Software engineering
- Data abstraction theories
- Artificial intelligence

The OODBMSs have been made possible by the availability of high performance workstations, networking environments, and by the growing popularity of the object-oriented paradigm. The increasing popularity and modeling power of the object-oriented paradigm ensures the viability and practical use of the OODBMS technology. Recently, the first users, who had been evaluating the suitability of OODBMSs for their applications, have started to develop real applications.

However, there are many open issues in the modeling, design, and implementation of an OODBMS. These include:

- Accepted model of an OODBMS, which supports relationships between objects, has not been defined, yet.
- Declarative interface language to the OODBMS (the object-oriented analogy of the SQL language) has not been standardized, yet.
- Techniques for the simultaneous access and modification of objects are needed for better support of cooperative engineering databases.
- Mechanism for the support of a rich object-oriented data model carries performance problems.

This thesis contributes to the object-oriented database research in three areas by proposing:

1. A new object-oriented database modeling notation that directly supports relationships between objects.

2. A new object-oriented database model that can be used effectively by applying our modeling notation.
3. A new kernel C++ implementation that reflects our object-oriented database model.

Some proposed notations are added to the Entity-Relationship¹ model (ER) [14]. We believe that our notation diagrams provide a better conceptual modeling tool for object-oriented database modeling. These are directly supported in our database model and transformable to the relational database model as well.

Our database model is based on the C++ data model and it is extended to support relationships between objects. We consider the C++ class definitions as user-defined types of objects that can be stored in a database. The database is a collection of *Categories* that are containers of persistent objects. A category (as opposed to a class) is also a vehicle for maintaining relationships between persistent objects.

Two approaches have appeared in the implementation of Object-Oriented Database Management Systems (OODBMSs):

1. Additional object-oriented features are added as another layer of an underlying relational DBMS (e. g. POSTGRES [53]) or
2. an object-oriented programming language is extended with the functionality of a DBMS (e. g. GemStone [35] or O_2 [18]).

We decided to take the approach which starts with an object-oriented language and adds database functionality because we believe that building an OODBMS from the scratch can better incorporate the object-oriented features into OODBMS and so achieve better performance.

We adopted useful features from previous generations of DBMSs. We have also added new functionality to the object-oriented paradigm that is integrated with previous generation DBMSs seamlessly. The proposed set of the OODBMS is a

¹The ER modeling notation is not standardized and it was extended in various ways [42,49,57]

superset of functionalities of previous generation DBMSs. We believe that this is absolutely necessary requirement for moving data stored in previous generation DBMSs to the next-generation DBMSs².

We have chosen the C++ language [54] for both the model base and the kernel implementation because:

- the language supports most of the object-oriented features,
- it is one of the most efficient object-oriented languages,
- it is compatible with C and this enables us to integrate all C code that has already been written, and
- it is one of the most widely used OOPs.

Our kernel prototype - the Object Manager - supports the persistence of objects, iteration, and querying, and mutation of persistent objects.

The rest of this thesis is organized as follows. Chapter 2 provides the background which is necessary to the rest of this work. Chapter 3 describes our new database modeling notation. The object-oriented database model, the architecture of our OODBMS, and the implementation of the kernel are discussed in Chapter 4. The survey of existing OODBMS's and then the comparison of our work with the related work are given in Chapter 5. Chapter 6 gives the conclusion and directions for future work.

²We refer to the next generation DBMS and the OODBMS interchangeably because the object-oriented features should be supported in all next-generation DBMSs

Chapter 2

Background Information

This chapter introduces the terminology and concepts that are necessary to understand the work presented in this thesis. For more complete introduction to OODBMSs see Kim [28]. Section 2.1 describes the origin of the OODBMS technology. Section 2.2 defines two types of data models and in Section 2.3 we examine the requirements for the OODBMS that stem from previous generations of database systems. Section 2.4 describes features of the OODBMS that come from the object-oriented paradigm. Section 2.5 describes features that extend the functionality of the OODBMS, but they are not within the object-oriented paradigm. Section 2.6 explains conceptual overlaps between different database models and Section 2.7 presents application areas of OODBMSs.

2.1 Evolution of the Database Technology

The database technology is about 30 years old. Its evolution from one generation to the next has been driven by the increasing complexity of database applications. There have been four generations in the evolution of the database technology:

1. Systems with no data sharing among many users within the application environment, such as ISAM [25] or VSAM [5].
2. Systems that provide data sharing, but the navigational access to the shared data is tedious, such as IMS [17] or CODASYL [55].

3. Relational databases with declarative SQL languages and query optimizers, such as INGRES [40] or ORACLE [39].
4. Next-generation databases, where the object oriented databases are the most important development (See examples in Chapter 5).

This evolution is very similar to the evolution of the technology of programming languages, which has been evolving from machine code programming through assemblers, non-structural languages, structural languages towards the modern object oriented languages.

2.2 Data Models

Batini et al. in [7] define a *data model* as a collection of concepts that can be used to describe a set of data and operations to manipulate the data. The concepts in a data model are typically built by using abstraction mechanism and are described through linguistic and graphic representations; that is, a syntax can be defined and a graphical notation can be developed as parts of a data model. There are two types of data models:

Conceptual models are tools for representing reality at high level. Using conceptual models, one can build a description of reality that is easy to understand and interpret. The typical example of the conceptual model is the Entity Relationship model.

Logical models support a descriptions that can be processed by a computer; they include hierarchical, relational and object-oriented models.

In a design of a database, we use conceptual models first to produce a high level description of the reality and then we translate the conceptual schema into logical schema. The motivation of this lies in the difficulty of abstracting the structure of complex databases. A *schema* is a representation of a specific portion of reality, built using a particular data model. More properly, a schema is a static, time invariant

collection of linguistic or graphic representations that describe the structure of the data interest. An *instance* of a schema is a dynamic, time-variant collection of data that conforms to the structure of that schema.

2.3 Requirements for the Relational DBMS

The OODBMS is the fourth generation in the evolution of the database technology and as such, it must preserve all requirements formulated from the previous generation DBMSs. The requirements that must be met by the third generation DBMS are:

Persistence. Data resides in a persistent storage rather than in a volatile memory, and it can be used across many program executions.

Querying. Information stored in the database can be defined and accessed by a high-level and easy-to-use interface (e. g. a declarative query language or a graphical user interface).

Security. Data is protected against unauthorized access.

Transactions. Interactions with the database are monitored to ensure the consistency (or correctness) and stability of data.

Recovery. The system is able to recover from a crash to some defined stable state.

Distributed processing. Multiple interacting users and/or applications can use the same data simultaneously.

The increasing complexity of new database applications (See examples in Section 2.7) have quickly exposed several serious shortcomings of the relational database technology. We review several of the well-known shortcomings of the conventional database technology.

1. A conventional model is too simple for modeling entities, such as design and engineering objects or complex documents.

2. Conventional database systems support only a limited set of atomic data types, such as integer, char, string, etc.; they do not support general data types found in programming languages.
3. The conventional data model does not contain a number of frequently useful semantics concepts, such as generalization and aggregation relationships.
4. The performance of conventional database systems is unacceptable for various types of computation intensive applications, such as simulation programs in CAD applications.
5. Database languages of conventional databases are different from programming languages and this causes the impedance mismatch between them.
6. Conventional databases do not support long-duration transactions necessary in interactive, cooperative design environment, the notion of time, and versions.

These limitations of conventional databases have stimulated the research in the object-oriented databases and new features for an OODBMS have been formulated. The features stem mostly from the object-oriented paradigm, however some of them originated from other research. The following sections describe both groups of features.

2.4 Object-Oriented Features of the OODBMS

The notion of an Object-Oriented Database System (OODBMS) has different meaning to different people, because there is no generally accepted definition for an object-oriented database model.

The first attempt to define an object-oriented database model was published in the paper known as the "Manifesto" [2]. The paper played a crucial role in clarifying the definition of OODBMSs. Key features of an OODBMS according to the "Manifesto" and our survey of OODBMSs are described in the following sections.

2.4.1 Composite Objects

An OODBMS deals with objects. The object has its state, represented by the instance variables, and has a set of operations that are applicable for that object. Composite¹ objects are built from components that are objects themselves. The components can be composite objects themselves, or they can be instances of primitive types (integer, real, char, . . .). The primitive types are defined in every database model. A composite object can be defined as an object with a hierarchy of exclusive component objects that form its structure.

2.4.2 User-defined Types

Users can define their own data types which, once defined, are treated exactly the same way as the built-in types. A data structure and a set of operations on a type must be defined for each new user-defined type. User-defined types can model the structure and behavior of complex real-world entities easier than built-in types only, and therefore they are more suitable for design and implementation of real applications.

2.4.3 Object Identity

Object identity, based on an object identifier (OID), characterizes the object independently of its state. An object is thus treated as a pair (OID, **state**). Two objects in the same state are treated as two different entities. This allows us to distinguish between *equality* of objects and their *identity*. The equality of two objects is based on the *equality* of object's states defined by the equality operator. (e. g. two peaches are equal if the equality operator defined for the type Peach takes these two peaches, compares their weight, color and taste and returns boolean value true).

The object identity is characteristics of object's uniqueness. Two references access the same object (i. e. they are identical) if and only if they use the same object identifier. Obviously, each OID must be system-wide unique. It is impossible

¹Composite objects are sometimes referred as complex or structured objects.

to change the OID during lifetime of the object, and even after it has been deleted. The OIDs may not be visible to the user of the system. They are rather an underlying concept for implementation of shared objects and for their correct updates in the database (see Figure 2.1). The OIDs eliminate the need for defining primary keys for entities, as would be required in the relational data model.

Figure 2.1 Uniqueness of the OID

Programming languages use addresses in volatile memory usually to identify objects. On the other hand, relational databases use attribute values to identify persistent objects, thereby using the content rather than address for identification. Both approaches have their shortcomings. The former binds the object with a particular environment where the object resides. The latter does not preserve the object identity while its attribute values are modified.

The notion of OIDs in OODBMSs solves these shortcomings because it provides the following three types of independence:

1. *Location independence* implies that the object identity is not changed by migration of the object.
2. *Value independence* means that object preserves its object identity through changes in its attribute values (i. e. its state).
3. *Type independence* involves the preservation of object's identity through changes in its type.

2.4.4 Encapsulation

Encapsulation¹ means that an object is manipulated by operations which are specified in the operator interface for the type of that object. The implementation details of the operations and the inner structure of the object are hidden. Therefore the implementation of these operations can be changed without affecting the user of these operations. The implementation of an object could be changed, for example, to improve performance, fix a bug, or consolidate code. Encapsulation is not unique to object-oriented languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in conventional languages.

2.4.5 Types, Classes, and Categories

There is a difference between a type and a class. LaLonde and Pugh [30] clarify this difference. The *type* captures the conceptual side of objects, and it specifies the behavior of a set of objects, while the *class* describes a specific implementation of a type. We add the notion of the *Category* to provide the mechanism of grouping persistent objects of the same type because we feel that it is necessary to distinguish between a specific type implementation (class) and a collection of objects of that type.

The class that represents type definition enables one to express the commonalities of a set of objects with the same structure and operations. The class provides a mechanism for creating objects.

The category enables one to store, query and manipulate its instances. A category is an object itself, and as such it has a set of attributes and methods associated with it, which apply to the category object, as opposed to the objects grouped in the category. Such attributes and methods are sometimes called *category attributes* and *category methods*, respectively. A category attribute of a category is often used

¹Encapsulation is also referred to as information hiding.

to capture an aggregate property of the instances that belong to the category. A category method is often for aggregate operations on all instances of the category.

2.4.6 Type, Class, and Category Hierarchies

A type can be defined as a subtype of other type(s). Subtyping means that an instance of a subtype can stand in for an instance of its supertype, i. e., it has all functionalities of its supertypes. All types are organized into a type hierarchy. The implementation of subtyping is not important.

A class can also be defined as a subclass of other class(es), forming subclassing (also called inheritance) relationship between the classes. Subclassing is an implementation mechanism for sharing code and representation. All classes are organized into a class hierarchy. The type and class hierarchies may or may not have the same structure.

The instances of the same class are grouped into the category that is parameterized over that class. Since an object can belong to many classes it can be a member of many categories. These categories must reflect the class hierarchy, however there can be more categories parameterized over the same class.

These three hierarchies are very often interchanged because of their interrelationships.

Most of the object-oriented languages support only the class hierarchy, however there is an effort to separate the type and class hierarchies in the C++ language [8].

Some systems allow a class to have only one superclass, while others allow a class to have multiple superclasses. The former is called *single inheritance*, the latter is called *multiple inheritance*. In a system which supports single inheritance with a single root, the classes form a tree. In a system which supports single inheritance with more than one root, the classes form a forest.

If a system supports multiple inheritance, the classes form a connected directed acyclic graph (DAG), sometimes also called a *class lattice*. There has been a lot of discussions whether multiple inheritance is really necessary. Although it complicates the data model, it appears that multiple inheritance is necessary and its acceptance

is inevitable in database model because single inheritance causes duplication of information and database applications are more difficult to model and implement with single inheritance only.

2.4.7 Overloading, Overriding, and Late Binding

Closely connected to the class hierarchy are the requirements for overriding, overloading, and late binding. Overriding means that an inherited operation can be reimplemented depending on the additional semantics of a subclass. Overloading means that the same operator name can be used for different operations in different classes. Finally, late binding captures the dynamic binding of names to operations at runtime, according to the actual type of the object on which the operation is performed.

2.4.8 Computational Completeness

The OODBMS interface language should be computationally complete. *Computational completeness* of a language means that any algorithms can be written in that language. Computational completeness of the OODBMS interface language frees users from the impedance mismatch between the language used to access persistent data and the language used to perform computations on this data.

The interface languages to the relational DBMSs are relational complete. The language is relational complete if any result of a query, which is relation, can also be specified by relational calculus or relational algebra.

2.5 Additional Requirements of the OODBMS

Resulting from ongoing research, new requirements, that do not originate from the object-oriented paradigm, have been formulated for the OODBMSs. These requirements also can apply to other database models. A short description of each of these requirements is given in the following sections.

2.5.1 Active Data

A conventional database only executes explicit transactions and therefore it is "passive" from a user point of view. However, there are additional semantics embedded in application programs that are common for all data from a certain data domain. In conventional DBMSs, these semantics are distributed across many programs. This results in poor software engineering and potential performance bottleneck if the database is polled frequently.

An active DBMS tries to overcome these shortcomings. Since data is monitored by an underlying system, semantics can be stored in one place and actions on data can be optimized. Data, that can invoke certain actions on any data stored in the database, are called *active data*. The action is triggered by a change in the state of data, which subsequently invokes the action. For example, if the quantity of the widgets in an inventory database is smaller than a certain threshold amount, say 5 widgets, order another 10 widgets.

In the OODBMS interface language, actions are parts of *triggers* or *constraints*. Triggers and constraints are language constructs that allow the objects to "reason" about the object state, consistency, and validity. The constraints are a special form of triggers that restrict the data domain of a particular object. The triggers are expressed by rules, each consisting of a predicate and an action part. If the predicate value is true, then the action is fired.

2.5.2 Dynamic Schema Evolution

Some applications require considerable flexibility in first dynamically defining and then modifying database schema, i. e., the class definitions, the inheritance structure and the specifications of attributes and methods, without shutting down the application. This behavior is necessary, because databases are often part of an inherently dynamic development environment. Even databases, which are in practical use, must often be modified because of the evolution process of the application domain. Any modification must affect the database as little as possible. This requirement

is not new in the database world. Although already supported by many RDBMS's, this requirement is much more complex in object-oriented databases because the object-oriented data model is much richer.

2.5.3 Advanced Composition Support

An object can be a part of another object. Typically, a composite object is a heterogeneous collection of component objects from a number of different classes. Usually the "part-of" relationship is expressed by an instance variable of an object. The instance variable alone may not be sufficient when extended functionality is required for a heterogeneous set of objects forming a composite object, e. g., a transaction management must be based on composition or querying based on composition must be supported. Composition is different from aggregation of homogeneous objects, which refers to collections, such as sets, bags, or arrays.

2.5.4 Flexible Transaction Policy

New applications, such as collaborative engineering (CAD/CAE or CASE), need to work on objects for a long time. Therefore, it is not possible to use only the conventional transaction management supporting only short-duration transactions. Long-duration concurrent transactions must be supported and they must be able to communicate their results to each other in the middle of their execution to check consistency using *change notification mechanism*. This requirement violates the traditional notion of serializability of transactions, which ensures the consistency of the database. The correctness of data must be preserved by the means which take into account the semantics of the transaction and data types. Long-duration transactions promote concurrent cooperative work, because many users can concurrently modify the state of the same object.

2.5.5 Data and Schema Versioning

Version control is one of the most important data-modeling requirements in next generation database applications. After an object has been created, a new version of the object can be derived from any previous version of that object. Both data and schema should be versionable so that different versions could coexist in the database, and their updates do not cause loss of the old data or data stored under the previous version of the database schema.

Versions give rise to two new types of relationships. One is the *derived-from* relationship between a new version and an old version of an object. The other is the *version-of* relationship between an object and a class of that object. Versions form a directed graph which is usually restricted to an acyclic directed graph or even, in many systems, to a tree.

2.6 Conceptual Overlaps

The above mentioned features of OODBMSs are not unique to the object oriented database model. There are some conceptual overlaps with the previous database models, namely network², functional, and semantic database models.

The concept of the nested structure of records and navigational access in hierarchical databases is very similar to the concept of complex objects in the object oriented database model. The notion of methods in OODBMSs mirrors the notion of functions from functional database models. Finally, a rich set of semantic relationship among real-world entities, such as generalization, aggregation, and composition, can be found in semantic data models.

²They include also hierarchical databases because the hierarchical databases are restricted forms of the network databases where the nested structure is limited to a hierarchy

2.7 OODBMS Applications

The database technology is expanding its territory into new application areas. The list of these application areas includes:

- computer-aided design and engineering (CAD/CAE)
- computer-aided software engineering (CASE)
- geographical information systems (GISs)
- office information systems (OISs)
- knowledge base systems (expert systems)
- multimedia
- telecommunication and network administration

All above-mentioned application domains are much more complex and it is difficult to model, design, and implement using the conventional DBMS. They all deal with complex objects, such as component parts in CAD/CAE systems, modules in CASE systems, different objects on maps that are hierarchically organized according to the map scales in GISs, structured documents used in OISs, pictures and sounds used in multimedia, or objects representing networks used in the telecommunication and network administration. I had the opportunity to look at the complexity of the data structures involved in the design of the GIS during my work at the GIS Division, Energy and Mining Resources, Canada.

All application areas are also more dynamic and therefore they need to support the features, such as active data, dynamic schema evolution, and versioning. Finally, objects of these applications are sometimes accessed for a long period of time because they are permanently modified (CAD/CAE, CASE) or large (GIS, multimedia) and therefore a flexible transaction policy must be available.

Chapter 3

Modeling

A model is an abstraction of something for the purpose of understanding it before building it. Abstraction is a fundamental human capability that permits us to deal with complexity. Models have been made for thousands of years and software design is no exception. They provide different abstract views of the system and allow analysis of structure and behavior of a system before it is actually implemented. Models are usually cheaper than building a complete system and enable flaws to be corrected early.

Section 3.1 describes the previous work in the database modeling. Section 3.2 contains a description of each construct used in Relational Object Based (ROB) modeling along with its definition and an example of its use.

3.1 Related Models

3.1.1 ER Model

The Entity-Relationship (ER) approach [14] has been the most common approach to information modeling. The ER modeling is a graphical technique that has been popular because the notation is easy to understand, and still powerful enough to model many real problems. The second advantage of the ER diagrams is their direct translation into a conventional database implementation.

The ER constructs are entities, attributes, and relationships. Entities are principal objects about which information is collected, e. g., a person, place, or event. Attributes are used to detail the entities by giving them properties, such as name, color, or weight. Finally, relationships represent real world associations between one or more entities. There are two types of attributes identifiers and descriptors. The first are used to uniquely distinguish among the occurrences of an entity, whereas the second are used to describe an entity occurrence. Entities can be distinguished by the “strength” of their identifying attributes. Strong entities have internal identifiers that uniquely determine the existence of the entity occurrence. Weak entities derive their existence from the identifying attributes¹ of one or more “parent” entities on which a weak entity is dependable. Relationships have semantic meaning, which is indicated by the connectivity between entity occurrences. The binary relationship between occurrences of entities can be one to one, one to many, or many to many. The participation in the relationship may be either mandatory or optional for the entity occurrences belonging to the entities participating in the relationship.

3.1.2 Extended ER Models

The original ER model has long been effectively used for communicating fundamental data and relationship definition with the end user. However, using the ER model as a conceptual schema representation has proved to be difficult because of the inadequacy of the initial modeling constructs for complex applications.

There are many enhancements of the initial Chen’s ER diagrams. They are all referred to as the Extended Entity Relationship (EER) models. The concept of generalization was introduced by Scheuermann et al. [44]. The concept of subset hierarchies was introduced in [20, 38]. The subset hierarchies specify possibly overlapping subsets, while the generalization hierarchies specify strictly non-overlapping subsets. Ternary relationships and composite attributes were studied by Ling [33]. Other works have concentrated on such topics as constraints [58] or on more general integrity constraints [32].

¹Identifying attributes are sometimes called external attributes.

3.2 The ROB Modeling

The ROB modeling is also an enhanced form of the ER modeling. We adopt all useful features of previous information modeling notations, unify them, combine them to introduce four new concepts to the inheritance hierarchy modeling *partial-disjoint*, *total-disjoint*, *partial-overlapping*, and *total-overlapping* relationships. These four notions are based on concepts from set theory [1]. We also introduce a new dual concept *Relationship as Entity*.

3.2.1 Entities and Attributes

An entity type² describes a group of objects with similar properties (attributes), common behavior (operations), and having common relationships to other objects. *Person*, *Department*, *Company*, or *Window* can be examples of entities. Most objects have different attributes, behavior patterns, and relationships to other objects. However, they can be grouped together depending on the application domain and purpose of modeling.

Entity diagrams provide a formal graphic notation for modeling entities. An entity diagram is a template for describing many possible instances of data. The entity diagram is depicted by a box with a name of the entity enclosed in the box.

An attribute value is a data value held by the object. Each attribute has a name which must be unique within an entity. A domain is a collection of legal values with definitions of comparison operators ($=$, $<$, $>$, $\neq$, $\leq$, $\geq$). An attribute can be from a domain that can be one of the three types:

1. atomic
2. molecular
3. multivalued

²We will use further the term *entity* instead of *entity type* where the meaning is obvious from a context.

An atomic type supports the unstructured³, smallest pieces of information in a database system. The typical examples of such types are *Boolean*, *Character*, *Integer*, *Float*, or *String*. The molecular type is a heterogeneous structure of atomic types. The structure has a fixed number of atomic elements. Users have access to the parts constituting the molecular type through a defined interface to this molecular type. Examples of such types are: *PhoneNumber*, *Date*, *Address*, etc.

The multivalued type is a homogeneous structure of atomic or molecular types. The size of multivalued attributes is variable. Users can manipulate multivalued attributes through the defined interface to this multivalued type. The examples of such types are: *languageSkills*, *degrees*, etc.

An attribute diagram is an oval connected to the entity which it belongs to. The oval contains an attribute's name and the name of its type domain. An atomic attribute diagram is a plain oval, a molecular attribute diagram is an oval with two bars at the beginning and end of the oval, and finally the multivalued attribute diagram is an oval with an opening curly bracket at the beginning of oval and a closing curly bracket at the end of the oval. See Figure 3.1 for an example of the entity *Person* with atomic, molecular, and multivalued attributes, respectively.

Figure 3.1 Entity and its attributes

Attributes are pure data values, not objects. **Unlike objects, pure data values do not have identity.** For example, all occurrences of the strings "Maria"

³Atomic types are unstructured from the user's point of view.

are indistinguishable. There may be many objects which hold this value as their attribute.

There are only descriptive attributes because there is no need for defining primary keys in an object-oriented database modeling. The concept of primary keys is replaced by object identifiers which are generated automatically by underlying OODBMS. However sometimes the identifiers have meaning in the real world and they can be legitimate attributes, such as student identification number or license plate number.

3.2.2 Relationships

Relationships⁴ are semantic connections between entities. They are usually based on activities between these entities or logical associations between the entities which are modeled. We use the Chen's original diagram for a relationship which is depicted by a diamond and is connected by lines to the participating entities. Relationships are normally binary. However, sometimes relationships of higher order exist that connect n entities ($n > 2$) and must be modeled. We called those n ary relationships. The relationship between *Mortgage*, *Bank*, and *Person* is an example of a ternary relationship.

Figure 3.2 Relationship, roles, and cardinalities

⁴We use the term *relationship* between entities, where existence of entities is not dependent on the existence of this relationship.

A relationship can appear differently for each participating entity. We support this by introducing the notion of a *role*. The role of the entity in a relationship is a name showing the semantics of the participation of this entity in the relationship. The name of the role is written on the line connecting the diamond diagram and the entity diagram and should express the meaning while the role is read starting from the entity box to the relationship diamond.

The number of entities participating in relationship can be limited for each entity. There can be minimal and maximal constraints for each role in relationship. The constraints are expressed in the form (C_{min}, C_{max}) under the line connecting the diamond diagram and the entity diagram.

The notation for our entity and relationship diagrams with the example of relationship *Employee* between *Person* and *Company* are shown in Figure 3.2.

The role of the person in the relationship is *worksFor* and the role of the company is *employs*. In our example, a person can be either unemployed (minimal constraint is zero) or can work for N companies, where N is an integer greater than zero. The company's constraints are set to $(1, M)$ and therefore company must have at least one employee. The underscore symbol means, that there is no limit on the maximal number of employees in a company.

3.2.3 Dependent Entities and Relationships

Sometimes, there is a need in modeling real-world entities and their relationships, to express a dependence of an existence of one entity on the existence of another. For example, there is obvious existence dependency between the *Student* and *Assignment* entities. If a student is removed from the database, there is no need to store his/her assignments⁵.

We use a double-edged diamond for depicting a dependent relationship. An entity, whose existence depends on this dependent relationship is depicted by a double-edged box. In addition, the notation for strong entities and relationships is valid. Figure 3.3 shows diagrams for dependent entities and relationships.

⁵Unless we want to keep them for pure statistical purposes without reference to their author.

Figure 3.3 Dependent entity

3.2.4 Relationships as Entities

Sometimes, it is useful to expand an existing relationship with semantics of an entity in order to express other relationships. For example, we model the relationship *Shipment* between a *Truck* and *Product*. It is convenient to consider the relationship *Shipment* as an entity in order to model the origin and destination of shipments.

Figure 3.4 Relationship as an entity

We introduce a new dual concept *Relationship as an Entity*. The diagram for this new concept is a diamond enclosed in a box. The actual semantic (relationship or entity) depends on the way participants are connected to the diagram. If an

Figure 3.5 Example of relationship as an entity

entity box is connected to a diamond corner of a dual diagram then a dual concept plays a relationship role. If a relationship diamond is connected somewhere to a box (we recommend that it is not connected to the common points with an enclosed diamond to avoid accidental misunderstanding) then a dual concept plays an entity role. The general diagram of the dual *Relation as Entity* diagram is depicted in Figure 3.4. The example of its use is shown in Figure 3.5.

3.2.5 Dependent Relationship as Entity

The dependent relationship is a relationship that binds the existence of the dependent entity to other entities. Sometimes it can also be convenient that this dependent relationship participates in the entity form of another relationship. We allow this and introduce a new diagram the double-edged diamond enclosed in a double-edged box. The general diagram is shown in Figure 3.6. An example of its use for modeling the *superrvision* relationship between *Teaching assistant* entity and dependent relationship as the entity *work* is shown in Figure 3.7.

Figure 3.6 Dependable relationship as an entity **Figure 3.7** Example of the dependable relationship as an entity

3.2.6 Aggregation

Aggregation is the “part-of” relationship between a composite entity and entities which form components of the composite entity. The entities which are the components in aggregation are called *constituent* entities. Aggregation is a binary relationship with these properties:

transitive: if A is part-of B and B is part-of C, then A is part-of C

antisymmetric: if A is part-of B, then B is not part-of A.

The diagram used for aggregation is a modified form of a relation diagram whereby a smaller diamond is attached directly to the entity box which represents a composite entity and is connected by a line to the entity box which represent a constituent entity. There is no need to write role names because they are obvious in this specific relationship. The cardinality constraints can be written only for a constituent entity because the other cardinality is implicitly (0,1).

Figure 3.8 The “part-of” relationship

Figure 3.8 shows a general part-of diagram and its use for a simplified model of a bicycle consisting of only a frame and wheels.

3.2.7 Generalization and Specialization

Generalization and specialization are two opposite ends of the “is-a” relationship. They are powerful abstractions for sharing similarities among entities while preserving their differences. The “is-a” relationship is transitive across an arbitrary number of levels. The terms *ancestor* and *descendants* refer to entities across multiple levels of the “is-a” hierarchy. Any operators and attributes of the ancestor can also be used by all its descendants. Moreover, descendants can add new attributes and operators that reflect their additional semantics.

The generalization is a process of creating an ancestor by extracting common attributes and behavior from its descendants. The specialization is a process of creating a descendant from its ancestor(s) with a new semantics.

Our definition of the “is-a” hierarchy of entities is based on the definition of *object morphism*⁶ in [46].

Definition 3.1 (Subentity) *Let E_1, E_2 be entities. We say that E_2 is subentity of E_1 iff object morphism $h : o_1 \rightarrow o_2$ is satisfied for all objects o_1 from E_1 and all objects o_2 from E_2 .*

There are four relationships between entities in an “is-a” hierarchy in two dimensions. In the vertical dimension, we can find either *total* or *partial* relationships, and in the horizontal dimension, we can find either *disjoint* or *overlapping* relationships. The relationships in each dimension are exclusive, i.e., only one of them can be present in each dimension.

Let us define these four relationships between entities:

Definition 3.2 (Total Relationship) *Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is a **total** relationship in the “is-a” hierarchy iff each instance of the E_1 entity is also an instance of $E_2, E_3, \dots$ or, E_n entities.*

⁶Object morphism is based on the *behavior morphism condition*, *initiative morphism condition*, *codomain morphism condition*, and *valuation morphism condition*.

Definition 3.3 (Partial Relationship) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is a **partial** relationship in the “is-a” hierarchy iff there is an instance of the E_1 entity, that is not an instance of $E_2, E_3, \dots$ and E_n entities.

Definition 3.4 (Overlapping Relationship) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is an **overlapping** relationship in the “is-a” hierarchy iff there is an instance of $E_i, i = 2, 3, \dots, n$, that is also an instance of $E_j, j \neq i, j = 2, 3, \dots, n$.

Definition 3.5 (Disjoint Relationship) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is a **disjoint** relationship in the “is-a” hierarchy iff the set of collections of instances $c(E_i)$ and the set of collections of instances of $c(E_j)$ are mutually disjoint, that is $c(E_i) \cap c(E_j) = \emptyset$, for $i, j = 2, 3, \dots, n, j \neq i$.

In modeling real world applications, these four relationships are combined in four possible combinations. We describe these four possible combinations in the following sections.

3.2.8 Disjoint-Partial “is-a” Hierarchy

Definition 3.6 (Disjoint-Partial “is-a” Hierarchy) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is **disjoint-partial** “is-a” hierarchy iff there are disjoint relationships among entities $E_i, i = 2, 3, \dots, n$ and partial relationships between E_1 and $E_i, i = 2, 3, \dots, n$.

The diagram for disjoint-partial “is-a” hierarchy is a triangle with an embedded bar. An example of such “is-a” hierarchy is in Figure 3.9. We can find a vehicle, which is neither a car nor a truck, however, no car can be a truck and vice versa.

Figure 3.9 Disjoint-partial “is-a” hierarchy

3.2.9 Disjoint-Total “is-a” Hierarchy

Definition 3.7 (Disjoint-Total “is-a” Hierarchy) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is **disjoint-total “is-a”** hierarchy iff there are disjoint relationships among entities $E_{i,j}$, $2, 3, \dots, n$ and total relationships between E_1 and $E_{i,j}$, $i = 2, 3, \dots, n$.

The diagram for disjoint-total “is-a” hierarchy is a plain triangle. An example of such “is-a” hierarchy taken from the *XView* graphical user interface toolkit is in Figure 3.10. We cannot find an instance of *OpenWindow* in an application, which is neither an instance of *Canvas* nor an instance of *Textsoftw* (Text Software Window) and there is no instance that can be instance of both *Textsoftw* and *Canvas*.

Figure 3.10 Disjoint-total “is-a” hierarchy

3.2.10 Overlapping-Partial “is-a” Hierarchy

Definition 3.8 (Overlapping-Total “is-a” Hierarchy) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is **overlapping-total “is-a”** hierarchy iff there are overlapping relationships among entities $E_{i,j}$, $i = 2, 3, \dots, n$ and total relationships between E_1 and $E_{i,j}$, $i = 2, 3, \dots, n$.

The diagram for overlapping-partial “is-a” hierarchy is a triangle with one blackened half. An example of such “is-a” hierarchy is in Figure 3.11. We can find a person, who is neither a student nor an employee, e.g. employee’s dependent, and a student can be employed.

Figure 3.11 Overlapping-partial “is-a” hierarchy

3.2.11 Overlapping-Total “is-a” Hierarchy

Definition 3.9 (Overlapping-Total “is-a” Hierarchy) Let $E_1, E_2, \dots, E_n$ be entities, such that $E_2, E_3, \dots, E_n$ are subentities of E_1 . We say that there is **overlapping-total “is-a” hierarchy** iff there are overlapping relationships among entities $E_i, i = 2, 3, \dots, n$ and total relationships between E_1 and $E_i, i = 2, 3, \dots, n$.

The diagram for overlapping-total “is-a” hierarchy is a blackened triangle. An example of such “is-a” hierarchy is from the animal realm in Figure 3.12. We cannot find a creature in the animal realm, which cannot swim, fly, or walk⁷ and there are some creatures that can move in more than one way.

Figure 3.12 Overlapping-total “is-a” hierarchy

⁷We consider all types of the movement on the ground as forms of walking

3.2.12 Complex Entity

The last modeling concept is a *complex entity* introduced by Teorey in [56]. The purpose of the complex entity is to improve the understandability of large, complex objects, which can be viewed at various levels of details. The complex entity is a group of entities, different relationships among them, and possibly other complex entities which are abstracted into a single entity for presentation in a higher level of view. The complex entities can be recursively constructed until the desirable level of abstraction is achieved.

Figure 3.13 Complex entity

The way relationships of a complex entity are transformed to relationships of its constituents, is an implementation issue. The composite entity is a special case of a complex entity containing only "part-of" relationships and therefore there are

two ways of modeling the “part-of” relationships: either using aggregation symbols described in Section 3.2.6 or as a complex entity.

The diagram of a complex entity is a hexagonal embedded in a box. The example in Figure 3.13 shows the general diagram and is a very simplified model of the *Bicycle* entity modeled as a complex entity participating in another relationship, the *distribution* with the *Dealer* entity.

3.2.13 Multiple Inheritance

*Multiple inheritance*⁸ permits an entity to have more than one super-entity and inherit attributes and functions from all parents. This allows increase the modeling expressibility and reuse of the code during implementation. The disadvantage of multiple inheritance is a loss of conceptual and implementation simplicity because it introduces name conflicts of attributes and functions defined on different paths in a hierarchy. These name conflicts must be resolved in the implementations by introducing the priority rule, i.e., the superentities are ordered and the name ambiguities are resolved according to this order, or the conflicting names must be renamed in the entity which inherits these conflicting attributes and functions. In practice,

Figure 3.14 Multiple inheritance

we should try to avoid the name conflicts in multiple inheritance as much as possible to avoid ambiguities or misunderstanding, even if an implementation language provides means for resolving such conflicts.

⁸The term multiple inheritance is also used for the conceptual relationship, even though it should be used only for its implementation. The term “multiple generalization” or “multiple specialization” would be confusing.

The multiple inheritance diagram is a diamond with two embedded crossed bars. It is created by attaching two single generalization symbols (triangles) together, because the entity E_3 "is-a" E_1 and E_2 .

One may ask whether there is a difference between the overlapping relationship and multiple inheritance. The concept of multiple inheritance is more flexible because it allows one to define new attributes and functions for the entity in addition to the inherited ones. It also enables one to specify the superentities freely. The overlapping entities can share only attributes and functions which are already defined for the entities. These can have overlapped members. The overlapping relationship is referred to sometimes as *accidental multiple inheritance* [42]. The overlapping relationship has very little support in existing object-oriented languages.

If we need additional semantics of a teaching assistant (e. g. a course which he/she supervises) who is an employee and a student and is not reflected in definitions of these two entities, we should use multiple inheritance (See Figure 3.14) instead of relying on the overlapping relationship (See Figure 3.11).

Chapter 4

ROB System

This chapter describes the Relational Object Based (ROB) database management system that directly supports the ROB modeling. Section 4.1 describes the ROB database model. Section 4.2 contains a description of the whole ROB DBMS architecture. Section 4.3 describes the implementation of one layer, namely the *Object Manager*.

4.1 ROB Data Model

The object-oriented paradigm has already proven its viability. The programming languages, such as Smalltalk, C++, or Objective C are becoming more popular and used. Even though there are already commercially available object-oriented database systems (See Chapter 5), a standard object-oriented database model is not generally accepted, yet. Our data model is another attempt to find such a model in the object-oriented database world. This is based on:

- *Atomic types:* The built-in types of C++ language (e. g. **int**, **float**, **char**, ...).
- *Molecular types:* The heterogeneous types of the fixed size built from atomic types. (e. g. **Date**, **PhoneNumber**, **Address** ...).

- *Multivalued types*: The homogeneous variable size types built from atomic or molecular types (e. g. **Degrees**, **LanguageSkills** ...).
- *User defined types*: These are defined through the C++ class definitions. Objects of these types can become persistent objects. (e. g. **Person**, **Project** ...).
- *Associations*: These support relationships between objects (e. g. **Works**, **Manages** ...).
- *Categories*: These group persistence objects and associations.
- *Messages*: These provide the interface for accessing objects.

Atomic, molecular and multivalued types can instantiate only values which can be part of transient or persistent objects. They do not have OIDs and they are not shared by more than one object. They can persist only as attribute values of persistent objects.

The C++ classes define types of both transient and persistent objects in the ROB data model. The classes that do not have any superclasses are called *base classes*. The base classes must be either *transient base classes* or *persistent base classes* (total-disjoint generalization). The transient base classes can be superclasses of *derived transient classes* and the derived transient classes can be superclasses of other derived transient classes. The persistent base classes can be superclasses of *derived persistent classes* and the derived persistent classes can be superclasses of other derived persistent classes. However, users define only one class regardless of the life-time of objects defined by this class, i. e. transient or persistent, even though there are two class definitions for transient objects and for persistent objects. The persistent class definition of the class X is created when the first *category*, parameterized over the type defined by the class X, is created.

The transient object of the type X becomes persistent by the membership of the given transient object in the category capable of storing the objects of the type X. The category is able to store and retrieve persistent objects of the homogeneous type

The categories can be either *base categories*, which hold the instances of persistent base classes, or *derived categories*, which hold the instances of persistent derived classes. Since an object can be of many types (we support multiple inheritance and overlapping membership), it can be member of more than one category. The storage schemes supporting multiple membership are described in Section 4.3.6.

Figure 4.1 The ROB diagram for the ROB data model

Objects, stored in categories, can have relationships to other persistent objects through *associations*. The associations are normal objects and provide support for implementing relationships. They should always be used instead of direct pointers

from object to objects when implementing many-to-many and one to-many relationships. Our model supports direct pointers from a persistent object to a persistent object only for one-to-one, one-way relationships. The importance of the explicit support for relationships in the object-oriented community was initially presented in [41]. There are several reasons for supporting them explicitly by association objects, in addition to conventional inter-object references using object instance variables.

- Relationship is a higher semantic construct similar to classifications, generalization and encapsulation.
- Associations are symmetric and external which enables automatic handling of an inverse link.
- Associations provide implementation independence and easier modification of the participating objects.
- There are well-understood modeling techniques which rely on relationships (all Extended-Entity Relationship modeling techniques including ROB).
- A relationship usually has attributes which describe the link between objects and must retain these attribute values.
- Explicit associations provide a certain form of normalization.

An *association* is an implementation of a relationship. The relationship can be viewed from a different participating object differently. This is supported in the definition of the association by possibly different *roles*, *cardinalities*, and *attributes* for each relationship participant. Each relationship participant is defined by its name and type. The minimal and maximal cardinalities for each participant restrict the number of the instances of the participant in the association. The role is the name of relationship from the participant point of view. The associations support relationships of any order.

For example, association **Work** contains two participants **developer** and **task**. The **developer** (instance of **Employee**) views this relationship as **worksOn**. The

number of tasks for a **developer** is limited by the cardinalities **Cmin** and **Cmax**. The relationship has from developer's point of view one attribute **startingDate** which keeps track when a developer started to work on a task.

The **task** (instance of **Project**) views this relationships as **doneBy**. The minimal and maximal number of developers working on a task is **Dmin** and **Dmax**, respectively. There is no attribute from this point of view.

```
Association work {  
  Employee developer # worksFor # Cmin # Cmax # Date startingDate;  
  Project task      # doneBy   # Dmin # Dmax;  
}
```

Our associations are also stored in categories, which are called *association categories*.

The users, according to their authorization privileges, can send *messages* that invoke functions defined in the base classes, derived transient classes, derived persistent classes, and base or derived categories.

4.2 ROB Architecture

The architecture of an OODBMS should satisfy the following requirements:

Object data model. The architecture must directly support all object-oriented features mentioned in Chapter 2.

Openness (extensibility). A system is open if developers can control, modify, or extend some part(s) of its architecture or implementation according their demands. The requirement is necessary because database applications vary significantly and therefore it is not possible to satisfy all of them with a single uniform architecture.

Transparency. Transparency must be supported in several dimensions. The user should not be aware of the actual physical location of an object. The system should hide communications and distributed computation in the distributed environment (moving objects on a network) and all transfers of objects between the main memory and secondary storage. The architecture must also support the transparent access to data stored in previous generation databases.

Reusability. Developers of applications can reuse the system components for building their systems. This requirement reduces duplication and increases the quality. A large scale of reusable components should be supported.

Performance. The architecture must provide the performance needed for new database applications (See Section 2.7) that are computationally intensive.

Interfaces. Three types of interfaces to the object-oriented database should be supported: the seamless persistent extension of an object-oriented language (most likely C++), the object-oriented extension of SQL, and a graphical user interface. These three interfaces must be supported because users of OODBMSs can come from three distinct worlds. Those are:

- Programmers using object-oriented programming languages with or without experience with structured programming languages
- Database application developers who are experienced in the use of the 3-rd or 4-th generation languages and SQL
- Database application developers with no programming experience

The ROB architecture (See Figure 4.2) consists of five layers upon which a database applications are built. Each layer provides the functionalities for the next layer above it. Different types of the database users use different interface to access the database. The description of each layer follows:

Physical Storage Manager provides various storage schemes for storing objects on secondary storage, such as sequential files, B-trees, hash tables, quad-codes,

Figure 4.2 The ROB architecture

grid files, etc. It also provides indexing on defined indexed attributes. Physical Storage Manager does not know the notion of an object. It deals only with sequences of bytes. Users do not access these functionalities directly. Only the Object Manager calls functions provided by Physical Storage Manager.

Object Manager handles objects via their object identity. It is responsible for the creation, retrieval, and updating of an object. Each object resides in an *Object Category* and relationships between objects are handled by *Association Categories*. The Object Manager binds each object with its Object Category and keeps track of the actual location of the object and all relationships of the object.

Recovery Manager supports recovery mechanisms from hardware, software, and operation failures.

Transaction Manager checks the consistency of the database by controlling transactions on the database. It supports both short-duration and long-duration transactions.

DDL/DML layer is vertically fragmented into three fragments. Each fragment is targeted for different groups of application developers. The ROB language (persistent extension of C++) is for C++ programmers demanding a DBMS support for their application, the Object SQL interface is for SQL database programmers looking for object-oriented functionalities, and the graphical conceptual modeling tool is either for inexperienced programmers or for all database developers looking for a quick and easy database design. Each DDL/DML contains a different database schema definition and querying possibilities. The ROB language defines a database as a collection of categories of the same type defined by C++ classes. It has a simple querying strategy based on iteration over categories with selection function without any query optimization. The other two interfaces are built above the Query Manager layer which provides query transformation, optimization, and scheduling. The Object SQL extends the conventional SQL language with generalization, navigational access (chained dot expressions) to objects and the possibility of invoking functions for each user-defined type. The graphical modeling tool is a window based database schema builder allowing definition, querying, and updating of a database.

The implementation part of this thesis has been focused on the *Object Manager* layer. Description of this layer follows in the next sections.

4.3 ROB Object Manager

The *Object Manager* layer is a layer of the ROB responsible for creation, retrieval, and update of an object. It manages both transient and persistent objects. The Object Manager generates the unique OID for each persistent object. Each persistent

object resides as an instance of the *Object Category*. Associations between objects are handled by the *Association Categories*. The Object Manager binds each object with its Object Category and keeps track of all associations of objects by managing the Association Categories.

4.3.1 Persistence

The ability of data to survive a termination of the program is usually achieved by a file I/O system. Often this file system is rather primitive and may be sometimes external to the built-in language features, e. g., Pascal provides just sequential files. *Transient*¹ data are created and manipulated by the programming language facilities, reside in volatile memory, and their existence ends with the termination of the program. *Persistent* data reside in the permanent store and must be transferred between the program's volatile memory and secondary storage in order to persist them at the termination of the program. This clear distinction between volatile and persistent data has two major disadvantages:

1. A considerable amount of a code deals with mapping between these two classes of data. This is about 30% (See [3]). This is an unnecessary source of programmer's errors.
2. There can be a significant difference between the data typing facilities offered by the programming languages for these two classes of data. Data must be transformed from a persistent to transient storage format when read from secondary storage. The reverse transformation must be done when data is moved from a transient memory to a persistent storage.

The main goal of the language supporting persistent objects is to free users from the above-mentioned disadvantages by seamless incorporating of persistent objects into language. Seamlessness means that the persistent objects can be treated exactly in the same way as the transient objects and they can freely change their location without any explicit programmer's transformation.

¹Transient data are also called volatile data

4.3.2 ROB Object Identity

Our implementation of object preserves three kinds of independence:

Location independence. ROB objects can move freely keeping their identity.

Value independence. ROB objects can change their state without losing their identity.

Type independence. ROB objects can move in the type hierarchy preserving their identity.

The above-mentioned requirements are satisfied by implementation of an object identity through surrogates. The surrogates are system-generated, globally unique identifiers independent of the physical location. Users do not have any control over surrogates. They cannot access their values and they are not even aware of their existence. The user-defined keys, known from conventional databases, are unnecessary. The object identifiers are uniquely assigned for only one object and they are not reused even after object removal from the system.

The persistent object identifiers are generated by the ROB Object Manager. The ROB Object Manager only has the privilege to access and generate OIDs. Users use the Category Instance Memory Descriptors (CIMDs) described in Section 4.3.1.

An object preserves its OID regardless of its location when moving within persistent storage, volatile memory, or between these two storage locations. To speed up the localization process, each persistent object knows the category it belongs to. In the case where object belongs to more than one category (i. e. subcategories with the overlapping membership), it has information about the category which is the first common category of the overlapped categories in the category hierarchy. Location information for the object with multiple membership is stored in this common category.

The category number stored for each persistent object promotes type independence. The object can mutate its type by moving in the category hierarchy. The persistent object can drop one of its type (e. g. an employee was fired and lost its

Employee type keeping only his/her **Person** type) or add some of its types which are subtypes of its actual type (e. g. a person can be hired and obtain the new type **Employee**). The portion of the class definition of the Persistent Object Identifier (POID) follows:

```
class POID : public TypedValue {  
    friend class ACategory;  
    int catID; // category ID  
    long iNum; // OID  
    ...  
}
```

Since C++ does not have run-time type checking we implemented it by deriving all implementation classes, which need run-time checking, from the base class **TypedValue**.

We do not reuse the OIDs. We believe that overhead involved in finding all references to a deleted object is inappropriate. We attach the category ID to enlarge the domain of object IDs. If each category generates a new object every second and **iNum** has 32 bits then there is more than 136 years to exhaust the counter of instance numbers. Recall that there is a difference between values and objects. The values do not have OIDs.

4.3.3 ROB Persistent Objects

There are certain principles of persistence that should govern language design as stated in [4]:

1. Persistence should be a property of arbitrary values and not limited to certain types.
2. All values should have the same rights to persist.
3. While a value persists, so should its description (i. e. type).

The first two of these principles state that a value's persistence should be independent of its type, stated as *orthogonality* of persistence to type. The third principle asserts, it should not be possible to store a value as one type and subsequently read it as another. Additionally the transfers between stores should be inferred from the operations on data. Therefore the languages providing longevity for values of all types without requiring explicit organization or data movement by the programmer are called persistent programming languages. Discovering software engineering techniques to support an arbitrary persistent structure is a difficult task. Persistence has been added to C++ in several systems [13, 31, 45]. There are basically two approaches to adding persistence to object-oriented programming languages. In one approach data is moved between memory and disk through explicit commands to read and to write. In the other approach, for the objects defined as persistent, no explicit read or write statements are necessary. The object is brought to the main memory usually before its first dereferencing.

In our implementation we have taken the second approach where no explicit read or write statements are necessary for persistent objects and persistence is completely orthogonal to type. The ROB persistent objects are based on the C++ class definition and membership in the *category* for the corresponding object type. Persistent type definitions are counterparts to transient types based on C++ classes. The persistent class is generated automatically and the database user is not aware of this parallel class hierarchy. The core of the transformation is a generation of three functions, which the Object Manager relies on, and the replacement memory pointers with *Category Instance Memory Pointers* (CIMDs) which are described in next section. The transformation follows the principle that the object of longer life-time cannot point to the object of shorter life-time. This principle is well-known already from the C language where static variable should not point to data generated on the stack after return from the function call. The list of generated functions which must be synthesized for each persistent class as follows:

```
int size();  
ostream& save(ostream& );
```

```
istream& retrieve(istream& );
```

The first function helps the Object Manager to retrieve the size of the persistent object. This is necessary because ROB objects are of a variable size and the Object Manager must relocate them physically if their sizes changed and cannot be stored in their previous locations on the secondary storage. The second function stores a persistent object on the secondary storage and the third one retrieves persistent object into volatile memory.

We took this approach because we believe that an object “knows” best itself and it is the most natural place for the definition of these operations. This approach allows us to reconstruct the virtual tables which are used in C++ for dynamic dereferencing of the pointer according to the actual type at the run-time. The `retrieve` function uses `new` operator which builds up the virtual table of the object calling the appropriate constructor and copies the object from disk, component by component, rather than copying the whole object. The object is also written component-wise, thus its virtual table is not overwritten.

Recall that users do not have to write these functions, they are generated by the C++ preprocessor which synthesizes a parallel persistent class hierarchy.

4.3.4 Access to ROB Persistent Objects

The operator overloading techniques in C++ is used for accessing persistent objects. All persistent objects are accessed through the technique of “smart pointers” [54]. In this paradigm, a smart pointer encapsulates some kind of a raw pointer or complex handle. The smart pointer overloads the indirection operators in order to be usable within normal pointer syntax. In ROB, smart pointer behavior is implemented through the class `CIMD` (Category Instance Memory Descriptor). The most important part of the definition follows:

```
class CIMD {  
    static    ObjectMngr *om;// object manager for all CIMD's  
    struct porep {
```

```
    int      count;          // number of references
    status   objMemStat;     // status of object in memory
    POID     poid;           // persistent object identifier
    pObject *inst;           // memory ptr to instance
};

porep      *p;

public:
    ...
    pObject* operator->(); // dereferencing a smart pointer
    CIMD&    operator=(const CIMD& x); // assignment
    ...
};
```

The core of the CIMD functionality lies in its ability to access both transient and persistent objects. Users use CIMD structure to access their objects regardless where the object resides. The underlying system handles dereferencing an object through the overloaded operator `->`. In the first prototype implementation, the overloaded dereferencing operator checks the presence of an object in memory using an `if` statement. This can be avoided using the proposed exception catching in C++. Since the C++ exception handling is not available yet in the latest version of C++ compiler available to us (gnu g++ version 2.3.3.) and trapping the operating system signals is neither platform independent nor an attractive technique we used this not the most efficient technique of the object localization in our prototype implementation.

In addition to double pointers, the CIMD class contains information about status of an object in memory (i. e. **locked**, **committed**, and **updated** flags). The flag **updated** is used at the time of object destruction and it signals whether the object must be written back onto the secondary storage. The other flags will be used for the future transaction processing operations on objects.

Finally, the variable **count** is used to improve efficiency of the assignment and copy operations on persistent objects. The reference counting technique is used.

The objects are not copied but only pointers to `porep` structure assigned and the value of `count` is correctly adjusted. The object destructor deletes the object from memory only in the case when no other objects refer to it (i.e. `count` is equal zero).

4.3.5 Categories

The category provides a container for persistent objects of the same type, generates and deletes its member objects, keeps type information on its member objects, provides querying functionalities over its members, and keeps track of all other categories that have some associations to that category. The class definition of the abstract class `ACategory` which is common root for all categories follows.

```
class ACategory {
    friend class ObjectMgr;
protected:
    String          name;
    int             catNum;
    long            size;
    Map<long, PDescriptor> persOffset;
    long            freePosition;
    NISList<String>  relations;
    NISList<ACategory* > bases;
    Map<String, int> members;
    String          dataFile;
    String          controlFile;
public:
    ACategory(String& n, NISList<ACategory* > superCat);
    virtual ~ACategory();
    virtual pObject* newInst(long instNum) =0;
    virtual int      writeInst(pObject *p);
    virtual pObject* readInst(POID p) =0;
```

```
virtual int      deleteInst(pObject*) ;  
};
```

The **Name** and **catNum** identify the category. The long integer **size** keeps track of the number of persistent object belonging to the category. The **Map** template **persOffset** binds logical POID with the physical location of the persistent object and **freePosition** points to the first free position where another persistent object of the category can be stored. The **relations** is not intrusive single list (NISList) template for storing all association categories which manage relations of objects of the category. The **members** NISList template contains the category IDs of objects which are data members of of the category object. These are direct POIDs for representing one-way one-to-one relationship. The **dataFile** string is the name of file where data of objects are stored physically. The **controlFile** string is the name of file where all information for constructing the category is stored. The constructor needs two parameters the name of the category and list of already constructed supercategories. The destructor of the class which is abstract must be virtual (see [54]). Finally, there are four virtual member functions for creating new persistent object of the category, saving and retrieval persistent object, and its removal from the category. Each category is parameterized over the type which it is capable of storing. We use the C++ template class definition to achieve this type parameterization. However, all definitions (i. e. data and function members) which are common for all categories are defined in abstract category **ACategory** which is the common root for all categories. Only categories parameterized over persistent type can be instantiated.

4.3.6 Storage Schemes for Category Hierarchies

The user-defined types are defined in the ROB database by C++ class definitions. The C++ classes can be organized in the class hierarchy that supports multiple inheritance. The ROB persistence objects are defined by membership in the category of corresponding type. If persistent objects are defined by the class which is a

subclass of several superclasses, the natural question about the storage scheme of these persistent objects arises.

The storage scheme for the category hierarchy which reflects the logical view of the database replicates corresponding data of an object in each category along the path of all supercategories, i. e., categories parameterized over all superclasses of the persistent object. This storage scheme is flawed with a lot data redundancies. These redundancies cause problems when objects are updated in the database because the update must be done on all replicated data. On the other hand, if users query database adopting the storage scheme that directly reflects the logical view, then retrieval of data is faster.

Figure 4.3 The storage scheme reflecting a logical view of a database

An example of the storage scheme reflecting logical view of the database is in Figure 4.3. All data defined in a class (e. g. **Person**) are replicated in all subclasses of that class (**Student**, **Employee**, and **TeachingAssistant**). We did not adopt this storage scheme.

The second storage scheme stores persistent data as low as possible in the category hierarchy that reflects the class hierarchy. In this storage scheme, data of

a persistent object are held in the category which is parameterized over the most specialized type of that persistent object.

Figure 4.4 The storage scheme "store as low as possible"

For example, all information about **Person** George, who is also **Student** and **Employee**, is stored in the category **TeachingAssistant**, which stores all persistent instances of the class **TeachingAssistant** with all their data inherited from all its superclasses.

This storage scheme supports effective updates and keeps all data about an object at one place. However, a query over a category of objects must also access all subcategories of that category to get all objects belonging to that category.

An example of the same class hierarchy represented by logical view depicted in Figure 4.3 is stored in the storage scheme "as low as possible" in Figure 4.4. We adopted this storage scheme for the generalization hierarchy that does not contain overlapping membership.

The third storage scheme, depicted in Figure 4.5, is based on the principle that the data of the persistent object are stored in the category corresponding to the class where the data was defined. In this storage scheme, a persistent object is spread among all categories which are parameterized over all superclasses of that persistent object. For example, if the attribute **Name** is defined in the class **Person**, then all persistent objects of classes **Student**, **Employee**, and **TeachingAssistant** have

their **Name** value stored in the Category which is parameterized over class **Person**.

Figure 4.5 The storage scheme "store where defined"

This storage scheme preserves update consistency and also supports associative queries over the values of persistent objects belonging to a certain category. However, the reconstruction and destruction of whole persistent object in main (transient) memory must access several categories.

We adopted a modified version of the scheme "where defined" for effective support of overlapping relationship in generalization hierarchy.

For other approaches to physical organization of object-oriented database systems see the works in [9, 29, 34, 48, 52, 59]. The implementation techniques of storage managers have been studied in [13, 26, 37].

Chapter 5

Previous Work

Research on the OODBMS started in the early 80's with the initial effort to extend the Smalltalk programming language [23] into a database system [15]. It became very active in the mid 1980's with projects such as ORION [6], Observer [50], O_2 [18], Damokles [19], IRIS [21], and EXODUS [11], to name a few. In the late 80's, a number of start-up companies were created (VERSANT, ITASCA, Objectivity, Object Design and O_2) and joined already existing object database companies (Servio and Ontos). In this chapter we look at nine OODBMS's and their solutions of following issues:

- Data model¹
- Inheritance support
- Object store management
- Composite object facilities
- Dynamic schema evolution facilities
- Transaction features and concurrency
- Versioning

¹We describe the language constructs of OODBMSs which support an object-oriented database model.

- Query management
- Authorization and security

The comparison of the solutions to above mentioned issues can be found in Table 5.1 and Table 5.2 at the end of this chapter.

5.1 ORION/ITASCA

The ORION project was initiated by the Advanced Computer Technology (ACT) Program at Microelectronics and Computer Technology in 1985. It evolved through three phases and integrated research in AI, CAD/CAM and OIS.

Data model: The data model is based on the LISP Object-Oriented Programming System (LOOPS), developed at XEROX PARC, however C and FORTRAN interfaces are also available. It supports both typed and non-typed data, library of utility classes, such as aggregates, sets, collections, etc. It also provides a shared Internet accessible repository of reusable objects for registered users, called ObjectShare.

Inheritance support: The ORION/ITASCA supports multiple inheritance where name conflicts can be resolved by:

1. *Preferential inheritance*, where the user may choose the superclass for a conflicting operations or variables to inherit from, or
2. *Rename-after-inheritance*, where the inherited variables or operations with conflicting names must be renamed in the derived class.

Object store management: ORION and ITASCA adopt a client/server architecture for data storage and distribution with object-level² access in addition to page-level³ operation access.

²the unit of the I/O operation is an object

³the unit of the I/O is a page of a memory

Composite object facilities: The ORION data model supports *part-of* relations between objects with detailed semantics for defining the composite behavior and the composite object hierarchy. A non-root class in a composite object hierarchy is called the *component class*. Each non-leaf class of the composite class hierarchy has one or more instance variables whose domains are the component classes. These instance variables are called the *composite instance variables*. A constituent object of a composite object is referenced through a composite instance variable. This reference is called a *composite reference* or *composite link*.

The composite references can be classified according to two criteria. First, they can be either *shared*, i.e., the X object is a part of the Y object and possibly of other objects, or *exclusive*, i.e., the X object is part only of the Y object. Secondly, they can be either *dependent*, i.e., existence of a subpart is dependent on the existence of its superpart, or *independent*, i.e., existence of objects is independent, references. ORION provides the following advanced composite object modeling features:

1. *Value propagation* allows the sharing of values of instance variables between composite objects
2. *Clustering* enables the storing of composite objects with their subobjects as separate storage units on a disk
3. *Redefinition* allows the transformation of a composite object into a non-composite one and vice versa
4. *Locking* allows a composite object with all its constituent objects to be a unit of the locking granularity.

Dynamic schema evolution facilities: The ORION database schema can be changed with consistency and referential constraint checking. The ORION schema modification facilities are one of the most complete and they include:

- Changes to a class attribute (add, drop, change a name, change a type, change a default value).
- Changes to a method (add, drop, change a name, modify a code).
- Change a name of class.
- Change a class hierarchy (make a class S a superclass of a class C, remove a class S from the superclass list of a class C, change the order of superclasses of a class C, add a new class, drop an existing class, create a new class as a generalization of n existing classes, partition a class C into n new classes, coalesce n classes into one new class).

Transaction features and concurrency: Each user may have at most one *private database* at any given time and have the authority to perform any operation on it. Owners of a private database may grant access to their database to other users for data sharing. The data of all users with the public access form *shared database*. The union of the user's private and shared database form a *global database* for this user. ORION supports two transactions types:

1. *Normal* transactions whose updates are reflected in the database at the commit time and
2. *Hypothetical* transactions that always aborts so that the database is not changed. It is useful for "what-if" experiments.

ITASCA provides two additional type of transactions:

1. *Short* transactions, which are atomic, independent, and serialized in execution and
2. *Long* transactions, which operate on private databases and may last for an unusually long time. The objects are *checked-out* from a shared database into a private database. Checking an object back into the shared database ends the long transaction for that object.

Versioning: ORION distinguishes three types of versions [28]:

1. *Transient* can be updated or deleted by a user who created it.
2. *Working* versions can be deleted but not updated.
3. *Released* versions can be neither deleted nor updated.

Each versionable object is derived from a *generic versionable class* which encapsulate information regarding the version count, the default version number, the next version number, and the set of version descriptors which creates the version derivation hierarchy. In addition to versions of objects, it also provides *versions of database schema* for preserving schema modifications and elaborate algorithms for mapping an object to the appropriate schema to which it belongs. Each database, *private*, *shared* and, *global*, has the default version bound to the latest version from its database. Users can select the default reference. Classes can be declared as versionable or non versionable. There are operations for the promotion from a transient version to a working version and from a working version to a released version.

Query management: ORION provides its own query language, the syntax of which is similar to Smalltalk. The query statements are formulated in terms of true object-oriented message-passing. It provides a query optimizer that automatically determines an optimal path to execute a given query. ITASCA enables the *scope* of the query to be defined by user. There are three types of query scopes:

1. *Private* scope, where the user queries the user's private database.
2. *Shared* scope, where the user queries the shared database.
3. *Global* scope, which integrates the private and shared scopes.

Authorization and security: The ORION database is secured by access codes and passwords. Different areas of the database have different access protocols and security measures. ITASCA provides an elaborate authorization scheme for controlling user operations and access to data. The authorization scope consists of three domains:

1. *Role* domain, which describes the user or group of users, specifies a subject of the authorization.
2. *Object* domain, which can be any data entity, specifies an object of the authorization.
3. *Type* domain, which specifies operations, such as read, write, generate, read definition, which can be performed on an object by a subject of the authorization.

Each of the three domains of an authorization are organized in the three hierarchies such that an authorization with a particular node implies the same authorization on all direct and indirect descendants of that node. For example, a manager should be able to access any information which the employees of the manager may access; if the user has write authorization on a class, then he should have the same authorization on any instance of the class; and if a user has the write authorization on an object, he should also have the read authorization on the object.

Authorizations can be specified in two dimensions:

1. The first dimension determines how an authorization is defined. There are two ways of its definition:
 - *Positive* authorizations describe what is allowed on database for each of the three above mentioned domains.
 - *Negative* authorizations are an explicit specifications of exceptions on each of the three domains of the authorization.
2. The second dimension specifies, whether an authorization can be overridden:
 - *Weak* authorization on a node of one of the three authorization hierarchies may be overridden at a lower level of the hierarchy.
 - *Strong* authorization cannot be overridden on a lower level of the hierarchy.

The object authorization domain is divided into the following categories:

1. *class authorization*, where the object of the authorization is a class,
2. *instance authorization*, where the object of the authorization is an instance of a class,
3. *attribute authorization*, where the object of the authorization is an attribute of an instance, and
4. *method authorization*, where the object of the authorization is a method.

5.2 GemStone

GemStone is a commercial product developed by Servio, Inc. in C.

Data model: GemStone supports the Smalltalk [23] data model using the Data Definition Language and Data Management Language (DDL/DML) called OPAL. It has all Smalltalk's features, such as pure objectiveness (everything is an object), dynamic typing at runtime, classes with their instance and class variables. The methods can belong to a class or to an instance of a class. Several kernel classes, such as sets, associations, arrays, bags, etc. are provided in OPAL for application development. GemStone also has a C/C++ interface.

Inheritance support: Classes are organized in a single class hierarchy. There is no support for multiple inheritance.

Object store management: GemStone supports a client/server architecture for data storage and distribution. A *shadow* table of the object table is created for each client from the most recent committed object table called the *shared* table. The user updates this shadow object table and the changes are reflected at the commit time by promoting the user's shadow table to the new shared table.

Composite object facilities: An object can be a part of another object, but no additional support for composition is provided as mentioned in Section 2.5.3

Dynamic schema evolution facilities: GemStone provides the definition of new classes. It also supports the addition and deletion of class and instance variables.

Transaction features and concurrency: GemStone transactions are defined interpretively at runtime. It supports both optimistic transactions, where conflicts are checked at the transaction commit time, and pessimistic transactions, where potential conflicts are avoided by checking at the transaction start time. Three locking modes are supported: read, write (other clients can read but not modify an object), and exclusive (nobody else can access an object). Locks must be acquired and released explicitly.

Versioning: No version management facilities are provided up to version 1.5C.

Query management: Queries over instances of a class or over a collections of objects are based on message passing expressed in the OPAL language. Query functions include *select*, *detect*, *in*, *reject*, etc. There is also indexing of collections based either on *equality* or *identity* of objects.

Authorization and security: The GemStone database is secured by passwords and user's ID in UNIX-like fashion. Each user has a private data space which cannot be accessed by other users unless they are given the proper privilege by the owner. Access to an individual object may be protected procedurally by application functions that check the user's ID before releasing the data.

5.3 ONTOS

ONTOS is a commercial product written in C++ developed by Ontos, Inc.

Data model: The ONTOS data model is based on the C++ object model. C++ is the DDL/DML language. Since C++ is a static, strongly typed programming language ONTOS does not provide a flexible runtime environment. It provides the library of the C++ classes for application development.

Inheritance support: ONTOS supports multiple inheritance. The name conflict in the multiple inheritance are resolved according to the order in which the base classes are specified in the definition of the derived class. Virtual (only one common instance of a base class exists for all instances of a derived class) and private (members of base class are inaccessible for a derived class) persistent base classes are not allowed in ONTOS.

Object store management: ONTOS adopts a client/server architecture for data storage and distribution with object-level access in addition to page-level access.

Composite object facilities: ONTOS provides the standard facilities for creating C++ composite objects (data member of a class can point to another object or an object of another class can be embedded in an object). In addition, it enables definitions of reverse relationships in the composite class. Various extensions to C++ for handling composite objects can be done using sets, aggregates, and other collection classes.

Dynamic schema evolution facilities: ONTOS allows some form of schema modification. New classes can be defined at runtime using the database *type* objects and then the C++ code for these objects can be automatically generated.

Transaction features and concurrency: ONTOS transactions are defined statically at compile time in the C++ code. They can be nested to 127 levels of depth. ONTOS serializes all transactions and provides a set of exceptions to handle access conflicts and the possibilities of transaction waits, aborts or deadlocks. These exceptions enable the application to deal with a problem appropriately or do something else while waiting for another transaction to complete. A runtime definition of transactions is possible through a dynamic C++ linker. ONTOS has four locking modes:

1. *read*: lock indicates read access only,

2. *write-intent*: lock is used on an object that is likely to be modified
3. *write*: lock is not used by applications, just by the internal server, and
4. *no-lock*: enables reading of objects without their locking.

There are two concurrency control strategies:

1. *pessimistic concurrency control* strategy, which prevents a *read* lock from being granted on an object that has been granted a *write-intent* lock and
2. *optimistic concurrency control* strategy, which allows a *read* lock on a *write-intent* locked object and attempts to resolve access conflicts at the transaction commit time.

ONTOS queues lock requests and raises selected exceptions in the event of lock or access conflicts. However, it does not provide the application the lock status of the object at any time. There is no change notification supported.

Versioning: No version management is implemented in Version 2.0. It is expected to be supported in Version 3.0.

Query management: ONTOS provides SQL interface to its object base. It allows some of the basic SQL statements, such as *select*, *from*, *where*, *and*, and *or* and the standard boolean and relational operators, such as *between*, *is in*, or *is not in* etc. One advantage of ONTOS Object-SQL (OSQL) is that member functions may be called within query statement to yield one or more of the parameters of the query. In addition, direct references to objects may be chained to any depth in query expressions. ONTOS queries are executed by a library class, **QueryIterator**, which takes the OSQL expression as a string argument and evaluates it. Queries are executed over instances of classes and aggregate objects. Alternatively, all queries may be implemented by the user in C++ using low-level access and iterator functions provided by ONTOS. It also allows *keyed* access to data elements in dictionary and array aggregate objects.

Authorization and security: Anybody with read access to the database file may read it and with write access they can update it freely.

5.4 ObjectStore

ObjectStore is a commercial product developed by Object Design, Inc., written in C++.

Data model: The ObjectStore data model is based on the C++ model as discussed in [54] and it extends it in several ways, e.g., the C++ operator `new` can allocate a persistent object from a persistent pool. It also supports some notion of *composite objects* and *configurations*.

Inheritance support: ObjectStore supports fully multiple inheritance as defined in [54]

Object store management: ObjectStore adopts a client/server architecture for data storage and distribution using a local area network or shared memory. It uses a virtual memory hardware to enhance the performance of accessing a persistent object without checking whether the accessed object is already retrieved from secondary storage to memory. The object can be stored in files or in disk partitions using ObjectStore's secondary storage management facilities. ObjectStore maintains a client cache of recently used pages in virtual memory of the client host to decrease the network traffic. The server keeps track of the the clients cache pages and maintains their validity. The content of each updated client's cache is written back to server after each transaction is committed.

Composite object facilities: The ObjectStore utility class `Configuration` provides composition facilities between objects which can be grouped into components. The configurations are units for versioning, locking and clustering. ObjectStore supports the definition of inverse relationships between objects.

e.g., if one is deleted the inverse relationship pointer is also deleted, using **inverse member** keyword for definition of a data member in a class. It also provides facilities for detecting dangling pointers. In this case, either an exception is raised or a dangling pointer is set to NULL.

Dynamic schema evolution facilities: ObjectStore provides limited facilities for schema evolution through a graphical user interface. It allows definition of new classes in the schema. A class can be modified only if it has no existing instances.

Transaction features and concurrency: Two types of transactions are supported:

1. *read-only* transactions obtain a snapshot copy of an requested object and they cannot modify an object state and
2. *update* transactions may update a requested object state.

Nested transactions are supported to any arbitrary depth. The access conflicts can be handled in ObjectStore either by exception handling or by aborting the conflicting transactions. It does not support shared or grouped transactions and does not have any notion of long-duration transactions. The locking is done implicitly and there is no explicit *unlock* protocol. The two-phase locking protocol is strictly enforced. The granularity of locks can be either pages or segments. ObjectStore does not support *locality* of nested locks, i.e., a lock acquired in nested transaction is not released until the highest level transaction terminates.

Versioning: ObjectStore supports powerful versioning mechanism similar to that of ORION/ITASCA. ObjectStore has a similar notion of shared and private workspace. An object can be check-out from a shared workspace to a private one and check back in back. Multiple users can share data in a cooperative fashion using various versions of the same object. It increases concurrency.

The alternative versions of the same object, which must be merged back together, reconcile their differences resulting from the parallel updates. The merging operation is left to the user to implement. Instances of any class can be versionable. The object version can be also frozen to prevent further versioning.

Query management: ObjectStore provides its own custom query language for navigation and associative retrieval of data from database collections. The general form of a query is:

collection-expression [:**integer-expression**:]

where

collection-expression is the name of a collection

and

integer-expression is the selection criteria which can include operators $<$, $>$, $\leq$, $\geq$, $=$, etc.

It provides dynamic indexing on collections and performs query optimization of both, run-time and compiled, queries.

Authorization and security: ObjectStore has its own file system with UNIX like authorization features.

5.5 VERSANT

VERSANT is a commercial OODBMS developed by Versant Object Technology Corp. implemented in C++.

Data model: VERSANT's data model is based on the C++ data model. The DDL/DML can be either C (using C interface functions to "object base") or C++. The library based on NIH C++ library [24] is provided with facilities for querying and versioning.

Inheritance support: VERSANT supports multiple inheritance as defined in C++, but all persistent classes must be inherited from `pObject` class.

Object store management: VERSANT adopts a client/server architecture for data storage and distribution.

Composite object facilities: In VERSANT, the object link from one object to another is defined by special syntax in the class definition `<type>.link`. A separate class is generated to hold the linked objects. The link class results in a level of indirection, but it allows more transparency in traversing the link. However, an attribute can also be defined to be an aggregate link, `<type>.agg`. In this case, all objects in aggregate belong to the parent object and cannot have longer lifetime.

Dynamic schema evolution facilities: New classes can be added, attributes can be added or deleted only in the leaf classes. Changes to a non-leaf class results in dropping all of its subclasses along with their instances.

Transaction features and concurrency: VERSANT supports two types of transactions:

1. *short* transactions are atomic in nature, following the two-phase commit protocol and
2. *long* transactions can last for any amount of time.

The following lock types are provided:

1. *read* locks are issued on objects that are potentially read,
2. *write* locks must be obtain before objects are updated,
3. *snapshot* creates a copy of the object regardless its lock status,
4. *monitor* allows a specification of events, e.g., release lock, acquiring lock, broken soft lock, object read, object updated, deleted or new version of an object created, which will cause the user be notified by email, if such event occurs.

Various queueing strategies are available: wait until it times out, try and no-block, or reserve place and notify when available. A lock can have two priorities:

1. *soft*: a lock having soft priority can be broken by a lock having a hard priority and
2. *hard* a lock having hard priority cannot be broken by a lock having any priority.

Locks may survive crashes (*persistent locks*) or may not (*short locks*). The locks are acquired and released explicitly, at the object level, or implicitly by some database function, such as *select*.

Versioning: VERSANT's versioning is similar to that of ORION/ITASCA. It includes the notion of release, working and transient versions, version derivation graphs, generic instances, dynamic and static binding, and integration and coordination of versions across multiple databases during check-in and check out.

Query management: Queries can be done on instances of a class via a **select** method that is defined in a **pClass**. The **pClass** is a "metaclass" that stores information about classes. A **pClass** instance, defined by **pClass.<class>**, is defined for each defined class. This **select** method takes a search condition, which can be a list of predicates to be satisfied conjunctively. These predicates are defined on attributes of the classes and can transverse links. Queries can be also specified to search recursively the subclasses' instances. To increase performance, a B-tree or hash table can be defined on the attributes of the class.

Authorization and security: The access file rights determine VERSANT's security. The object lock rights for various group of database users can be specified.

5.6 Ode

Ode (Object Database and Environment) is a database system designed by the AT&T Bell Laboratories in C++.

Data model: Ode's data model is based on C++ [54]. The database schema is a set of classes. The database is a collection of persistent objects. The DDL/DML is either O++, i.e., a superset of the C++ programming language allowing database functionality, or CQL++, i.e., a SQL like language extended with object-oriented features. They both can coexist with each other. O++ language extends C++ by allowing objects to be persistent. Every objects has an object id that uniquely identifies it. This object id is referred as a pointer to a persistent object. Each class has an associated cluster in which its persistent instances reside. Cluster can be partitioned into subclusters. O++ introduces sets as a general type constructor. A declaration of the form:

type variable [[size]]

declares **variable** to be a set of elements of type **type** with space to be allocated for **size** elements. Set variables are used as symbolic names for referencing objects which reside in clusters. They facilitate the manipulation, across query boundaries, of a single object or a set of objects. O++ provides facilities for associating constraints and triggers with objects [22]

Inheritance support: O++ supports multiple inheritance. A derived class inherits all data items and functions of its base classes. If an inherited function is declared virtual in a base class, then it can be redefined in the subclass. The name ambiguities are resolved by an explicit renaming of ambiguous names in the derived class.

Object store management: The Ode object manager is a software layer built on the EOS storage system [10]. The EOS manipulates objects as uninterpreted sequence of bytes stored on disk pages. The format of each such objects consists of an object header, i.e., tag followed by an actual object length, and

the object itself. Ode adds to this information a pointer to the object's type descriptor.

Composite object facilities: are supported by data members in class definitions of O++ or CQL++.

Dynamic schema evolution facilities: As of December 1992, the O++ or CQL++ classes can be defined only statically.

Transaction features and concurrency: are supported by an explicit transaction definition which has the form

```
trans { .... } .
```

It can be aborted using the **tabort** statement inside definition. There are several transaction events which can invoke actions defined by the following transaction triggers:

- **tbegin**, transaction begins,
- **tcomplete**, execution of the transaction is complete but it has not yet attempted to commit,
- **tcommit**, transaction commits, and
- **tabort**, transaction aborts.

All previous keywords can be used in conjunction with the qualifiers **before** and **after** to specify actions triggered by these events. The transaction management is left for the database designer who has the possibility to restore previous state of X using the **old(X)** macro. The macro **changed(X)** helps to identify the change within the course of the current transaction.

Versioning: All objects can be versionable in Ode. Ode is able to support historical databases [12]. A user can ask for the latest version or a specific version. The creation of a new object version is not automatic, after each update, but explicit upon the user's request.

Query management: O++ provides a low level iteration facility for accessing objects in a set or cluster. Ode also provides a CQL++ interface, which enables querying, updating, grouping and ordering objects. CQL++ is a declarative, SQL like interface language. A user of CQL++ can query and update a database defined and populated by an O++ program, and in particular should be able to invoke functions and triggers defined by O++. CQL++ takes as its input one or more sets of objects and returns a set of objects. Input can be interchangeably cluster, temporary cluster, or CQL++ sets. Ode also provides the graphical user interface called OdeView, which allows to perform complex operations over sets of objects, such as selection, projection, display, creation, deletion, and invocation of member functions.

Authorization and security: can be supported by constraint definitions within O++ class definitions.

5.7 O_2

O_2 was initially designed and developed by the Altair research consortium in C. The project started in September 1986. At the end of 1990, a commercial company, O_2 Technology was created. It is now developing, maintaining and marketing the product.

Data model: The data model of O_2 is based on *values* and *objects*. A value has a *type*. This type is recursively definable from atomic types (boolean, character, integer, real, string and bits) and type constructors (tuple, list, set and bag). An object has an *identity*, a value and a behavior defined by its *methods*. It belongs to a *class*. The class is defined by its type and its methods. The main difference between values and objects is in sharability. The objects can be shared by other objects while values cannot. This come from different semantics of the assignment operation, i.e., copying whole value versus obtaining just new reference to the object.

Inheritance support: Multiple inheritance is supported. The name collisions are solved by explicit renaming of ambiguous names in the derived classes. The overriding of attributes and methods is possible as long as this redefinition respects the subtyping semantics.

Object store management: The a client/server architecture is adopted. The server deals only with pages and it does not understand the concept of objects. The mapping between objects and pages is done on client's side leaving the server to perform the task connected to page management. The client work station contains the cached pages retrieved from server via remote procedure calls (RPC). The server also maintains a page cache to improve performance.

Composite object facilities: An object or value may refer to other objects via their identities. Similarly, an object or value may be composed of subvalues.

Dynamic schema evolution facilities: The O_2 Schema Manager allows dynamic class creation, modification and deletion. The deletion of class which is not a leaf in the inheritance hierarchy is forbidden. A schema designer is free in choosing the order of creating classes because temporary incomplete class definition are allowed.

Transaction features and concurrency: Concurrency control is handled with hierarchical, two-phase locking on files and pages. The global lock manager runs on the server, and the lock manager, caching the set of granted locks, runs on the workstation. The optimistic write ahead log principle⁴ is used with modification for B-trees.

Query management: The O_2 Query language is an SQL-like query language extended with capabilities to handle complex objects and values. It is subset of the O_2C languages which enables programming, database manipulation and user interface generation. O_2 Query can be used for *ad hoc* interactive queries.

⁴The write ahead logging strategy requires that the log records must be written out to the stable storage before a transaction commits

The querying is also possible by using *O₂Tool*, which is a window based environment supporting the development of *O₂* applications.

5.8 POSTGRES

The POSTGRES DBMS has been under construction at the University of California, Berkeley since 1986. It is written in C.

Data model: The fundamental notion in POSTGRES' data model is a class which is a named collection of instances of objects. Each instance has the same collection of named attributes and each attribute is of a specific type. Moreover, each instance has a unique, never changing identifier (OID). A user can create a new class specifying the class name, along with all attribute names and corresponding types. There are three kinds of classes:

1. *real*, whose instances are stored in database,
2. *derived*, whose instances are not physically stored in database and they are materialized only when necessary, and
3. *versioned*, where a class is stored as a differential relative to its parent class.

POSTGRES supports three kinds of types:

1. *base types* are any arbitrary abstract data types with user-defined conversion from and to the string data type,
2. *homogeneous sets* store instances of the same type, and
3. *heterogeneous sets* store instances of a different type.

POSTGRES provides three kinds of functions:

1. *C functions* can have arbitrary arguments and can be dynamically defined and loaded while the system is running. Functions with a class name

attribute are called methods and have similar syntax to C++ member functions.

2. *Operators* are functions with one or two operands which use the standard operator from the query language. They use indices in processing of queries and contain a hints for the query optimizer.
3. *POSTQUEL functions* are any collections of POSTQUEL commands packed together. This collection can be parameterized.

Inheritance support: A class can optionally inherit data elements and functions from other classes. Only unambiguous parent classes can be inherited. There is no attribute resolution.

Object store management: POSTGRES supports a no overwrite storage model. Using this technique the old records remain in the same database whenever an update occurs. This approach enables to model the back time travel and very fast crash recovery.

Composite object facilities: POSTGRES' composite objects can contain other objects or sets of objects as its attributes.

Dynamic schema evolution facilities: POSTGRES assumes that types, operators and functions can be added and subtracted while the system is running.

Transaction features and concurrency: POSTGRES supports standard serializability supported by page level locking and the recovery system discussed in [51].

Versioning: The time travel allows access to old versions of data.

Query management: The POSTGRES query language is called POSTQUEL. It is a set oriented query language that extends a relational SQL with support for user defined functions and operators, arrays, path (dot) expressions, nested queries, transitive closure, inheritance, and time travel.

Security and authorization: will be supported by the POSTGRES rules system in version 4.1 scheduled to release in February 1993.

5.9 IRIS

The IRIS project started in 1986 at the Hewlett-Packard Laboratories. It is written in C.

Data model: The IRIS data model is based on three constructs:

1. *objects*
2. *types*
3. *functions*

Objects represent entities and concepts from an application domain being modeled. They have their unique identity regardless of their attribute values. The data model distinguishes between *literal objects*, such as character-strings and numbers, and *nonliteral objects*, such as person or companies. Nonliteral objects are represented internally in the database by object identifiers. Literal objects have no user-accessible object identifiers and are directly represented. As such cannot be created, destroyed, or updated by the user. Objects are classified by type. The objects, belong to the same type, share common functions. Types are organized into a type hierarchy with inherited functions. Objects are described by their behavior, and can be accessed and manipulated only by means of functions. Functions can have various numbers of parameters and they provide direct support for n-ary relationships. A property is a function of one parameter, i.e., an object which is applied to, that returns a value when applied to the object. The functions can be overloaded.

Inheritance support: The IRIS supports multiple inheritance. The type structure is a directed acyclic graph.

Object store management: The IRIS object manager is built above a conventional relational storage subsystem.

Composite object facilities: No special composite object facilities except structured objects and n-ary functions, which enable n-ary relationships, are provided.

Dynamic schema evolution facilities: Type graph can change dynamically. New types can be created or existing types can be deleted. Object type mutation is supported in a type hierarchy. The addition and deletion of subtypes are limited but they plan to support it in future.

Transaction features and concurrency: The objects can be locked, unlocked, check-out, and check-in for controlled sharing among multiple users. The lock and unlock operations support conventional locking strategy. The and check-out and check-in control prevents and enables versioning of objects, respectively.

Versioning: The IRIS versioning schema is supported by *generic* instance which hold all information about the set of all object versions. It can also hold properties whose values are uniform for all versions of an object. Any reference to a versioned object can be either specific to particular version or generic which is coerced implicitly to a specific version object according to a defined strategy. Objects can be transformed from unversioned to versioned ones. Versions are created explicitly by users.

Query management: Iris queries are expressed in terms of functions and objects. Query processing consists of query translation and query interpretations. The Query Translator compiles queries from their object representation to a relational algebra representation and optimizes it. The Query Interpreter evaluates the transformed query, invoking the Storage Manager to access the database and foreign functions or to access other data sources. The Object SQL interface is provided to supported.

Table 5.1 The Comparison of the OODBMSs

<i>Feature</i>	<i>ORION</i>	<i>GemStone</i>	<i>ONTOS</i>	<i>ObjectStore</i>	<i>VERSANT</i>
<i>Data Model</i>	LOOP	Smalltalk	C++	C++	C++
<i>Inheritance</i>	multiple	single	multiple	multiple	multiple
<i>Object Man.</i>	client/server	client/server	client/server	client/server	client/server
<i>Composition</i>	advanced	basic	advanced	advanced	advanced
<i>Schema Mod.</i>	dynamic	dynamic	dynamic	static	dynamic
<i>Trans. Man.</i>	advanced	advanced	advanced	basic	advanced
<i>Versioning</i>	yes	no	no	yes	yes
<i>Querying</i>	mess. pass.	mess. pass.	OSQL C++	mess. pass.	OSQL
<i>Security</i>	advanced	UNIX-like	basic	UNIX-like	UNIX-like

Table 5.2 The Comparison of the OODBMSs (Cont.)

<i>Feature</i>	<i>Ode</i>	<i>O₂</i>	<i>POSTGRES</i>	<i>IRIS</i>
<i>Data Model</i>	C++	own	own	own
<i>Inheritance</i>	multiple	multiple	restr. multiple	multiple
<i>Object Manag</i>	EOS	client/server	no-overwrite	client/server
<i>Composition</i>	basic	basic	basic	basic
<i>Schema Modif.</i>	static	dynamic	dynamic	dynamic
<i>Trans. Manag.</i>	basic	basic	basic	basic
<i>Versioning</i>	yes	?	yes	yes
<i>Querying</i>	OSQL C++	OSQL C++	OSQL	functions
<i>Security</i>	constraints	?	rules	?

Chapter 6

Conclusion

Our research covers the following areas of the object-oriented database research:

- Object-oriented information modeling,
- Object-oriented database model,
- OODBMS architecture, and
- Implementation of the Object Manager layer of the OODBMS architecture.

We proposed new ROB modeling notations. The ROB modeling notations are extended forms of the ER diagrams. We introduced two new concepts to the previous EER diagrams. First, we proposed a dual concept *Relationship as Entity*. Relationship as an entity integrates the modeling capabilities of the relationship and entity. It simplifies modeling by allowing one to model an entity and relationship using one symbol. This symbol has dual semantics according to participants connected to. Secondly, we defined four different relationships in the "is-a" hierarchy *partial-disjoint*, *total-disjoint*, *partial-overlapping*, and *total-overlapping* relationships. These notions combine two orthogonal characteristics of the "is-a" hierarchy. The partial versus total characteristic spread along one dimension and the disjoint versus overlapping characteristics spreads along the other dimension.

We proposed ROB database model which effectively reflects our modeling notations and is based on the C++ classes that define the ROB user defined types.

associations between objects, and the notion of *Category*. The category provides the mechanism to group persistent objects of the same type together. The category also enables associative queries and modifications of a collection of objects.

We defined the ROB architecture in our OODBMS supporting the ROB database model and implemented its *Object Manager* layer. The Object Manager is responsible for creation, retrieval, and update of an object. It manages both transient and persistent objects. It is also responsible for generating also the unique OIDs for each persistent object. Each persistent object resides in the *Object Category* and associations between object are handled by *Association Categories*. Object Manager binds each object with its Object Category and keeps track of all relationships of objects by managing the Association Categories. All the above-mentioned functionalities are written in the GNU C++¹ version 2.3.3 using inheritance, templates, and operator overloading technique. The run-time type checking to C++ has been added. The functionalities were tested on a simple database created for this purpose.

The ROB DBMS development is in the prototype phase and many functionalities of the whole architecture are implemented either in a restricted form or they are not implemented, yet. The objective in the thesis is to implement the Object Manager layer. To test the functionality of this layer the underlying Storage manager layer was implemented as a collection of sequential files. The layers above Object Manager are not implemented, yet.

6.1 Directions for Future Work

There are several issues in the design of our ROB database which were impossible to finish in this project because of time constraints. We did not also cover in detail the problems related to our ROB database system mentioned in following sections.

¹GNU C++ compiler is actually called g++.

6.1.1 Long-Duration Transactions

The techniques for the support of *long-duration transaction* are subject of intensive research for CAD/CAM/CASE/GIS application environments. The way in which the end-users operate in this environments requires fundamental changes in the notion of database integrity and transaction. The conventional model is unacceptable where the duration of a transaction lasts hours or days. The reasons for such a long time transaction on objects can be either permanent modifications of objects by many developers or very large objects on which the operations last very long.

There have been a number of proposals for modeling long duration transactions. However, more research is necessary before a truly satisfactory model can be found which will remove the undesirable aspects of the conventional model of transactions and ensure database integrity.

6.1.2 Versioning

There is no consensus on any type of versions, namely, versions of single object, versions of a composite object, versions of class, and versions of schema. A few systems implemented versions (see Chapter 5) but each has its own version model which differs from system to system.

6.1.3 Active data

Active data is a very powerful concept. However, more research has to be done to integrate various proposals (O_1 , O_2) for incorporating this feature into object oriented databases.

6.1.4 OODBMS Tools

An object-oriented language as an interface to an OODBMS is a too low level tool for many developers of database applications and therefore a higher level interface is necessary. There is no standard for object SQL even though almost every existing OODBMS provides some sort of SQL-like language with object oriented extensions.

The high level graphical tool providing interactive way of definition database based on the ROB diagrams is desirable for non-programmers to design their applications. The graphical schema builder must check the semantic constraints defined for each graphical modeling symbol representing a different entities and relations. It must generate corresponding class and category definitions. This tool should also provide functionalities for defining an application user interface containing menus, forms, graphics, and possibly sounds for the end-user.

References

- [1] Jiří Adámek. *The Theory of Mathematical Structures*. D. Reidel Publishing Company, Prague, Czechoslovakia, 1983.
- [2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system Manifesto. In *Proceedings of the first International Conference on Deductive and Object-Oriented Database Systems*, 40 – 57, Kyoto, Japan, December 1989.
- [3] M. P. Atkinson, P. J. Bailey, P. Cockshott, K. J. Chrisholm, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360 – 365, 1983.
- [4] M. P. Atkinson and O. P. Buneman. Type and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105 – 190, June 1987.
- [5] W. Atkinson and P. DeSanctis. *Introduction to VSAM*. Hayden Book Co., 1983.
- [6] J. Banerjee et al. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, January 1987.
- [7] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relational Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.
- [8] Gerald Baumgartner and Vincent F. Russo. Type abstraction and subtype polymorphism for object-oriented languages. Draft, December 1992.

- [9] E. Bertino. An indexing technique for object-oriented databases. In *In Proceedings of the 7th IEEE International Conference on Data Engineering*, 160 - 170, Tokyo, Japan, February 1991.
- [10] A. Biliris and T. Panagos. The architecture of the ESO object store. Technical report, Computer Science Department, Boston University, 1991. Technical Report 91-014.
- [11] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD Conference*, 413 - 423, Chicago, June 1988.
- [12] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. J. Shekita. The architecture of the exodus extensible DBMS. In K. Dittrich and D. Unmeshwar, editors, *In International Workshop on Object-Oriented Database Systems*, 52 - 65, New York, September 1986. IEEE Computer Society Press.
- [13] M. J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Storage management for objects in EXODUS. In W. Kim and F. H. Lochovsky, editors, *In Object-Oriented Concepts, Databases, and Applications*, 341 - 369. Addison - Wesley, Reading, Mass, 1989.
- [14] P. Chen. The entity relationship model - towards a unified view of of data. *ACM Transaction on Database System*, 1(1):9 - 36, March 1976.
- [15] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings 1984 ACM-SIGMOD International Conference on the Management of Data*, 316 - 325, June 1984.
- [16] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Mass., 1986.
- [17] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass., 1986.

- [18] O. Deux et al. The O_2 system. *Communication of the ACM*, 34(10):35 - 48, October 1991.
- [19] K. R. Dittrich. Object-oriented database systems: the nation and the issues (extended abstract). In K. R. Dittrich and U. Dayal, editors, *In International Workshop on Object-Oriented Database Systems*, 2 - 4, New York, September 1986. IEEE Computer Society Press.
- [20] R. Elmasri, R. Hevner, and J. Weeldreyer. The category concepts: An extension to the entity-relationship model. *Data Knowledge Engineering*, 1(1):75 - 116, 1985.
- [21] D.H. Fishman et al. IRIS: an object-oriented database management system *ACM TOIS*, 5(1):48 - 69, January 1986.
- [22] N.H. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 327 - 336, Barcelona, Spain, 1991. Springer-Verlag.
- [23] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., July 1983.
- [24] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990.
- [25] Daniel Grosshans. *File Systems Design and Implementation*. Prentice Hall, 1986.
- [26] M. F. Hornick and S. B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on office Information Systems*, 5(1):70 - 95, January 1987.
- [27] S. E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, Mass., 1989.

2 OF/DE 2

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

- [28] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA, 1991.
- [29] W. Kim, K. Kim, and A. Dale. Indexing techniques for object-oriented databases. Technical report, Microelectronics and Computer Technology Corporation, 1987. Technical Report DB-134-87.
- [30] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming (JOOP)*, 1(1):57 - 59, January 1991.
- [31] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Object Store database system. *Communication of the ACM*, 34(10):51 - 63, October 1991.
- [32] M. Lenzerini and G. Santucci. Cardinality constraints in the entity relationship model. In G. C. Davis et al., editors, *The Entity Relationship Approach to Software Engineering*, 529 - 549. Elsevier North-Holland, 1983.
- [33] T. Ling. A normal form for entity-relationship diagrams. In *Proceedings of the 4th International Conference on the Entity-Relationship Approach*, 24 - 35, Chicago, 1985. IEEE Computer Society Press.
- [34] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In K. Dittrich and U. Dayal, editors, *In International Workshop on Object-Oriented Database Systems*, 171 - 182, New York, September 1986. IEEE Computer Society Press.
- [35] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *In Proceedings of the Object-Oriented Programming Systems and Languages Conference*, 472 - 482, July 1986.
- [36] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, N.J., 1988.
- [37] J. E. B. Moss. The design of the Mneme persistent object store. *TOIS*, 8(2):103 - 139. April 1990.

- [38] S. B. Navathe and A. Cheng. A methodology for database schema mapping from extended entity relationship models into the hierarchical model. In G. C. Davis et al., editors, *In The Entity-Relation Approach to Software Engineering*. Elsevier North-Holland, New York, 1983.
- [39] Oracle Corporation. *Oracle User's Guide*, 1983.
- [40] Relational Technology Inc., Alameda, CA. *INGRES Reference Manual Version 4.0*, November 1985.
- [41] James Rumbaugh. Relations on semantic constructs in an object-oriented language. In *Proceedings of the 2nd ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, 466 - 481, Orlando, Florida, October 1987.
- [42] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [43] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An introduction to Trellis-Owl. In *In Proceedings of the 1st ACM Conference on Object-Oriented Programming System, Languages and Applications*, 9 -16, November 1986.
- [44] P. Scheuermann, G. Scheffner, and H. Weber. Abstraction capabilities and invariant properties modeling within the entity-relationship approach. In P. Chen, editor, *In Entity-Relationship Approach to System Analysis and Design*, 121 - 140. North-Holland, Amsterdam, 1980.
- [45] D. Schuh, M. J. Carey, and D. DeWitt. Persistence in E revisited. In *In Proceedings of the 4th International Workshop on Persistent Object Systems*, 345 - 359, Martha's Vineyard, Massachusetts, September 1990.
- [46] A. Sernadas and H.D. Ehrich. What is an object, after all ? In R.A. Meersman, W. Kent, and S. Khosla, editors. *Working Conference on Object-Oriented*

- Databases: Analysis, Design & Construction (DS-4)*, Windermere, U.K., July 1990. Elsevier Science Publisher B.V. (North-Holland).
- [47] A. V. Shah, J. E. Rumbaugh, J. H. Hamel, and R. A. Borsari. DSM: An object-relationship modeling language. In *In proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Language and Applications*, 191-202, October 1989.
- [48] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *In proceedings of the 14th Annual ACM Conference on the Management of Data*, 325-336, Portland, Oregon, June 1989.
- [49] Sally Shlaer and Stephen J. Mellor. *Object-Oriented System Analysis: Modeling the World in Data*. Yourdon Press, Englewood Cliffs, New Jersey, 1988.
- [50] A. Skarra, S. Zdonik, and S. Reiss. An object server for an object oriented database system. In *Proceedings of the 1986 International Workshop on Object Oriented Database Systems*, 196-204. Computer Society Press, IEEE, April 1986.
- [51] M. Stonebraker. The design of the POSTGRES storage system. In *In proceedings of the 13th International Conference on Very large Databases*, 289-300, 1987.
- [52] M. Stonebraker. Managing persistent objects in multi-level store. In *In Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 2-11, Denver, Colorado, 1991.
- [53] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communication of the ACM*, 34(10):78-92, October 1991.
- [54] B. Stroustrup. *The C++ Programming Language*, 2nd. ed. Addison-Wesley, Reading, Mass, 1991.

- [55] R. W. Taylor and R.L.Frank. CODASYL database management systems. *ACM Computing Surveys*, 8(1):67 - 104, March 1976.
- [56] Toby J. Teorey, Guangping Wei, Deborah L. Bolton, and John A. Koenig. ER model clustering as an aid for user communication and documentation in database design. *Communications of the ACM*, 32(8):975 - 987, August 1989.
- [57] Toby J. Teorey, Dongquing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relational model. *Computing Surveys*, 18(2):197 - 222, June 1986.
- [58] N. Webre. An extended E-R model and its use on a defense project. In *Proceedings of the 2nd International Conference on the Entity-Relationship Approach*, 175 - 194, Washington, D.C., 1981. North-Holland.
- [59] M. J. Willshire. How spacy can they get ? space overhead for storage and indexing with object-oriented databases. In *In Proceedings of the 7th IEEE International Conference on Data Engineering*, 14 - 22, Tokyo, Japan, February 1991.

END

14-12-93

FIN