INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

EER-Schema Mapping Tools for Interoperable Database Representations

by

Latika Khanna, M.C.A.

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 for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
March 03, 1997

© Copyright
1997, Latika Khanna

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-22108-3

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**EER-Schema Mapping Tools for Interoperable Database
Representations**
submitted by
Latika Khanna, M.C.A.
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

February 28, 1997

Abstract

Object-Oriented (OO) Technology has influenced the fields of *Data Modeling* and *Database Design*. Scores of OO Database Management Systems have been developed to provide storage for objects created through an OO Programming Language. These systems are missing several database management features, notably, the absence of good, conceptual data modeling, concurrency control, transaction management and tools to facilitate the database design process.

The Object-Base (OB) model [Pet93] overcomes the semantic limitation of existing OO data models. The key contributions of this work are:

- Formalizing the OB model to reflect a true data model
- Design and implementation of the OB-Tool for automating the design and mapping process.
- Formalizing the mapping of Enhanced Entity Relationship (EER) modeling construct to OB representations.
- Establishing interoperability between Extended Relational and OO databases.
- Design of a new EER-Text Markup Language.

The OB-Tool and the robust modeling presented here attempt to bridge the gap between well established OO technology and traditional databases.

Acknowledgements

Foremost, I would like to thank my supervisor Prof. Ekow Otoo for his innovative ideas and patient guidance. I am also grateful to him for his constant support and the opportunity to gain some real work experience. I would also like to thank Prof. John Oommen for his help, advice and optimism.

Thanks are also due to my fellow graduate students and the superb support staff at the School of Computer Science who make this place a second home for all students.

I take this opportunity to thank my family without whom this work would not have been possible. Even though physically absent, they were with me at every cross-road. Thanks to my adopted family in Ottawa for their support and involvement. Many thanks to my patient cheerleading team in Chicago. Thank you, Mohinder for your strategic planning and for showing the twilight to a "moving train".

Contents

Abstract	iii
Acknowledgements	iv
List Of Tables	viii
List Of Figures	ix
List Of Abbreviations	xi
1 Introduction	1
1.1 Motivation	3
1.2 Results	4
1.3 Organization of Thesis	5
2 Enhanced Entity Relationship Design	7
2.1 Definition of Data Model	7
2.1.1 Types of Data Models	8
2.2 Conceptual Models	9
2.2.1 Classifications of Conceptual Models	9
2.2.2 Requirements for Conceptual Models	10
2.3 Entity-Relationship Model	11
2.3.1 Entities and Attributes	11

2.3.2	Relationships	13
2.3.3	Dependent Entity and Relationship	15
2.3.4	Relationship as Entities	16
2.3.5	Dependent Relationship as an Entity	18
2.4	Enhanced Entity Relationship Model	18
2.4.1	Generalization-Specialization Relationship	19
2.4.2	Complex Entity	24
2.4.3	Multiple Inheritance	25
2.5	Representational Models	25
3	Object-Base Data Model	27
3.1	Overview of OO concepts	27
3.2	Object-Base Data Model	28
3.2.1	Constructs	29
3.2.2	Operations	37
3.2.3	Constraints	41
3.2.4	Meta-data Storage	42
3.3	Implementation of OB constructs	42
3.3.1	Database	42
3.3.2	Categories	43
3.3.3	Operations on OB	45
3.4	Differences with the ROB Model	47
4	Context for Object Base Model	48
4.1	Motivation for the OB Model	48
4.2	Taxonomy for OO models	51
4.3	Survey of OO Data Models	52
4.3.1	ObjectStore	52
4.3.2	O_2	54

4.3.3	Ode	55
4.3.4	Versant	56
4.3.5	Orion	57
4.3.6	GemStone	58
4.3.7	IRIS	59
4.3.8	POSTGRES	60
4.4	Interoperability with the Relational Model	61
4.4.1	Compatibility with SQL3	63
4.5	Comparison with the OMG-93 Standard	66
5	OB-Tool	68
5.1	OO Database design by EER-to-OB Mapping	69
5.1.1	Sample Case to illustrate EER-to-OB Mapping	71
5.2	Overview of the OB-Tool	76
5.2.1	OB-Tool Operation	76
5.3	Implementation of the OB-Tool	79
5.3.1	EER Diagram Editor	83
5.3.2	OB Schema Generator	85
5.3.3	EER-Text Markup Language translator	86
6	Conclusion	88
6.1	Future Work	90
A	EER Text MarkUp Language Constructs	93
B	Function prototypes for the OB-Tool	95
	References	112

List of Tables

3.1	Data for a Bicycle Store	34
3.2	Class and Category specification for OB model	35
3.3	Association specification for OB model	36

List of Figures

2.1	ER diagram depicting an entity and types of attributes	13
2.2	ER diagram depicting a relationship and participating entities	14
2.3	ER diagram depicting dependent entity	16
2.4	EER diagram depicting Relationship as Entity	17
2.5	Example of Relationship as Entity	17
2.6	Dependent Relationship as Entity	18
2.7	Example of Relationship as Entity	19
2.8	EER diagram of Disjoint-Partial "is-a" hierarchy	21
2.9	EER diagram of Disjoint-Total "is-a" hierarchy	22
2.10	EER diagram of Overlapping-Partial "is-a" hierarchy	22
2.11	EER diagram of Overlapping-Total "is-a" hierarchy	23
2.12	EER diagram of Complex Entity	24
2.13	EER diagram of Multiple Inheritance	25
3.1	Three dimensional properties of an object in the OB model	31
3.2	EER diagram for the Table 3.1	32
3.3	Homogeneous objects contained within a category	33
4.1	Mapping between the OB model and the Relational model	62
5.1	EER diagram for the Encyclopedia application	72
5.2	OB-Tool main window application	77

5.3	Block diagram for module interdependency of the OB-Tool	80
5.4	Structure of OB-Tool	82
5.5	EER diagram for the EER Diagram Editor	84
5.6	EER diagram for the OB Schema Generator	85
5.7	EER diagram for the EER-text Markup Translator	85

List of Abbreviations

ADT	- Abstract Data Type
CAD	- Computer Aided Design
CAM	- Computer Aided Manufacturing
DDL	- Data Definition Language
DBMS	- Data Base Management System
ER	- Entity-Relationship
EER	- Enhanced Entity-Relationship
OB	- Object-Base
ODE	- Object Database and Environment
OMG	- Object Management Group
OO	- Object Oriented
OODB	- Object Oriented Database
OODBMS	- Object Oriented Data Base Management System
OOPL	- Object Oriented Programming Language
RDBMS	- Relational Data Base Management System
SQL	- Structured Query Language

Chapter 1

Introduction

Object-Oriented (OO) technology has gained widespread popularity in application development. It has proved to be a powerful and practical approach for the development of software systems including Database Management Systems (DBMS). Among its many benefits are significant improvements in modularity, reusability, flexibility and extensibility. The benefits of OO technology have been sufficiently enumerated in the relevant OO literature [Kim90a, Cat91]. The database community has taken advantage of the OO approach and has made attempts to adopt the OO paradigm in all phases of database application development. Its influence can be seen in database modeling, database design and analysis to database languages. There are several good reasons for the influence of OO techniques in the areas of:

1. Data Modeling:

- The OO concepts are much closer to the human mental model of the real world and express hierarchical and complex relationships significantly.
- The OO concepts overcome the inability of traditional data models to meet the data management requirements of many complex applications.

2. Database Design:

Existing design methods which are based on some variant of the Entity-

Relationship model make it easy to replace entities with the powerful notion of objects. Objects as opposed to entities, can additionally represent behavior and composition.

3. Database Management Systems (DBMS):

Advent of Object-Oriented Programming Languages (OOPL) required objects to be stored persistently and managed using advanced features of present day DBMS. Traditional DBMS are limited in their abilities to manage complex objects and to support the principles of object-orientation.

Scores of Object-Oriented Database Management Systems (OODBMS) were developed to meet the requirements for a DBMS for OOPLs. However, their success was limited due to:

1. Absence of a strong object-oriented data model.
 - In most OODBMS, the data model is only an implementation of the OO programming concepts. Specifically they allow the user to create classes with attributes and methods, support inheritance between classes, create instances with an object identifier, retrieve instances, load and run methods [Kim90b].
 - OO data models do not meet the important conceptual requirement of modeling relationships and constraints between two distinct entities explicitly.
 - Management of extensional data in OO models is questionable. Under the Relational Model, it is straightforward to keep all data in the form of extensions of the defined relations.
 - Meta-data management should conform to the rules and constraints of the data model.
2. Query facilities on object collections are restricted. Early OODBMS support only navigational access to meet the needs of early applications. The com-

combination of associative access and navigational access is indispensable in any database application.

3. Current OODBMS do not support interoperability with earlier database systems. Relational DBMS will stay for a long time and bi-directional co-operation between the two Database Management Systems is missing.

1.1 Motivation

In this thesis, we formalize the Object-Base (OB) model [Pet93] to be a true implementation data model for OODBMS. The OB model is an attempt to standardize the OO data model for OODBMS by:

1. overcoming the shortcomings of the current OO data models discussed in the preceding section.
2. supporting both associative and navigational access on object collections.
3. introducing additional features to improve the query facilities on object collections
4. supporting inter-operability with the traditional data models
5. managing meta-data in consistent form as regular data.

Absence of good conceptual tools for OO database design is a practical aspect responsible for the lukewarm success of current OODBMS. Various methodologies for the design of database application systems have been proposed. The Entity-Relationship (ER) diagrams [Che91] are a popular technique because of their modeling capability and simplicity. The ER model has been extended to incorporate missing features and adopt the OO concepts. The extended model is called the Extended Entity Relationship (EER) Model [DNF79, SSW79, Web81, TYF86]. We extend the EER model further by proposing two additional constructs to support further semantics. We implement a tool to support our set of EER constructs and

their representations. The tool also support database design by translating the EER model to:

1. The OB model schema and
2. Textual representation of EER model.

1.2 Results

The Object-Base (OB) model [Pet93] is a class based model for objectbases. It has been defined within the framework of the data model definition proposed under section 2.1. The OB model extends the concepts of the C++ Language to meet the following requirements for OO data models:

- Makes a clear distinction between the extensions and intensions of data. Class definitions are defined as data intensions. Collection of class instances are extension of *Categories*. *Categories* are new constructs defined within the OB model as containers of homogeneous, persistent objects with the flexibility to organize objects in a variety of pre-defined organization structures.
- Makes a clear semantic distinction between the "part-of" relationship and relationships between two or more entities. The *Association* construct maintains the semantics of the information being modeled, eliminates the need for representing relationships through references and provides support to represent relationships of order higher than two.
- Operations are defined for the associative and navigational access of object collections within *Categories*. Additionally, categories enhance the query performance on object collections by clustering objects, supporting a cache for previously retrieved objects, and maintaining a collection of user-defined indexes for ad-hoc queries.

- Meta-data for the OB model requires maintaining information of all instances of categories created in an object-base under a meta-category. The meta-category being a category itself, conforms to the rules for data storage, operations and constraints defined for the OB model.
- Uni-directional support for interoperability between the relational model to the OB model is supported easily. However, data in the OB model cannot be made compatible to the existing relational data model because the capabilities of the existing Relational data model need to be upgraded. Features to support molecular data, storage of procedures and a non-procedural language that provides support to manage both need to be built into present day RDBMS for a full bi-directional interoperability. Examples of such are presented in SQL and POSTGRES95.

The OB-Tool is a prototype conceptual tool that promotes database design for object-oriented database. It supports both a graphical and textual interface for EER diagrams. The textual interface is provided through a new EER-Text Markup Language which promotes portability, uniformity, and case tool independence. The OB-Tool also eases the transition from conventional databases to OO database by mapping the popular EER model to their equivalent OB representations.

1.3 Organization of Thesis

In Chapter 2 of this thesis, we give the definition for data models, briefly describe the types of data models and types of conceptual data models. ER models are a popular conceptual model. We present the basic constructs for ER model and their enhanced versions with illustrative examples. We also propose two additional concepts to further enhance the EER model.

In Chapter 3, we introduce the Object-Base data model and describe the constraints and operations applicable on it. We briefly summarize the implementation details for the OB model.

In Chapter 4 we introduce the motivation behind the Object-Base model and discuss the classification of OO models on the basis of their underlying model used for persistent storage. A survey and comparison of few popular OO DBMS with the OB model is covered. Furthermore, we compare the conformance of the Object based model to the industry standards and SQL3.

Chapter 5. details the translation of EER model to the OB model together with a sample case study. The design and implementation of our Conceptual Modeling tool is presented.

In Chapter 6, we summarize and point out areas for future work.

Chapter 2

Enhanced Entity Relationship Design

It is important that the interpretation of the real world be described in terms of data. A data model is an intellectual tool that provides an interpretation of reality in a sufficiently abstract and powerful form. The role of the data model is to facilitate understanding of real world data elements and their inter-relationships. For an accurate representation of the real world, the data model should cater to recording of data (i.e. facts) and its interpretation (i.e. meaning) together.

In Section 2.1 we provide a formal definition of a data model and review types of conceptual models under section 2.2. Under section 2.3 we review the ER model and its variants as the popular conceptual modeling tool. We extend the Enhanced Entity Relationship (EER) model by introducing two new modeling constructs under section 2.3.4 and section 2.3.5.

2.1 Definition of Data Model

Formally, a data model M may be defined as a tuple $\langle S, C, O, K \rangle$ of four elements, where

- S is a set of rules for Structure specification
- C is a set of rules for Constraint specification

- O is a set of Operations on the structures
- K is a set of rules for Meta-data specification

Structure specifications define the allowable structures for the data within the data model M . Constraint specifications define the restrictions for the data models. Constraints may be explicitly stated or implied inherently through S . Operations are allowable actions that can be performed on the data within the context of the structure specifications. Meta-data specification define the *apriori* configuration of information about the data model M . Meta-data should be designed to follow the structure specification S , constraint specifications C , and allow the application of valid operations O defined for the data model M .

2.1.1 Types of Data Models

Classification of data models can be done on the basis of the level of abstraction the model represents as well as the target user group. The higher level provides a more abstract view of the data and its inter-relationships, shielding the user from the actual implementation level complexities.

- High-level or Conceptual Models
- Representational or Implementational Models
- Physical Models

High-level, conceptual data models provide concepts that are close to the way users perceive data. High level data models use concepts such as entities, attributes and relationships. An entity represents a real world object. An attribute represents some property of interest that further describes an entity. A relationship, among two or more entities, represents an interaction amongst the entities.

Representational Models provide concepts that may be understood directly by the user, yet are not too far removed from the direct implementation of data within

the computer. They hide some details of data storage but can be implemented on a computer system in a direct way. The three most widely used data models that fall under this category are - **relational, network and hierarchical**.

Physical data models describe data in the form it is stored within the computer by representing information such as record format, record orderings and access paths. DBMS Designers and Implementors use the physical model for the design and implementation of the DBMS.

2.2 Conceptual Models

Contributions to the field of conceptual modeling have come from research areas of Artificial Intelligence, Programming Languages and of Databases [BMS84]. The influence of the database field led to the development of database models such as the hierarchical, network and relational model. However, more powerful modeling concepts and techniques were needed for modeling data-intensive systems based on simple, logical reasoning instead of building data structures leading to the development of Conceptual Models.

2.2.1 Classifications of Conceptual Models

Conceptual models have been classified by [LZ92] based on their modeling orientations as either:

- Semantic Model
- Process Model
- Event Model

Semantic Data Modeling is concerned with the static and structural aspects of the information. It is a formal mechanism for representing objects, relationships and constraints in the Real World. Two basic approaches dominate semantic modeling:

the entity-relationship and binary-relationship approaches. The entity relationship approach led to the development of the Entity-Relationship (ER) model [Section 2.3] and the Infological model [Lan73]. The binary-relationship formalism avoids the distinction between attributes and relationships. This approach has given rise to models such as the Semantic Data Model [HM78] and NIAM [NH89].

Process Data Models are concerned with the specification of activities in an application area of the Universe of Discourse. The formalism is dominated by "data flow" models found in software engineering.

Event Data Models are concerned with modeling the relationship between time and information. Two kinds of temporal information which need to be modeled have been identified; namely - events and transaction times.

2.2.2 Requirements for Conceptual Models

An ideal conceptual model should fulfill the following requirements:

- **Implementation Independence**

The specifications should be devoid of all implementation aspects like data representation, organization and access methods.

- **Abstraction**

Abstraction involves the ability to emphasize details essential to the problem domain and to suppress all irrelevant details [BMS84]. Abstractions allow for organization of elements into hierarchies, provide constructs for semantic relationships and design of complex objects.

- **Formality**

Concept descriptions should be stated in a formalism with unambiguous syntax which allows one to relate the descriptions in the formalism to the world being modeled.

- **Constructibility**

Provide facilities for gathering and representing knowledge in a natural and

convenient fashion in order to be easily understood by the user and the analyst. A graphical representation facilitates understanding and representation of the modeled domain.

- Ease of analysis

A conceptual model needs to be analyzed in order to determine whether it is ambiguous, incomplete or inconsistent.

- Traceability

Support mapping procedures for generating implementation level specification.

2.3 Entity-Relationship Model

The Entity-Relationship model(ER) is a popular conceptual model. It is supported by a formal graphic notation called ER diagram. The constructs and semantics of the ER model may be mapped to different representational models [NC83, TYF86]. This thesis formalizes the mapping of ER Model to an object-oriented representational model called the Object-Base (OB) Model [Pet93]. It also describes the implementation of the Object-Base tool which supports the creation of EER diagrams and its conversion to equivalent OB representations.

The Entity-Relationship (ER) model was proposed by [Che91]. It meets the requirements of a good conceptual model and finds widespread applicability in the requirement phase of application development. Its popularity lies in its constructibility, traceability, ease of analysis and implementation independence.

The basic ER model supports three classes of data objects - **entities**, **attributes** and **relationships**.

2.3.1 Entities and Attributes

Entities are principal data objects, about which data is to be collected. They denote a person, place or event of informational interest. The older term used was

"entity set" to describe a group of objects with similar properties and behavior. A particular occurrence of an entity is called an entity instance or an entity occurrence.

Attributes provide description details about the entities. Attributes can be assigned to relationships as well. A particular occurrence of an attribute within an entity or a relationship is called an attribute value. Attribute names must be unique within an entity. A domain is a collection of legal values for an attribute. An attribute can be either of the following types, based on its value:

- atomic versus composite
- stored versus derived
- simple-valued versus multi-valued

An atomic attribute supports the smallest piece of information in a database system. The typical examples of such types are *Boolean*, *Character*, *Integer*, *Float* or *String*. The composite type is a heterogeneous structure of atomic types. The structure has a fixed number of atomic elements. Examples of such types are *Date*, *PhoneNumber*, *Address* etc..

Most attributes have a single value for an entity; they are called single-valued. The multivalued type is a homogeneous structure of atomic and composite attributes. The size of the multivalued attribute is variable. The examples of multi-valued attributes are such types as *languageSkills*, *degrees* etc.

Derived attributes have values that are derivable from either other related attributes or entities. For example the attribute *Age* of a *Person* can be determined from another attribute *Birthdate* and current system date. The former attribute *Age* is derived from *Birthdate* which is a stored attribute.

All attributes for an entity are either descriptors or key attributes. Key attributes have a distinct value for each entity instance of an entity type. The values of the key attributes are used to identify each entity instance uniquely. Descriptors are attributes that specify additional characteristics of the entity.

The entity construct is depicted by a rectangle in ER diagrams with its name written inside. An attribute is represented in an ER diagram by an ellipse connected

Figure 2.1: ER diagram depicting an entity and types of attributes

to the entity or relationship it describes. The ellipse contains an attribute's name and the name type of it's domain. An atomic, single-valued, stored attribute is a plain ellipse, a composite attribute has its name is enclosed within two bars and a multivalued attribute name is enclosed within curly brackets. Derived attributes are represented by a broken ellipse. See Figure 2.1 for an example of an entity *Person* with atomic, composite, multivalued and derived attributes, respectively. The value of *SIN* is unique for each person, hence it is a key attribute and the others are Descriptors.

2.3.2 Relationships

Relationships represent real world associations among one or more entities, and as such have no physical or conceptual existence other than that which is inherited from their entity association. A particular occurrence of a relationship is called a relationship instance or relationship occurrence.

In ER diagrams, relationships are represented by diamond-shaped boxes connected by straight lines to the entities participating in the relationship. Each relationship is assigned a unique name, describing the association, written within the box [See Figure 2.2].

Figure 2.2: ER diagram depicting a relationship and participating entities

Relationships are described in terms of degree, structural constraints and roles of the participating entities. The degree of a relationship is the number of entities participating in the relationship. In Figure 2.2, the degree of the relationship *worksFor* is two. Relationships can be of any degree (n-ary), but the binary ones occur most commonly. Relationships may have certain constraints that are determined from the miniworld situation that the relationship represents. These restrictions are represented in the form of structural constraints in the ER model. Structural constraints restrict the possible combination of entities that may participate in relationship instances. Two main types of relationship constraints can be distinguished:

- **Cardinality ratio** which specifies the number of instances the entity can participate in;
- **Participation constraint** which specifies whether the existence of an entity depends on it being related to another entity through the relationship type. Participation constraint may be total or partial and is also known as existency dependency.

Structural constraints can be expressed in the form (C_{min}, C_{max}) under the lines connecting the relationship to the participating entity in ER diagrams.

Role is the function an entity plays in a relationship. The role name signifies the role of the entity in a relationship. The rolename is a comprehensive, semantic construct for knowledge representation in the model. In ER diagrams, rolenames are written above the lines connecting the participating entities to the relationship.

In Figure 2.2, the relationship *worksFor* depicts the association between two entities *Person* and *Company*. The degree of the relationship is two. The role of the *Person* is *worker* in the relationship *worksFor* and that of the *Company* is *employer*. Each *Person* can be unemployed at any point in time or work for N companies. In, which case it's structural constraints are represented as $(0, N)$. A minimum value of 0 indicates, that unemployed *Persons*, are not related to any *Company* and subsequently, do not participate in the relationship; A max value of N indicates that a person can work for a maximum of N companies. On the other hand, each *Company* must have at least one person working for it and can employ an unlimited number of persons. This is represented through the structural constraint of $(1, *)$; where $*$ indicates unlimited participation in the relationship. A min value of 0 in a structural constraint represents partial participation, indicating the possibility of certain entity instances not participating in the relationship at any point of time. A minimum value of 1 represents total participation; that every entity must be represented at least once in the relationship instance.

2.3.3 Dependent Entity and Relationship

It is sometimes necessary to model the dependency of an entity on another entity in the real-world. The former dependent entity is also referred to as weak entity. In such cases, the existence of a weak entity and its relationship are dependent upon the existence of the independent entity. For example, there is obvious existency dependency between *Student* and *Assignment*. If a student is removed from the database, there is no need to store his/her assignments¹.

¹The use of this information depends upon the scope of the model: It may be considered irrelevant and discarded or stored for pure, statistical purpose

Figure 2.3: ER diagram depicting dependent entity

We use a double-edged diamond box for depicting the dependent relationship in ER diagrams. An entity whose existence depends upon the dependent relationship is depicted by a double-edged rectangular box. Figure 2.3 also shows an example of a dependent entity and dependent relationship.

Notice, that the dependent entity type always has a total participation constraint, with respect to its dependent relationship, because a dependent entity cannot be identified without its owner entity.

2.3.4 Relationship as Entities

The new dual concept of *Relationship as an entity* is introduced. Certain situations in the miniworld being modeled find the description of associations very restrictive. The semantic requirements call for treatment of the relation as an independent entity. An example of such a situation is the relationship *Shipment* between *Truck* and *Product*. The *Shipment* can be considered as an identity to model the origin and destination of each shipment instance.

The *Relationship as Entity* construct can be depicted in ER diagrams as a diamond enclosed within a rectangle box. The diamond within a rectangle denotes the relationship within an entity, to signify the Relationship as Entity construct. The semantics of attaching the entities to the Relationship symbol within the box in the

Figure 2.4: EER diagram depicting Relationship as Entity

Figure 2.5: Example of Relationship as Entity

ER diagram plays an important role. If the entity is attached to the corner-points of the diamond within the enclosed box, then the dual concept plays a relationship role. If either a participating entity or relationship is not connected to the either corner points of the diamond box, but to some other point of the bounding box, then the dual concept holds. The general diagram for the dual *Relationship as Entity* is given in Figure 2.4 and Figure 2.5 presents an example of it's use.

Figure 2.6: Dependent Relationship as Entity

2.3.5 Dependent Relationship as an Entity

Dependent relationship as discussed in the preceding section, binds the existence of a dependent entity to an independent entity. The preceding notion of *Relationship as Entity* can be extended to cover, dependent relationships too. It is denoted by a double-edged diamond enclosed within an double-edged rectangular box, depicting the dependent relationship within a dependent entity. The importance of connecting the participating entities to the correct points of the double-edged diamond holds again (Refer to previous section). The general diagram is depicted in Figure 2.6. An example of its use in modeling is shown in form of the *Supervision* relationship between the *Teaching Assistant* entity and *Submit* relationship in Figure 2.7.

2.4 Enhanced Entity Relationship Model

The Enhanced Entity Relationship (EER) Model is an extension of the ER model discussed in the previous Section 2.3. Need for an enhanced model was felt to incorporate additional semantics for modeling the more complex applications of engineering design, cartographic and geological databases among others.

Several additional constructs were proposed to be included in the initial proposal of the ER diagrams. The concept of generalization and aggregation was proposed

Figure 2.7: Example of Relationship as Entity

in [SS77]. The concept of subset hierarchies was introduced in [NC83]. The subset hierarchies specify overlapping subsets, while the generalization hierarchies specify strictly non-overlapping subsets. Ternary relationships and composite attributes were studied by Ling in [Lin85]. Constraints were studied in [Web81]. Other extensions were proposed by [SSW79, DNF79, TYF86, GH91].

2.4.1 Generalization-Specialization Relationship

Generalization is the process of extracting common attributes and behavior from several types of entities into a generic higher level entity type called superclass entity type or supertype. In other words, a Subclass is a subset of another entity called the Superclass. Subclass and Superclass represent the same entity in the miniworld.

Generalization can also be described in terms of inheritance, which specifies that all attributes of a superclass are propagated down the hierarchy to entities of a lower type.

Specialization is the same concept as Generalization but in reverse. Specialization is a top-down abstraction process as opposed to bottom-up abstraction as done for Generalization. Specialization is the process of creating subclasses from a superclass. New attributes and operators may be added to the subclasses to reflect additional semantics.

Both, Generalization and Specialization represent the "is-a" relationship of the miniworld. The "is-a" relationship is transitive across an arbitrary number of levels. The "is-a" relationship is needed in the following situations:

- Certain attributes may apply to some but not all entities of the superclass entity type
- Some relationship types may be participated in only by entities that are members of the subclass

The "is-a" relationship can be further classified by two important constraints on the subtype entities: Disjointness and Completeness.

- **Disjointness** requires that subtype entities be mutually exclusive; Subtype entities not disjoint, are called **overlapping**.
- **Completeness** constraint requires that subtype entities be all inclusive of the supertype entities. Subtype entities can be classified to provide **total** or **partial** coverage of the supertype entity.

The two constraints can be combined to give four different combinations of the "is-a" hierarchy, in real-world applications:

- Disjoint-Partial
- Disjoint-Total
- Overlapping-Partial
- Overlapping-Total

Each of the constraints has been defined as follows in [Pet93] ;

Definition 2.4.1 (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 $E_i, i = 2, 3, \dots, n$.*

Figure 2.8: EER diagram of Disjoint-Partial "is-a" hierarchy

The diagram for the disjoint-partial "is-a" hierarchy is a triangle with an embedded bar, as depicted in Figure 2.8. An example of its application is depicted there too. *Vehicle*, *Car* and *Truck* are the participating entities with *Car* and *Truck* being subentities of *Vehicle*. The disjointness constraint holds between the subentities *Car* and *Truck*. An entity instance of *Car* does not belong to *Truck*. The Partial condition holds for the "is-a" hierarchy as all instances of *Vehicle* are not necessarily *Cars* or *Trucks* only.

Definition 2.4.2 (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 $E_i, i = 2, 3, \dots, n$ and total relationships between E_1 and $E_i, i = 2, 3, \dots, n$.

The Disjoint-total "is-a" relationship is depicted by a plain triangle. Figure 2.9 depicts the syntax for the "Disjoint-Total is-a" hierarchy and its example between the entities *Person* and *Adults* and *Minors*. Disjointness constraint holds between the subentities *Adults* and *Minor*. A person cannot be an adult and a minor². The "is-a" relationship between *Person* and its subentities is Total, because all instances of *Person*, have to be subclassified as either *Adults* or *Minors*.

²Here all persons above the 18 year age group are considered adults, else minors

Figure 2.9: EER diagram of Disjoint-Total "is-a" hierarchy

Figure 2.10: EER diagram of Overlapping-Partial "is-a" hierarchy

Figure 2.11: EER diagram of Overlapping-Total "is-a" hierarchy

Definition 2.4.3 (Overlapping 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 **overlapping-partial "is-a" hierarchy** iff there are overlapping 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 Overlapping-partial "is-a" relationship is depicted by a half-blackened triangle. Figure 2.10 depicts the syntax for the "Overlapping-Partial is-a" hierarchy. An example of such "is-a" hierarchy is found between the superentity *Person* and subentities *Student* and *Employee*. The subentities can overlap, with a person being both a student and an employee. However, there may be certain Persons, who are neither, so the hierarchy is Partial.

Definition 2.4.4 (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 Overlapping-total "is-a" relationship is depicted by a blackened triangle. Figure 2.11 depicts the syntax for the "Overlapping-Total is-a" hierarchy. An example of such "is-a" hierarchy is found in the animal world., where all creatures

Figure 2.12: EER diagram of Complex Entity

can either, swim, fly or walk³. There are some who can do more than one form of movement.

2.4.2 Complex Entity

The complex entity groups entities with different relationships among them, into a single entity for a higher-level view. The concept was first introduced in [TGBK89]. The complex entities can be recursively constructed, until a desired level of abstraction is achieved. The complex entity facilitates comprehension and understanding of large, complex objects which can be viewed at various levels of detail.

The diagram for a complex entity is a hexagon embedded in a rectangular box to depict an entity. Figure 2.12 shows the syntax with an example of *Bicycle* entity modeled as a complex entity participating in the *Sales* relationship with another independent entity - *Supplier*.

³Here all types of movement on the land is considered as walking

Figure 2.13: EER diagram of Multiple Inheritance

2.4.3 Multiple Inheritance

In the "is-a" relationship discussed previously, the subclass entity types have only a single entity as their superclass. Multiple inheritance is a construct, to depict a subentity with more than one entity as its parent. Under multiple inheritance, the subentity inherits the attributes, relationships and functions of all its parents. Multiple inheritance facilitate model comprehension at the expense of conceptual and implementation simplicity.

The multiple inheritance diagram is a circle with two embedded, crossed bars, to depict two-generalization symbols (triangles) together. These two generalizations denote that two or more entities form a "is-a" hierarchy with a subentity depicted at the lower-level of the diagram.

2.5 Representational Models

Representational models are also know as Implementational models because they represent concepts directly understood by the user, yet are close to direct implementation of data within the computer system. A database is categorized on the basis of its implementational model. The three most traditional data models that fall

under this category are - **relational, network and hierarchical**. Object-Oriented (OO) databases need to be represented by a similar representational model supporting OO concepts. Chapter 3 gives an overview of OO concepts and discusses OO data models.

Chapter 3

Object-Base Data Model

Conceptual models and languages should be influenced in the future (as they were in the past) by the research areas of databases, programming languages and artificial intelligence. In these three areas, the object-oriented programming approach has rapidly become crucial to the information industry. They have incorporated many new ideas and well established concepts such as information hiding into a coherent set of rules for data structures and data operations.

In this chapter, we briefly review the basic OO-concepts of Programming languages under Section 3.1. Section 3.2 describes our model of object-bases called the Object-base data model. In Section 3.3 we describe the implementation of its major constructs in the C++ language. Section 3.3.3 deals with the language for definition and operation on the model.

3.1 Overview of OO concepts

The object-oriented approach generally includes the four characteristics of identity, classification, polymorphism and inheritance [RBP⁺91]. *Identity* means that data is composed of discrete things called objects; objects can be physical things like entities in the real world or concepts. *Classification* is the declaration of objects in classical OO Programming languages. A *Class* is a type definition which

specifies the structure(attributes) and the behavior(operations) for objects. Each object is then considered an instance of a class, and each instance of a class has its own value for each attribute but shares attribute names and operation names with other objects of that class. Operations that pertain to a particular class are called *methods*. Classification includes the concepts of data abstraction and encapsulation, the separation of the object's identity and function from the implementation details of that function.

Polymorphism is the ability of a given object or operation to assume different forms. The functional view of Polymorphism, is the ability of the same operation to behave differently in different classes [RBP⁺91]. Finally *inheritance* in classical OO is a relationship between classes where one class is the parent class of another. It is the characteristic that attributes and operations among object classes can be shared in a hierarchical relationship. Each class can be divided into subclasses; each subclass inherits the attributes and functions of the superclass in addition to defining its own unique properties. In OO programming, inheritance is often referred to in terms of code reuse because similar classes can be made to use common code in certain situations.

More comprehensive coverage of OO concepts can be found in [RBP⁺91, Weg90].

3.2 Object-Base Data Model

The Object-Base (OB) model is a class-based object-oriented data model for Object-bases. It is an extension of the C++ language. An Object-base is a collection of objects. The OB model is based on the ROB Data Model constructs proposed by Petras [Pet93]. Section 3.4 summarizes the difference in presentation of the ROB model and the OB model.

The Object-Base management system needs to deal with two types of objects:

- **Transient** objects that exist within the scope of the program or process that created it. Transient objects are typically stored on the stack or the heap.

- **Persistent objects** has a lifetime that exists beyond that of a program by being permanently stored on a storage device. Persistent objects can be shared across applications, and have to be explicitly deleted once created.

The Object-Base data model is a conceptual model for persistent objects. When referring to objects during our description of the OB model, we mean persistent objects in the context of an object-base. An object-base provides persistent storage for objects.

3.2.1 Constructs

The OB model supports the following constructs:

- Values
- Objects
- Classes
- Associations
- Categories

Values have types. They may be *atomic* or *molecular*. Further, they may be classified as being *simple* or *multivalued*.

- *Atomic types* are built in types supported by C++ language namely `char`, `int`, `long`, `float`, `double`..... These singular, non-decomposable types are also referred to as simple-valued types or *literals*.
- *Molecular types* are user defined heterogeneous, fixed size types. They are recursively definable from atomic types and type constructors.
- *Multivalued types* are user-defined, variable size, homogeneous types. They are formed by applying the collection type constructors to the atomic or molecular types.

A **Domain** is a set of data values of the same data type. Domains of nonatomic types are usually identified by a name. An object is characterized by its properties or **Attributes**. Attributes are defined on some underlying domain. That is, they can assume values from the set of values in the domain.

Objects represent entities and concepts from the application domain. Every entity in the mini-world has its own unique identity even though it ceases to exist. The properties of the entity may be changed, or replicated but the identity of the entity is considered immutable. The OB model supports these real-world semantics by assigning system generated, immutable identifiers to every new object.

Values do not have identifiers and are distinguishable from objects. Values are parts of objects and cannot be shared by more than one object. To distinguish between an object and a value, assigning a value yields a copy operation and assigning an object gives a new reference to the same object.

Every object in the OB model is described by the the following three-dimensional properties [Figure 3.1]:

- i. descriptive properties
- ii. functional properties
- iii. compositional properties

Description provides characterizing details of an object. The Descriptive property of an object is the group of all attributes describing the current state of an object. The descriptive part is responsible for data storage in an object.

Function details the behavioral aspects of an object. Functional property details the various functions that the **object may perform on itself**. Only functions defined for an object can manipulate its descriptive properties. Objects support the object oriented concepts of encapsulation and

Figure 3.1: Three dimensional properties of an object in the OB model

abstraction. The descriptive properties are encapsulated within objects with restricted access through its functions.

Composition of an object defines the individual components that are 'part-of' of the complex object. A complex object may be composed of n components each of which may be complex in its own right. The composition aspect of an object is multivalued. Composition properties define the 'part-of' semantics of an entity in the application domain.

The first two properties are mandatorily defined for any object. All three properties are recursively defined for any object.

Class is a specification of the descriptive, functional and compositional properties of an object. It defines the intension for an object in an object-base. Classes support descriptive and functional inheritance. They can be organized in taxonomies in which the more specialized objects inherit the data and functions of their ancestor.

Classes contain a Constructor function for object instantiation and a corresponding destruction function for its deletion. Generally, the constructor has the same name as the class for which it is defined. The destructor mirrors the

Figure 3.2: EER diagram for the Table 3.1

class name too but is preceded with a $\sim$.

The ER diagram for a Bicycle Store application is presented in Figure 3.2. The class specifications for the Bicycle and Supplier entities are illustrated in Table 3.2.1.

Associations are classes that support relationships between classes. They represent the relationships between participating entities in the real world. Supporting associations provides for:

- n-ary, bi-directional relationships, apart from the "is-a", relationships inherent in the OO models.
- capturing the relationship semantics of the the traditional database applications as a single conceptual entity
- support for referential integrity through association memberships; explicit support for the declaration and enforcement of inverse relation constraints depends on the system

Figure 3.3: Homogeneous objects contained within a category

- specification of participation and cardinality constraints.
- ability of an association to act as an independent object with structure and function definition.

The descriptive aspect of an association contains declarations for describing the relationship between the participating classes and additional data attributes. The functional aspect declares the additional functions that may be performed by the specific association class.

Categories are container classes for grouping homogeneous, persistent objects [Figure 3.3]. Categories are powerful constructs, providing a choice of grouping and access mechanism for storing the contained objects. Categories provide a higher level of abstraction, encapsulation and are independent of the collection types used for their implementation.

Categories can be directly mapped to the persistent store through "a category to a cluster" concept. In addition, categories contain an inbuilt cache for facilitating querying on contained objects.

An object, can be made persistent by explicit insertion into a category or by making it a component of a complex persistent object. The latter approach is called *Persistence through reachability*. Associations too can be made persis-

SUPPLIER

S#	SName	SStatus	SCity
S1	Smith	10	London
S2	Jones	7	Paris
S3	White	3	London

BICYCLE

B#	BMake	BModel
B1	Peugot	221
B2	Triumph	118
B3	Peugot	007

FRAME

F#	FMake	FDesc
F1	Peugot	T-crossbar
F2	Triumph	Universal

WHEEL

W#	WMake	WSize
W1	Anony	24
W2	Anony	21
W3	Anony	24

Table 3.1: Data for a Bicycle Store

tent through their storage into categories.

The **state** of the database reflects a particular state of the mini-world being modeled. It is the set of valid object occurrences stored in the database at a given point in time. An operation on the database transfers the database from one valid state to another.

Figure 3.2 shows the ER diagram for a wholesale Bicycle Store. The Bicycle Store maintains an account of the bicycles sold by them to the various petty suppliers. Bicycles need to be assembled before they are sold. Assembling requires attaching the Bicycle frame to two, compatible bicycle wheels in stock. Four independent entities can be identified for the Bicycle Store; namely BICYCLE, SUPPLIER, FRAME and WHEELS. Bicycles sold are complex entities comprising of its component frame and wheels. The relationship SALES identifies the bicycles sold to a supplier by the Store.

Table 3.1 shows a set of the sample data for the Bicycle Store arranged in tables for each entity. Each tuple in the table can be visualized as an object in the OB

Class Specification

```

class Supplier{
    long OID;
    char sno[4];
    char sname[31];
    int sstatus;
    char scity[21];
public:
    Supplier(supplierno);
    ~Supplier();
    SCity();
    SStatus();
    ...
}

class Bicycles{
    long OID;
    char bno[4];
    char bmake[31];
    int bmodel;
    List< classWheels > wheellist;
    List< classFrame > framelist;
public:
    Bicycle(bikeno);
    ~Bicycle();
    ...
}

```

Category specification

```

Category< Supplier, Set > supplierCategory;
Category< Bicycle, Set > bicycleCategory;

```

Table 3.2: Class and Category specification for OB model

model. Bicycles would be complex objects comprising of two wheel objects and a frame object. Table 3.2.1 shows the class declaration for the Supplier and Bicycle entity under the OB model. The Bicycle Class declaration contains a declaration for Wheel and Frames within its structure declaration to represent the composition *part – of* relation. The relation between the entities Supplier and Bicycle is represented by the *Sales* relationship. Its declaration as an Association in the OB model is shown in Table 3.2.1.

Association Specification

```
class Sales :: public Association {  
    long OID;  
    Participant< Supplier > participant1;  
    Participant< Bicycle > participant2;  
    public:  
        Sales(participant1, participant2);  
        ~Sales();  
        ...  
}
```

```
Category< Sales, Set > salesCategory;
```

Table 3.3: Association specification for OB model

3.2.2 Operations

The OB model, supports a variety of operations. Being based on the C++ language model, the DDL is an extension of the class declarations of the C++ language. Data definition for an application includes the declaration of classes and subsequent declaration of categories as persistent stores for the pre-declared classes [Section 3.2].

Data Manipulation Operations

1. Creation and Deletion

Objects and Category instances are created and initialized through constructors. Constructors are functions that are automatically called to create the class values for objects. Destructors are functions called for object destruction. Categories are persistent and created on the implementation dependent persistent store. Transient objects are made persistent when inserted into a category.

Objects can be created as:

- (a) *automatic objects* - created each time its declaration is encountered, using the constructor and destroyed using the destructor. An example of creating automatic objects is

```
Supplier S1("S1","Smith",10, "London");
```

- (b) *free store objects* - created using the `new` operator and destroyed using the `delete` operator.

```
Supplier* S2 = new Supplier;  
delete S2;
```

Further details on object creation and deletion can be found in [Str91].

Categories are automatic objects, created through their default constructor and destructors. During the creation of a Category, the collection type for organizing the objects within the category is specified. Objects within the Category may be organized as Set, List or Queue.

```
supplierCategory = Category<ClassSupplier, Organize * OType >;  
cdelete supplierCategory;
```

Certain constraints are involved on the deletion of a Category object. All objects contained within a category are deleted with it [Section 3.2.3].

2. Collection operations

Homogeneous objects and values within the OB model can be grouped into collections. The collection types may be Set, List or Queue. Collections are considered finite, although their size need not be defined during class declarations. Members of a collection may be atomic or molecular. Functions that can be performed on collections include

- (a) *create and delete* - can be used to create the collection types and delete the collection types through their constructors and destructors.
- (b) *insert(m)*- can be used to insert object m into a collection. An example would be:

```
supplierCategory.insert(supplier);
```
- (c) *remove(m)* - can be used to delete a member m from a collection. Certain constraints are imposed on the deletion of a member from a collection. These are discussed under Section 3.2.3.
- (d) *remove_all* - can be used to delete all members m from a collection. Certain constraints hold prior to deletion of all members from a collection [Section 3.2.3.
- (e) *is_member(m)* - can be used to find whether a member m belongs to a collection.

3. Set operations

Regardless of the underlying organization schemes, set operations can be performed on categories containing the same type of objects. The set operations that can be performed include union, difference and selection operations.

- (a) *union and difference* - the + and - operators can be overloaded, to represent the union and difference respectively.

```
supplierCategory = subsupplier1Category + subsupplier2Category;
subsupplier2Category = supplierCategory - subsupplier1Category;
```

- (b) *selection(condition)* - is used to find all members that meet the selection condition. The condition is a statement comprising of object member functions separated by conditional operators. The conditional operators being:

```
AND      &&
OR       |
NOT      !=
```

To get all suppliers who stay in the city of "London", the selection operation would be:

```
Set<Supplier>* suppliersInLondon;
...
suppliersInLondon =
    supplierCategory.selection(SCity="London");
```

For multiple selection criteria in the selection condition, the operation would be:

```
Set<Supplier>* suppliersInLondon;
...
suppliersInLondon =
    supplierCategory.selection(< SCity = "London" >
        && < SStatus = 3 >);
```

4. Group operations

Group operations are performed on specific data members of a category. A category is a collection of homogeneous persistent objects. Group operations

apart from count take as their parameter, an expression which returns a numeric value and needs to be applied to every object within a category. The expression may contain, member functions of the object too. Group operations include:

- (a) *Count* - gives the count of the members within a category.
- (b) *Average* - given an expression with at least one data member, contained within it, the expression is applied to all objects contained within the category. The sum of the results obtained is divided by the count of the objects for numeric fields.

```
supplierCategory.Average(2*SStatus);
```

- (c) *Min (Max)* - Gives the minimum (maximum) values for a numeric data member within a Category.

```
supplierCategory.Max(SStatus);
```

5. Navigation Operations

Every category has a currency indicator for traversing through the objects of a category. The functions include,

- (a) *First* - returns the first object within the category.
- (b) *Next* - returns the next object in the category. The navigation can begin with the first member and traverse every next member of the category till the last member has been traversed.
- (c) *Last* - returns the last object in the category. A sample illustration of using the three navigation operations is illustrated below:

```
supplier* s;  
...  
s = supplierCategory.first;  
while (s != supplierCategory.last){  
    ...  
}
```

```
        s = supplierCategory.next;  
    }
```

6. Miscellaneous Operations

Additional operations for indexing on specified data attribute can be defined. Categories can be extended to include user defined data and functions too.

3.2.3 Constraints

The following set of constraints are enforced by the OB model:

- Entity Integrity constraint is enforced through unique object identifiers for every entity. The object identity is not replicated even if the object ceases to exist.
- Every object can be contained within only one category of an object- base. The relationship between a category and contained objects is $1 : N$.
- Objects can be "part-of" only one object. There exists a $1 : N$ functional mapping between the complex object and its components. A complex object may contain several components of the same type.
- Objects can participate in several relationships with other objects, through associations. Cardinality and Participation constraints can be specified for associations between objects.
- Deletion semantics:
 - If a Category is deleted, then all its objects are necessarily deleted. The deletion is propagated to all sub-components of the complex object.
 - A Category participating in an association cannot be deleted. The deletion of the association category precedes the deletion of the participating categories.

- An object cannot be deleted if it is associated with another object.
- Deletion can be manual or automatic. Automatic deletion purges all objects from the Object-base. Manual deletion is two-step process, which marks the objects as deleted but requires an explicit purge for subsequent removal of the objects from the Object-base.

3.2.4 Meta-data Storage

The OB model is complete as per our four-tuple definition of the data model in Section 2.1. The meta-data for the OB model, can be stored within the Object-base using the operations and constructs defined in the preceding sections. A category containing category instances is easily stored within the persistent store. Similarly, class specifications with nested classes declarations can be stored as regular complex objects within a category.

3.3 Implementation of OB constructs

Two methods for incorporating Persistence in OODBS are:

1. Explicit definition of a class as persistent
2. Making a class reachable through a persistent class

The OB model follows the later approach. In OB, Objects are made persistent by making them reachable through a Category. Categories are stored within the Object-base.

3.3.1 Database

It is the responsibility of the DBMS ObjectManager, to maintain an account of the databases and their location. This is part of the database metadata maintained by the DBMS. Each database is treated as a persistent object. It is uniquely identified by its system generated identifier and the user-defined database name which

acts as the object key. Object key is akin to the primary key in relational databases. Performance attributes may be the other members of the database object. Additionally, the database serves as a container for category instance descriptors and the category instances. Each category instance descriptor, maintains meta-data about the categories and information to map categories to their physical location.

3.3.2 Categories

A Category, by definition, is a collection of homogeneous persistent objects. The collection can be organized as - a list, stack, queue or set. Each of these data structures, are implemented as parameterized types for user defined classes. The definition for a Category would be:

```
template <Class OBJECTYPE>
class Category : class Persistence {
protected:
    int                catId;
    String             catName;
    long               catCount;
    Database           catDbId;
    Organize*          catOrganizePtr;
    Cache*             catCachePtr;
    Index*             catIndexPtr;
    Neighbor*          catNeighborPtr;
public:
    Category(String& name, ORGANIZETYPE OT="Set");
    ~Category();
    int insert(OBJECTYPE m);
    int remove(OBJECTYPE m);
    ...           /* Other Collection operations */
    Category<OBJECTYPE>& operator+(Category<OBJECTYPE>& s1,
```

```

        Category<OBJECTYPE>& s2);
Category<OBJECTYPE>& operator-(Category<OBJECTYPE>& s1,
        Category<OBJECTYPE>& s2);
Category<OBJECTYPE>& selection(Condition);
...          /* Other Group and Navigation operations */
ORGANIZETYPE* re-createOrganize(OBJECTYPE, ORGANIZETYPE);
void delOrganize();
void createCache(OBJECTYPE);
void delCache();
void joinNeighbor(long&);
}

```

The Category class is a *class template* requiring a single type parameter to generate a category instance for a specific, user defined class. Each Category Instance is uniquely identified by a system, generated immutable category Identifier `catId` and Category name `catName`, supplied with its constructor. The Category name is the Object key. `catCount` maintains the count of Objects stored. `catDbId` is the database identifier in which the category with contained objects would be stored. `catOrganizePtr` points to the Organization List for the Objects within the Category. The Organization list may be a list, queue, stack or set. Each Organization mode is parameterized to contain the Object type. The Organization structures contain operations for manipulation of each individual node that contains the Object data. The Organization structures and their operations are provided as OB library classes. A data node in the organization structure, contains the Object Identifier, pointer to the Object, version number for the Object and a time-stamp to track the time the object is added to the Category.

`catCachePtr` is the pointer to the Cache for the Category. The Cache is not created automatically for every category and needs to be explicitly created by the user for a category. The performance of the database or access of objects can be improved with the definition of a cache at the expense of space overheads. The

cache may be defined to maintain a log of last-n objects from the persistent storage. `catIndexPtr` points to the Index data for the Category. Indexes need to be explicitly created for each category. The Index define the key attributes, and provide the option for indexing techniques. The access techniques could be inbuilt to provide options for several indexing options like B-Trees, R-Trees, etc. Additionally, **Neighbors** are list of user-defined related, categories who are friends. Neighbors have access to the object data contained within the category. `catNeighborPtr` is a pointer to a list of friendly neighbors. Additionally, the user has the privilege to define a subclass of Category and specialize it to contain additional functions or data members.

3.3.3 Operations on OB

Category(String& name, ORGANIZETYPE OT="Set"); Constructor function for a Category which is responsible for its creation and initialization. It takes the Category name and Organization type as its parameters. The default organization for the objects within the category is a Set.

~Category(); Destructor for Category; deletes the category object and the Category instance memory descriptor from their physical locations; Issues involved are handling of association categories which reference the category targeted for deletion, Neighbors which reference this category;

int insert(), int remove(); Functions insert and delete are collection operations for inserting and deleting instances from Categories. Other collection operations like `remove_all()` etc. have been explicitly listed under section 3.2.2.

Operator +() and Operator -(); the union and difference operations have been implemented by overloading the + and - operators.

Selection(); the selection operation depends on the object storage method within the category and selects those objects satisfying the specified selection condi-

tion. The condition can be decomposed into its constituent clauses separated by the operators AND, OR.

ORGANIZETYPE* re-createOrganize(OBJECTYPE, ORGANIZETYPE);

Creates an Organization structure to hold the collection of persistent objects. The parameters accepted are the object type of the Category and the type of organization structure that needs to be created. The organization structure also holds the identity of the object and additional system information about it. The default **ORGANIZETYPE** is a Set. This function is called by the The function is primarily used when the Organization structure for the Category needs to be changed from one type to another.

void delOrganize(); Deletes the Organization list. This function is used to change the Organization structure of a predefined Category by first deleting it and then recreating the Organization structure.

Cache* createCache(OBJECTYPE, int, int); Creates a cache of specific size, with additional parameters to store the access to the last k-objects, and the log for retrieval of last n-objects.

void delCache(); Deletes the cache for a category, free the space.

void addInst(long instId); Adds an object to the Category; The parameter is the Object Identifier, which is stored in the organization list;

void updateInst(OID); update an Object, contained in the Category.

void delInst(OID); deletes an object of the category; Issues involved could be referential integrity, manual or automatic deletion, two phase deletion for manual deletion.

3.4 Differences with the ROB Model

The *Association* and *Category* constructs were first introduced in the ROB Model. However, the presentation of the ROB model constructs in [Pet93] is from an implementational viewpoint. Other constructs supported by the OB model are *Atomic*, *Molecular*, *Multivalued* and *User defined types*. Under [Pet93] the architecture of a system based on the ROB model is described too.

This thesis defines the OB model from a conceptual viewpoint within the framework for a true data model (defined in Chapter 2). The constructs supported by the OB Model are *Values*, *Objects* *Classes*, *Associations* and *Categories*. The thesis also defines the different operations and constraints supported by the OB model. It discusses the implementation of these functions and uses parameterized type in C++ for proposing the implementation of the *Category* construct.

Chapter 4

Context for Object Base Model

The current literature has not adopted a standard representational data model for Object-oriented databases. The Object-Base (OB) model [Pet93] is a representational model which attempts to fill this gap. In the previous chapter we formalized the OB model to represent a true implementational data model. Object-base model incorporates the important features proposed for present day OO models [ABD⁺90] and overcomes their semantic limitations.

Section 4.1 explains the motivation behind the design of the OB model. Section 4.2 describes the taxonomy for present-day Implementation models and the rationale behind classifying the OB model as a class based object-oriented model. Section 4.3 compares the OB model with a variety of current object-oriented models. Section 4.4 describes the Interoperability of the OB model with the Relational Model and under Section 4.5 we compare the standing of the OB model against the OMG-93 industry standard.

4.1 Motivation for the OB Model

There has been no common agreement on a data model for an OODBMS. The OB model is introduced to provide a new, conceptual approach for Object-Oriented databases.

1. It attempts to improve some of the problems that OO data models of present-day OODBMS suffer from:
 - OO models view objects as being both intension and extensions of data. The OB model distinguishes between the two and clearly declares classes as being the data intension (schema) and objects as data extensions of an objectbase.
 - OO models represent and implement all relationships(except the "is-a" relationship) between entities through membership pointers. This approach has many drawbacks. Chief among those being ability to deal with relationships as independent semantic constructs and maintenance of referential integrity. The OB model overcomes these shortcomings by representing relationships as a separate *Association* construct.
 - The OO model represents composite objects through unidirectional or bidirectional pointers with special operations defined for its access. By embedding the component objects within the composite object, the OB data model provides a clean semantic difference between the "traditional" relationship and the "part-of" relationship for composite object. Components of a composite object are accessed as members using a special path operator (.).
2. The OB model enhances the existing OO data models by incorporating the following:
 - A new construct called *Categories*. Categories are containers for homogeneous persistent objects. Objects are made persistent transparently through their storage into categories. Categories support operations for storage and retrieval of singular and complex objects. Categories support a number of features to improve the retrieval performance of OODBMS.
 - Storage and configuration of Meta-Data is an integral part of OO data models. Meta-data is data too and should be stored and retrieved uni-

formly as regular data with restricted access. In the OB model, meta-data can be configured and stored within Categories as regular data. Each application contains a category containing information about the category instances defined for an application.

3. The OB model improves performance of OODBMS by proposing the following:
 - Improved ad-hoc query facilities to query simple and complex objects, retrieve or store singular or a set of objects from categories.
 - Provide a richer set of collection types. The collection types can be used to organize objects contained within a category or form Abstract Data Types. The OB model supports group operations on collection types.
 - Implementation of a category level cache to facilitate the retrieval of objects. The cache size and capacity can be user defined to keep a log of the *last-k* objects accessed together with their offset locations.
 - Flexibility for defining a set of indexes for different attributes of a class using different indexing methods.
4. The OB models supports interoperability with the Relational model. Tables in the Relational model can be mapped onto Categories in the OB model. The tuples in the tables are objects contained within the categories. The mapping is bi-directional. However, the capabilities of the Relational model need to be enhanced to handle the functional and compositional aspects of the OB model. Functional enhancement would include storage of object methods. Relational tables need to store, complex data without disaggregation. Additionally, OB model provides direct support for SQL3. SQL3 is upward compatible with SQL-92 with the facility to define complex data types and implement routines using SQL3 procedural extensions. This has been discussed in further detail in Section 4.4.
5. The OB model lays the foundation for supporting all mandatory and optional functionalities that should be provided by the OODBMS as specified in the two

manifesto reports [ABD⁺90, Kim90c]. Features pertinent to a DBMS namely Concurrency, Transactions, Recovery are not treated within the scope of the data model discussed in this thesis.

6. By using an extension of the C++ language to implement the OB model, the "impedance mismatch" problem faced in the development of conventional, DBMS applications is eliminated.

4.2 Taxonomy for OO models

The OO data models may be characterized on the basis of the implementation technique for storage of objects as:

1. Classic Implementation Models
2. Object Oriented Implementation Models

Classical Implementation models use the traditional record based approach for storage of objects. Three types of traditional models used are - the relational, hierarchical and network models. Of these the Relational model has been most popular for providing persistence and incorporating the object-oriented concepts. All OO models based on this approach are generically called the Extended Relational Model. A major drawback with Extended-Relational approach is the disintegration of nested-complex objects into several, flat-file relations.

As opposed to the classical models, OO implementation level models manipulate and store objects directly. Present day OO models can be further classified on the basis of their object handling capabilities as falling into either one of the following categories:

- Pure Object-Oriented Models

They are developed or based on pure OO languages like SmallTalk, Common Lisp which treat objects as first class system objects. Both objects and values are treated uniformly as objects.

- **Class Based Models**

Under class based models, the concept of classes is the basis for OO. They support all the characteristics of the object-oriented paradigms, yet differentiate between objects and values. In this model, classes are used as templates for objects. The implementation models developed are based on class based languages like C++;

Our model supports persistence and extends a class based language (C++) for providing object oriented features. It thus qualifies as a Class Based Object-oriented Model.

4.3 Survey of OO Data Models

Under this section we survey the individual data models which form the basis for present day OODBMS. The OODB paradigm inherits properties of OOPL systems and persistent systems. The power of OODB comes from the seamless treatment of persistent data residing in the database and transient data as found in executing programs. Our survey covers the major features of data models of current OO databases. A comparison with our OB model is made. The survey deals more with data modeling issues in the various OODBMS.

4.3.1 ObjectStore

ObjectStore is a popular, commercial OODB system developed by Object Design Inc. It is widely used in the industry for engineering databases, CAD/CAM etc. It is based on the C++ model.

Features

- Supports Object Identity through 32-bit pointers which indicate the actual physical location of the object.
- Persistence is orthogonal to type like ODE.

- Other features include an advanced versioning system, navigational and associative queries like ODE.

Comparison with OB model

- ObjectStore provides persistence to objects in two ways:
 - by explicit creation of application dependent root objects using the *persistent* operator
 - by overloading the *new* operator

Unlike Objectstore, the OB model makes persistence transparent without requiring an explicit declaration of all persistent objects. This is achieved by propagating persistence to all objects inserted into a category.

- Although, the meta-data(schema) in ObjectStore is stored as objects, separate operations are required for its storage and retrieval. The OB model provides a uniform treatment for storage and access of meta-data and regular data.
- Objectstore handles relationships by explicit declaration as a set of $1 : n$ pointers in the participating classes. The OB model too requires explicit declaration for relationships but as a separate construct and not as members of the participating classes. The advantages of providing a separated relationship construct have already been pointed out in Section 3.2.1.
- ObjectStore supports automatic referential integrity of binary relationships through inverse membership declarations. However the onus still lies on the programmer to explicitly declare and maintain the inverse relationships for all participating classes. On the other hand the OB model requires a singular declaration for $n - ary$ relationships. Maintenance of referential integrity is automatically done by the system for all participating classes.
- ObjectStore supports parameterized and non-parameterized collection types but provides only some basic operations for the collection types. Group collection operations like unions and difference of sets are notably missing. This

restricts the data manipulation operations of the associative query language. The OB model supports a variety of collection types. It supports union and difference operations on collection types and a variety of group functions like count, average.

4.3.2 O_2

O_2 was designed and developed by the Altair Research Consortium. The programming language interface is embedded in C and Basic; the resulting language are called O_2C and $BasicO_2$ respectively.

Features

- Distinguishes between types and classes explicitly. Class declarations are divided into types and methods. Types reflect the structural aspect and methods the functional aspect of the object
- Persistence is handled by:
 - explicitly declaring the object as *persistent*
 - making the object reachable from a *persistent* root
- Distinguishes between the ownership semantics and reference semantics
- Multiple inheritance requires explicit rename command to resolve ambiguities
- Facility to define a new type structure on the fly for the values in a query result is an important feature not available in all OODBS

Comparison with OB model

- O_2 is a bit weak in handling relationships. It represents the relationship in a singular direction, with no indication that the two attributes in the same class represent the same relationship. This makes maintenance of relationships very difficult. The OB model supports $n - ary$ relationships explicitly between all participating classes. Maintenance of referential integrity is automatic.

- O_2 supports collections like tuples, lists and sets. Tuples are used to declare types whereas lists and sets are collection types. O_2 supports set union and difference operations only. The OB model supports sets, lists and queue collection types, each of which can be used to declare multivalued types. OB model supports a variety of collection operations for collection types including the union and difference operators.
- O_2 does not support navigation operations. The OB model supports navigation operations for atomic and composite objects.
- Meta-data in O_2 is not configured to be supported like regular objects. In the OB model, meta-data is stored and retrieved uniformly like regular objects.

4.3.3 Ode

Ode (Object Database and Environment) is a database system designed by the AT&T Bell Laboratories in C++. The language used for object creation and manipulation is O++ which is an extension of C++ or CQL++ which is a SQL like language extended with object-oriented features.

Features

- Sharp distinction is made between the persistent objects and transient objects. Although the migration between the two is transparent, explicit declaration for persistent objects is required to store the object in the external store
- Introduced the concept of supporting commands to control the storage of objects into cluster and subcluster.
- Sets are the only constructs supported by the ODE data model. Some advanced set operations including iterations over set members, set union and difference are supported
- Facility to define explicit constraints for the application is supported by ODE. Constraints can be inherited by subclasses

- Triggers and timed trigger specification are supported
- ODE provides restricted but powerful operations on sets. It also provides the ability to perform arbitrary joins, recursive queries on transient and persistent objects through navigational access.

Comparison with OB model

- OB model provides explicit support for traditional relationships between objects. This facility is implemented in ODE through inverse membership
- Meta-data within ODE is not configured to comply with the data model as is the case for the OB model
- The OB model provides more collection constructs for organizing the stored data.

4.3.4 Versant

Versant is a commercial OODBMS developed by Versant Technology Corp. using C++.

Features

- The Logical Object Identifier (LOIDS) consists of the database identifier and an object identifier. The actual physical location is maintained in the Storage Object Identifier called SOIDS. LOIDS have to be mapped to the SOIDS. LOIDS are independent of any physical disk location and able to move objects around independently on the disk

Comparison with OB model

- Versant supports its own object model which includes features of C++ such as multiple inheritance and features of SmallTalk such as dynamic class creation. OB model supports an extended C++ model.

- Versant supports links to represent persistent, unidirectional relationship between two objects. All relationships in Versant are supported through links. Links are pointers to an object. The OB model represents relationships through associations which are objects. Associations support $n - ary$, bidirectional relationships.

4.3.5 Orion

Orion was initiated by the Advance Computer Technology (ACT) program at Microelectronics and Computer Technology in 1985. It is based on the List Object Oriented Paradigm (LOOPS).

Features

- The Object identifiers for Orion are tuples as opposed to simple, atomic types for other OODBS. The object identifiers are assigned as $\langle class_id, instance_id \rangle$ which leads to difficulty in inter-class object migration.
- Supports both typed data as well as non-typed data. Most OODBS support only typed data.
- Objects can be shared by more than one object or exclusive to a singular object.
- Supports multiple inheritance and two policies for resolving name ambiguities, which are *Preferential Inheritance* and *Rename-after-inheritance*

Comparison with OB model

- Relationships in Orion are unidirectional and do not implement reverse references. All references need to be maintained explicitly. The OB model has a semantic and implementation advantage by representing relationships through the association construct.

- Orion maintains relationships to composite objects through references. The OB model embeds the component objects to support the "part-of" relationships.
- Orion supports only queries on a single object at a time. The OB model supports set based queries as well as queries for accessing a single instance at a time.
- Orion implements only B-tree indexes. The OB model provides flexibility and variety of index choices to be constructed on an attribute of a class.

4.3.6 GemStone

Gemstone is a commercial product developed by Servio Inc. The model supported is an extension of SmallTalk-80 for database systems. It was one of the first OODBMS products.

Features

- Supports heterogeneous objects as set elements.
- Joins in the GemStone can be performed only between objects for which references have been pre-defined and stored in the objectbase.
- The query language facility is structured like Smalltalk.

Comparison with OB model

- GemStone Supports only single inheritance. The OB model supports multiple inheritance.
- GemStone does not support a global query facility like the OB model.
- Objects in GemStone are not deleted explicitly but collected by a garbage collector when the object is not referenced. Under the OB model, the deletion of objects, associations and categories is explicit.

- Relationships have to be explicitly maintained in GemStone using references. The OB model overcomes the semantic and implementational constraints of representing relationships through references by using predefined association constructs.

4.3.7 IRIS

The IRIS project was initiated by the HP Labs. It supports the DAPLEX functional model but is developed in C and Pascal. It is an extended functional database system. It supports OSQL which is an object-oriented version of SQL.

Features

- Supports the three main constructs of Objects, Types and Functions. Objects consist of literals and surrogates. Types are defined as a collection of objects capable of participating in a definable set of activities. Types have unique names and are modeled by *functions*
- IRIS supports a type hierarchy with inherited functions, even the data members are represented as functions
- Functions are explicitly declared over types. They can be one of the following:
 1. Stored functions: are functions stored in persistent data structures
 2. Derived functions: are functions which can be derived from other functions
 3. Foreign functions: are programs written and compiled in a different programming language outside IRIS. The object code is stored in IRIS and executed on function invocation

Comparison with OB model

- IRIS Supports $n - ary$, relationships through functions with inverse declarations to support referential integrity. This feature does not support the se-

mantics of representing relationships as independent constructs. This problem does not arise in the OB model.

- In the OB model, the function prototypes are stored as part of the objects. Their implementation details are stored elsewhere within the objectbase. The functions can be applied to each individual instance during associative or navigational access. OB model can be extended to support derived and foreign functions.
- The IRIS model does not support any enforcement for encapsulation. It is regarded as object-oriented because objects can only be manipulated through functions associated with type. OB model supports encapsulation of structure and functions.
- IRIS does not provide any special composite object management facilities. The OB model supports both simple and composite object management facilities.
- IRIS supports functional inheritance from multiple types. The OB model supports data and functional inheritance from multiple types.

4.3.8 POSTGRES

POSTGRES is a sequel to the INGRES relational database system. It is constantly being evolved by the University of California, Berkley since 1986. The data model extends the well founded relational model.

Features

- Class is named as a collection of object instances, with each instance having the same collection of named attributes of a specific type.
- Classes have been classified as *real* if their instances are stored in the database. *Derived* classes have their values computed as and when needed. *Versioned* class instances are stored as differential relative to its parent class.
- Supports a temporal dimension of data

Comparison with OB model

- Composite objects in POSTGRES are stored in separate tables, related by values. The OB model embeds component objects within the composite objects.
- POSTGRES supports only sets as constructs. However, the sets may be homogeneous or heterogeneous sets. The OB model supports a variety of collection constructs for homogeneous objects.
- POSTGRES treats composite objects as attributes of object instances. Under the OB model, composite objects have objects embedded within them.

4.4 Interoperability with the Relational Model

Interoperability between the two data model needs to be bi-directional:

- i. Relational model to OB model
- ii. OB model to Relational model

Relational model to OB model

The mapping from Relational Model to the OB model is direct enough. A Relational schema is defined to be a collection of Relational Intensions. A Relational intension is structured definition for tables. The actual database is a collection of relational extensions (tables) containing a set of related records conforming to their intension definition. Categories in the OB model, are collections of homogeneous, persistent objects. The attribute definitions form the data members of the class defined for the category. The tuples contained within a relational table can be mapped as objects stored within a category. The category on creation can be specifically structured to be a *set* for equivalence to a relational extension.

Figure 4.1: Mapping between the OB model and the Relational model

OB model to Relational model

The mapping from OB model to Relational model is more complex. Relational Intensions are two-dimensional structures but a class definition is a three-dimensional structure [Figure 3.2.1]. The Relational model needs to be enhanced to handle the functional and compositional aspects of the OB model. Relational model stores data as flat, 2-dimensional tuples. Objects contained within Categories are homogeneous, complex objects. Relational model needs to enhance its capability to handle composite, multi-valued types as data values and the ability to process user-defined member functions. With the advent of SQL3 the Relational model will have a database language to handle complex data types, support collection types and user-defined functions.

Relational model thus can be visualized as being a subset of the OB model [See Figure 4.1].

4.4.1 Compatibility with SQL3

The new extension of the SQL database language, SQL3 is being developed by ANSI X3H2 and ISO DBL. SQL3 is being developed for the last three years and is targeted for completion in 1997 [Kul94]. Some of the proposed features include:

- Enriching the type system by incorporating the notion of Abstract Data Types (ADTs). ADT definition corresponds to a set of attributes and application specific behavior.
- Attributes and behavior are encapsulated and their visibility is further controlled by PUBLIC, PRIVATE and PROTECTED.
- User defined unary and binary operators can operate on instances of ADTs
- ADTs can be - *object ADTs* with unique, system assigned identifiers or *value ADTs* whose instances have no unique identifiers associated with them
- ADTs support subtype-supertype relationships
- Collection types such as SET, MULTISSET, and LIST types must be provided as built in parameterized types

The Object management features contained in the draft specification of SQL3 are presented under the section on Object SQL [Gal92]. Object-SQL statements supporting the notions of Abstract Data Types (ADTs) are illustrated for the example of a bicycle store discussed under Section 3.2.1. All examples are given for the sample data defined in the Table 3.1. The first step requires the creation of the database with a given name.

CREATE SCHEMA BICYCLE_STORE

The next step requires the declaration of domains for a schema giving each a name and a type. Domains types may be simple or ADTs, single-valued or declared as a collection type to hold multi-values. The subsequent step requires declarations for the Abstract Data Types. The declaration of ADTs are similar to C++

class definitions and OB class definitions. Under Object-SQL the ADT specification becomes:

```
CREATE ADT    BICYCLE
  WITH OID INVISIBLE
  (
    PRIVATE bno INT(8)
    PRIVATE bmake CHAR(30)
    PRIVATE bmode INT(1)
    PRIVATE wheelist LIST OF classWheels
    PRIVATE framelist LIST OF classFrames
    PRIMARY KEY (B#)
    CHECK    COUNT(framelist) = 1
    CHECK    COUNT(wheelist) = 2
    CONSTRUCTOR FUNCTION BICYCLE (INT BIKENO)
      NEW
    END FUNCTION
    DESTRUCTOR FUNCTION ~BICYCLE
      END FUNCTION
  )
```

The class for Supplier can be defined in the same manner. It can be seen, that the class definitions for the OB Model depicted in Table 3.2.1 can be directly mapped to the ADT definitions for a database under SQL3. To support the association construct in the OB model, inheritance needs to be supported by SQL3. The UNDER statement supports the subtyping of ADT in SQL3.

```
CREATE TYPE SALES
  WITH OID INVISIBLE
  UNDER ASSOCIATION
```

```
(  
    PRIVATE participant1 Participant< Supplier >  
    PRIVATE participant2 Participant< Bicycle >  
    CONSTRUCTOR FUNCTION SALES (Participant< SUPPLIER >  
        participant1, Participant< BICYCLE > participant2)  
    NEW  
    END FUNCTION  
    DESTRUCTOR FUNCTION ~SALES  
    END FUNCTION  
)
```

All instances of an ADT need to be stored in Tables. A Table like a Category, acts as a container for ADTs. Object instances can be created using the **INSERT** statement on Tables. The Table instances are managed by the usual SQL **SELECT**, **UPDATE** and **DELETE** statements.

```
CREATE TABLE BICYCLECATEGORY OF BICYCLE  
CREATE TABLE SALESCATEGORY OF SALES
```

The above SQL statements create tables called BICYCLECATEGORY to hold instances for the ADT BICYCLE. Persistent instances of ADTs can be stored in tables and queried using the familiar SQL constructs. An example for retrieving all information about the wheels of bicycles in the city of "London" would be:

```
SELECT B.BWHEELS  
FROM   B   IN BICYCLECATEGORY  
WHERE  B.BMAKE  = "Peugot"
```

For accessing the component of an object within an object, the path-navigation operator (.) could be used. For example retrieving the information about the sub-component wheel of a bicycle, the query could be structured as:

```
SELECT B.BWHEELS.WSIZE
FROM   B   IN BICYCLECATEGORY.WHEELS
WHERE  B.BMAKE = "Peugot"
```

Data Manipulation Operations like INSERT and DELETE for single or multiple instances are supported. For inserting a single instance of an object into a bicycle category, the query would be:

```
INSERT (B#="B4", BMAKE="BMX", BMODEL=330,
        B.WHEELS.W#="w4", B.WHEELS.SIZE=21...)
INTO   BICYCLECATEGORY
```

Multiple Instances can be inserted by using one of the Collection types. The collection types supported by SQL3 include ARRAY, LIST OF, SET OF and RECORD. An example of the Insert and Delete Operations are shown below:

```
INSERT SET <(B#="B4", BMAKE="BMX", BMODEL=330,
             B.WHEELS.W#="w4", B.WHEELS.SIZE=21...),
            (B#="B5", BMAKE="VWM", BMODEL=651,
             B.WHEELS.W#="w2", B.WHEELS.SIZE=24...)>
INTO   BICYCLECATEGORY

DELETE ALL
FROM   BICYCLECATEGORY
WHERE  B.MAKE   = "Peugot"
```

4.5 Comparison with the OMG-93 Standard

The OB model conforms to the Abstract Object Model proposed by the Object Management Group by distinguishing between the Object Semantic and Object Implementation issues. It differs from the Classical Object-Oriented Model, in the manner in which objects interpret the messages. Under, the classical model, the client requesting the service identifies the object and the service to be performed by

the object. Under the *generalized* OMG object model approach, the client issues a request that identifies the operation and parameters to identify an object. Method selection may be based on any objects identified in the request, as well as the operation.

The OB model partially adopts the OMG object model approach. Categories can be unambiguously identified by *handles* or by values called *object names*. Categories can have multiple handles. Query requests issued by a user is sent to the root category, for broadcasting to the categories likely to perform the operation.

The OB model also conforms to the object implementation issues of object creation, method binding and activation.

Chapter 5

OB-Tool

Over the past decade, the need for computer support for application development not only became apparent but also a top priority for research and development in the software engineering fields. The overall aim of any software tool is to assist the software engineers through the different stages of the development process and improve the quality and productivity of the developed system. In the Requirement Analysis Phase of software development the software engineer is concerned with the identification and modeling of the important elements of the proposed system. Requirements Engineering is concerned with the application domain and its objective is to provide a basis of understanding real world phenomenon and the user's requirements.

EER diagrams are popular, conceptual tools for modeling the functional requirements of a software system. A generation of DBMS users have accepted the ER diagrams and enhanced it with OO constructs to overcome its semantic shortcomings. Majority of the CASE tools support EER diagrams and their translation into a specific RDBMS. However, there is a need to provide similar support for OO DBMS to facilitate their acceptability and popularity. OB-Tool is a prototype tool that provides this support.

To begin, the rules for translating the EER diagram to OB model are presented

in Section 5.1. Section 5.1.1 illustrates the process for this translation with a sample case. An Overview of the OB-Tool is presented in Section 5.2. The actual implementation details of the OB-Tool are discussed subsequently under Section 5.3.

5.1 OO Database design by EER-to-OB Mapping

The process of mapping from EER model to OB model can lead us into transformation from traditional semantic models into OO model. It can be described in the following steps.

Step 1 Simple Entity $\rightarrow$ Class $\rightarrow$ Category

An EER model works with entity types and their corresponding attributes. Simple entities are all entities that are not complex and not a participant of any "is-a" inheritance relationship. Mapping simple entity types to the OB constructs is an ordered, two step process, of Class declarations and Category declaration. Under class declarations, attributes of a particular entity form the data member of the Class for the entity [NNJ93]. The Class declaration is subsequently used for the Category specification.

Step 2 Complex entities $\rightarrow$ Composite Classes

Complex entities consist of component entities. Each component entity is recursively decomposed and each simple component within it is mapped onto its class and category declarations (Step 1). The composite class declaration include the specification of component classes as data members. Associations between complex entities and other entities are represented as other Associations (Step 4).

Step 3 Isa $\rightarrow$ Inheritance

The concept of inheritance associated with "Is-a" relationship in OO schema permits classes to be organized in a hierarchical taxonomy, with the specialized classes inheriting the properties of their ancestors. Four "Is-a" relationships

have been identified in EER diagrams. Their conversion to the OB model is described below:

- For Disjoint-Total "Is-a", a class declaration for all participating entity types is done. The specialized classes are declared as subclasses of the parent class and inherit its members and methods. The second step requires a category declaration for each of the specialized classes. No category declaration is required for the parent class which acts as an abstract class for this relationship.
- For Disjoint-Partial "Is-a", a class declaration and a category declaration for all participating entity types is required. The specialized class are declared as subclasses of the parent class.
- The Overlapping-Total and Overlapping-Partial "Is-a" relationship are not presently supported by the OB model. The OB model requires objects to be contained in one and only one category. Hence presence of an object instance in more than one category as required by the overlapping relationships is not supported.

Step 4 Relationship → class with superclass Association

In EER model, relationships are represented as named associations between entities. In the OB model, a class declaration for each relationship type is required. The relationship class is declared as a subclass of the pre-defined Association class. The attributes of the relationship type in the EER model become the data members of the association class. Details of the relation constraints and rolenames of the participating relationships are stored in the data members inherited from the Association class.

Step 5 Dependent Entity → Component Class

The weak entity can be treated as a component of the complex class in the OB model (Step 2). For each dependent entity type only a class specification is created. The class is contained as a component class.

Step 6 Relationship as Entity $\rightarrow$ Association

Relationship as Entity is used in situations where it need to participate as an entity type with other associations. It is mapped directly to an Association (Step 4). It is a participating member in the Association class for which it has been modeled as an entity.

Step 7 Dependent Relationship as Entity $\rightarrow$ Association $\rightarrow$ Component Class

Relationship as Entity is modeled as an Association. The dependency aspect requires the Association to be included as a component class(Step 5).

Step 8 Multiple Inheritance $\rightarrow$ Multiple Inheritance

Multiple inheritance requires that the specialized class declarations include all the parent class specification as inheritance from multiple classes. The Category declaration is done for the subclass with multiple inheritance.

5.1.1 Sample Case to illustrate EER-to-OB Mapping

The following is an EER model for an application Encyclopedia, which will be mapped to the OB model. Contents of the *Encyclopedia* are maintained online. Encyclopedia volumes can be issued by the library members. The database needs to maintain information about the Encyclopedia, its members as well as about the Issued items. Figure 5.1 shows an EER diagram for the Encyclopedia application which will be mapped to the OB model schema following the rules of the previous section.

The database maintains data about the entity types *Encyclopedia* and *Member*. *Encyclopedia* is a complex entity consisting of the entities *Index* and *Contents*. The entity *Contents* is complex too consisting of the simple entity *Volume*, *Chapter* and *Section*. Members of the library are uniquely identified by their membership Number [memberID]. Address[address], and Membership expiry date[expDate] are the other attributes for the *Member* entity. Members can be specialized as being Individual Persons or other Institutions. The Total-disjoint "is-a" relationship represents this

Figure 5.1: EER diagram for the Encyclopedia application

specialization of the *Member* entity into *Individual* and *Institution* entities. The relationship *Issue* records the *Encyclopedia* borrowed by the various members.

The conversion from the EER model of [Figure 5.1] to the OB model is depicted through the following steps:

Step 1 : No simple entities are identified for the EER diagrams of Figure 5.1. Proceed to the next step.

Step 2 : *Encyclopedia* is the complex entity identified. Recursively decompose the components of the Complex entity. *Encyclopedia* consists of simple entity *Index* and the complex entity *Content*. A class declaration and category declaration for the simple entity is done first.

```

class Index {
    ...
}
category< Index, Set > categoryIndex
  
```

Content is the other complex entity contained within *Encyclopedia*. Recursively applying the algorithm to *Content* creates a class and category definition for its simple components *Volume*, *Chapter* and *Section* too.

```
class Volume{
    ...
}
class Chapter{
    ...
}
class Section{
    ...
}
category< Volume, Set > categoryVolume
category< Chapter, Set > categoryChapter
category< Section, Set > categorySection
```

Further decomposition of the complex entity *Content* identifies the relationships *V_C* and *C_S*. *V_C* maps the volumes to chapters and *C_S* maps each chapter to its sections. An Association is declared for each of the relationships.

```
class V_C : class Association {
    Volume* participant1;
    Chapter* participant2;
}
class C_S : class Association {
    Chapter* participant1;
    Section* participant2;
}
```

The absence of further constructs for decomposition, returns the algorithm back to the *Content* entity. Its class declarations requires inclusion of the simple entities and associations contained within it. Subsequently, a class and category is declared for it, as -

```
class Content{
    Volume aVolume;
    Set< Chapter > chapters;
    Set< Section > sections;
    Set< V_C > volume-chapter;
    Set< C_S > chapter-section;
    ...
}
category< Content, Set > categoryContent
```

Since all the component entities for Contents have been declared, the algorithm returns back to complete the declarations for the association contained within *Encyclopedia* and the principal declaration for *Encyclopedia* itself.

```
class I_C : class Association {
    Index* participant1;
    Content* participant2;
}
class Encyclopedia{
    Content aContent;
    Index aIndex;
    Set< I_C > Index-Content;
    ...
}
category< Encyclopedia, Set > categoryEncyclopedia
```


Step 3 : Look for entities related by the "is-a" hierarchy. Entities *Member*, *Person* and *Institution* are identified. *Member* being the Parent class, is declared first. The "Is-a" inheritance is identified as being "total-disjoint", which means that the *Member* entity acts as the Abstract class. No category will need to be created for it. Declarations are then done for the specialized classes, *Person* and *Institution*. Both specialized class declarations exhibit inheritance from the parent. Categories for the three classes are declared as:

```
class Member {  
    ...  
}  
class Person : class Member{  
    ...  
}  
class Institution : class Member{  
    ...  
}  
category< Person, Set > categoryPerson  
category< Institution, Set > categoryInstitution
```

Step 4 : Look for Relationships between the main entities. *Issue* is a relationship between entities *Encyclopedia* and *Member*. Declare an Association construct with the two participating entities. Any attributes of Association become the attributes *Issue* too.

```
class Issue : class Association {  
    Encyclopedia* participant1;  
    Member* participant2;  
}
```

5.2 Overview of the OB-Tool

The OB-Tool is a prototype tool for supporting database application development. It supports the drawing of Enhanced Entity-Relationship diagrams for an application prior to beginning its development. The details for the EER diagram constructs supported by the OB-Tool are presented in Chapter 2. The OB-Tool takes EER diagrams as its input and translates them into one of the following outputs:

- Schema for the OB model
- Text Markup representation for EER diagrams

5.2.1 OB-Tool Operation

The OB-Tool application is evoked with a main application window for its visual data definition interface. The main application window comprises of:

- i) **Menubar** for grouping and evoking the application functions.
- ii) **Workarea** for supporting the actual creating/editing functions.
- iii) **Messagebar** for depicting the current mode of operation and functionality.

The Menubar groups the main functions under a set of standard pulldown menu commands labeled - "File", "Edit", "Dictionary" and "Help". The "File" menu contains the file related operations to "Clear", "Save", "Load" and "Exit" a diagram file. The "Clear" options are provided for clearing the canvas. This menu, is helpful for clearing the canvas prior to loading a new file. OB-Tool allows two EER diagrams to be merged, if they are loaded simultaneously onto the canvas. This can be done by not clearing the canvas prior to loading the new diagram. Of course, the merged diagram can then be saved if required onto a new file.

The Workarea is further composed of:

- i) Palette of icon push-buttons

Figure 5.2: OB-Tool main window application

- ii) Drawing Canvas
- iii) Diagram Title bar
- iv) Scrollbars

The **Palette** is positioned on the left of the canvas workarea in the main application window. It consists of a collection of iconic pushbuttons. The pushbuttons are separated into two parts - *Creating* and *Editing* to represent the two modes of operation for the application. Additionally, the leftmost button of the mouse is armed for the creating functionalities and the rightmost for the editing functions. The application supports two modes of operation:

1. **Draw Mode** - for drawing/creating new EER constructs for a diagram
2. **Edit Mode** - for editing the previously created symbols or their properties

The **Canvas** is used for the visual definition of the EER diagrams. The EER Diagram Files can be created from scratch or previously created files can be loaded for further editing onto the Canvas. The tool supports two file formats for input:

1. *OB-Tool local format* - Local format for loading EER diagrams. This is a quick loading format as the files are stored in the internal-EER representation for the OB-Tool.
2. *EER-Text Markup format* - A new HTML like format for EER diagrams. This is an ASCII, Textual representation of the EER-diagrams (See Sections 5.3.3 and Appendix B). It takes longer to load as an additional step for converting the Textual representation to the Internal EER-representation is required.

The EER diagram files can in turn be saved under three formats of:

1. *OB-Tool local format* - Local format for OB-Tools.
2. *EER-Text Markup format* - The new, Text Markup Language format for the EER diagrams.
3. *OB Schema format* - The conversion of the EER diagrams into their equivalent Object-Base Model constructs.

The Workarea also has a **namebar** to display the diagram name title. The default is an "Untitled" document that is subsequently changed to the user-specified diagram filename. The Workarea also supports two **scrollbars** for horizontal and vertical panning of the EER diagram file.

The "Edit" menu offers basic "Create", "Delete" and "Move" facilities for the symbols. "Create" is the additional alphanumeric facility provided for object creation as opposed to using the palette for diagram creation.

The "Dictionary" facilities offered are to view existing domains for a diagram and define new molecular domains.

The visual data definition interface of the OB-Tool comprise of the *graphical* and *alphanumeric* interface. The Interface has been designed in conformance to the standards laid down in the *OSF-Motif Style Guide*.

In the graphical mode, the designer builds the EER diagrams by picking the graphical symbols from a palette of push-buttons. The symbols can then be drawn on the canvas of the main window by positioning the cursor at the desired location on canvas, and clicking the leftmost mouse button. The symbols supported are the EER symbols described in Chapter 2. The Links are automatically drawn by selecting the participating entities. For "Is-a" hierarchy, the superclass and specialized classes are selected. The "is-a" hierarchy too is automatically drawn to connect the participating entities. Alternatively, a menu interface too has been provided for creating and editing the diagrams.

The alphanumeric interface is provided for defining the properties for each EER diagram construct drawn on the canvas. The properties are definable by clicking the right mouse button on any drawn construct in an entity. A display window is provided for the construct, wherein the name, attributes, properties and constraints for the construct can be defined or modified.

5.3 Implementation of the OB-Tool

The functional requirements for the OB-Tool were identified as:

1. Provision to support the EER model
2. Generate the OB model construct for the EER diagrams
3. Bi-directional Translation of EER diagrams into an EER Text Markup Language

Each of the functionality was separated into three modules of the OB-Tool:

- EER Diagram Editor
- OB Schema Generator
- EER-Text Markup Translator

Figure 5.3: Block diagram for module interdependency of the OB-Tool

However, the three modules were interdependent (Figure 5.3). The EER Diagram Editor was required for generating the EER diagrams as input which could be translated to any of the following outputs:

- **OB-Tool Local Format** - supports the saving/ loading of OB-Tool files in a specific, pre-defined format local to the module.
- **OB Schema Format** - supports the saving of an schema for the OB model generated from an EER diagram loaded into the OB-Tool.
- **EER-Text MarkUp Format** - supports the saving of an EER diagram from the OB-Tool Module to an EER-Markup Text Format in a file. Also provides support to re-load EER-Markup Text Format and generate its corresponding EER diagram.

The system was designed to be "event-driven". The event handling for the drawing canvas is complex and has been configured to react to runtime user control for the drawn objects. The program structure under the event-driven programming paradigm is:

1. Create the widget;
2. Register the event handlers for these widgets;
3. Go into the main event loop;

4. Activate the appropriate event handler when a specific event occurs on a specific widget. Return to the main event loop when done.

The functionalities were divided into ten implementation modules. The listing of the implementation modules and their functionalities is presented in Appendix B. Figure 5.4 shows the implementation structure of the OB-Tool and the inter-relationship between its various modules.

The *file module* is responsible for the external I/O interface of the OB-Tool. The file module, fetches the user-specified EER format file before evoking the correct format module (*toLocal* or *translateEERText*) for subsequent conversions to its Internal EER Representations. The module *check* is evoked to validate the Internal EER representations for correct EER schema constructs and constraints. This module is specifically important for validating the EER-Text MarkUp Schema Representation. These representations do not require the files to have been previously created through the OB-Tool. The routines under the *create* module are evoked on the well-formed Internal EER Representations for creating the graphical representations of the Schema on the Drawing Canvas in the main application window. The graphical EER Schema may be further edited or modified before being saved into either or all of the three output formats supported by the OB-Tool. The *file* module provides the output interface and evokes the correct conversions routines under the *toLocal*, *translateEERText* or *generateOB* modules.

The development has been done mainly using XLib, XToolkit and Motif development libraries under C and Solaris on the Sun Workstations. The system requirements to run OB-Tool are:

- X Server supporting any Window management facilities
- Mouse or similar pointing device
- Screen supporting Graphics

Figure 5.4: Structure of OB-Tool

5.3.1 EER Diagram Editor

EER models as opposed to any other preliminary database design model, were supported by the OB-Tool for their applicability in the following areas:

- In the Functional requirement phase of software development, EER diagram tools capture the information gathered by the system designers and help in the analysis of the gathered information.

Information gathered and represented in the form of EER diagrams also help the system users (or acceptors) in understanding the modeled system because of their simplistic and visual approach.

- In database application design, it is used for:
 - Capturing the user requirements.
 - Mapping the EER diagram constructs directly to the database schema.
- As Open UPPER CASE Tools

Today, there exist a wide variety of representational constructs for the EER model. However, there has been no common standard or agreement reached for their notational representation. The OB-Tool supports the EER constructs of Chapter 2. for the following reasons:

1. introduction of additional constructs for expressing "relationships as entities".
2. explicit distinction between the four types of "is-a" inheritance relationships and provision of an easily comprehensible and simple symbol for each of them.

The symbols are extensions of the inheritance symbol proposed by Rumbaugh in the OMT Design Technique [BPR88].

For representing the EER notations presented in Chapter 2, two types of entities could be primarily identified : *symbols* and *connectors*. Figure 5.5 shows the EER diagrams to help us define the implementation constructs for the EER diagram editor. Every notation is a Graphic Item. A graphic item can be a simple shape,

Figure 5.5: EER diagram for the EER Diagram Editor

Figure 5.6: EER diagram for the OB Schema Generator

Figure 5.7: EER diagram for the EER-text Markup Translator

like a rectangle, diamond, or circle. A complex shape is graphic item that may be a collection of shapes, to represent a single construct. A connector is a graphic item which may be adorned with simple shapes or a collection of simple shapes and connectors. Lines and the "is-a" hierarchy act as connectors. Attributes could be defined for most shapes while constraints and rolenames were supported for links.

5.3.2 OB Schema Generator

The OB-Tool outputs the EER diagram loaded into its environment to their corresponding OB constructs under the C++ language. The process for translating the mapping from the EER model to the schema for OB model is presented under Section 5.1. For this module a mapping needed to be maintained between the EER constructs and OB model constructs. Two major entities were identified as being *Graphic Item* and *OB-Construct*. The relationship *Maps_To* keeps an account of the specific instances of OB constructs mapped to a graphic item.

5.3.3 EER-Text Markup Language translator

For translating the EER diagram to their equivalent, Text Markup Language, the mapping between the constructs for the EER diagrams and those defined for the EER Text Markup Language needed to be maintained. Two principal constructs of the EER Language - entity and attribute, are described below in their equivalent Text Markup Representation.

```
Entities :      < ENTITY > ... < /ENTITY >
Attributes :   < ATTRIBUTE >
                < NAME = "..." > < TYPE = "..." > < LENGTH = ... >
                < KEY = " * " > < COMPOSITION TYPE = "..." >
                < VALUE TYPE = ... >
                < /ATTRIBUTE >
```

The full EER Diagram Text Markup Language Constructs are given in Appendix A. The advantages of generating an EER output in the equivalent Text Markup representation are as follows:

- Independence of OB-Tool

The Input for the diagram can be generated through any textual editor, without requiring the OB-Tool to be specifically used to generate the diagrams. The advantage is the ability to create a text file on any terminal, eliminating the need for expensive graphics compatible equipment to generate the input.

- Portability

The file can be created under a different platform, then converted to a unix compatible using a conversion facility. This predefined file can then be loaded into the OB-Tool.

- Future applications

In future, the different CASE tools can develop independent, stand alone mod-

ules for translating the OB-Tool Text Schemas into their mode of representation. This will promote heterogeneity, Case Tool compatibility and format independence.

- Aid to Upper Case tool

Upper Case tools are tools that have independent modules for assisting in each phase of software development. The OB-Tool can be the first step towards the phase of designing system requirements. Another tool that supports the next phase of development can be linked to the OB-Tool by developing an interface that translates the textual schema to its internal representation

- Distributed Development

The data file can be transferred, to the destination, edited by the designers and transferred back. Alternatively, the file can be loaded directly into the OB-Tool, re-edited and then stored in a similar format. The file can then be copied onto the original file and the remote location or a separate version made. The advantage of this approach is that the development need not be interactive, client-server based. It can be distributed and independent with each independent machine running its own, editing tool and transferring. The problem with this approach is the need for a mediator to decide the main version to be maintained, security of the document.

Chapter 6

Conclusion

Our research covers the following areas of Database research:

- Object-Oriented data model
- Object-Oriented database modeling
- Interoperability between the Relational data and OO data models
- Implementation of a prototype EER schema translation tool

This research is primarily directed towards extending the framework for a new, data model for Object-Oriented database [Pet93]. The Object-based (OB) model is a class based conceptual data model for OO DBMS. It extends the C++ language model to meet the requirements for databases. It is proposed with the objective of standardizing a data model for OO DBMS and meeting the limitations of existing OO data models. The salient features of OB model are:

- it introduces two new data model constructs of *Categories* and *Associations*.
- it establishes a clean distinction between data intensions as class declarations and defining data extension as collections of objects stored in the *Category*.
- it overcomes the semantics limitations imposed by OO data models for explicit modeling of relationships of any degree. Bi-directional relationships need not be maintained through references and inverse memberships but through the *Association* constructs.

- it represents the meta-data in the object-bases by the same constructs and operations defined for the data model.
- it makes persistence of objects transparent to the user by not requiring an explicit declaration of all classes to be made persistent. Objects are made persistent when inserted into a *category*.
- it provides both navigational and associative access on object collections.
- it facilitates object retrieval by introducing a cache for Categories.

Inter-operability from the Relational model to the OB model is supported. However, for the inter-operability to be bi-directional, major enhancements to the existing relational data models are required. Some enhancements include support for Abstract Data Types and Procedures by RDBMS.

EER diagrams are popular, conceptual tools for requirement specification of information systems. Their simple concise form is easily comprehended by system acceptors. Information system developers find it useful in identifying the system entities and their interrelationships. We extend the EER diagram by proposing the dual concept of *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 only one symbol. Additionally, we support a more expressive and distinct notation for modeling the four types of relationships of the "is-a" hierarchy.

Absence of support for designing object-oriented databases is a practical limitation for the success of an OO DBMS. To support our OB data model, we defined a translation process from the established EER model to the OB model. The translation process was implemented through a prototype, tool which supports our notation of EER constructs for creating and editing EER diagrams. Additionally, the tool supports storage and retrieval of EER diagram into an HTML like textual format, called EER-Text Markup Language. The reasons for supporting this format were:

- CASE tool independence

The EER diagram defined using the EER-Text Markup Language, can be created through any text editor and do not require a specific software or hardware to create the EER diagrams.

- Platform Portability

The EER diagram file created in accordance to the EER-Text Markup Language, under a different platforms can be converted UNIX compatible files through file conversion utilities. These can be viewed under Tools supporting the EER-Text Markup Language Interface.

- Format for data exchange between CASE Tools supporting the interface for generating EER diagrams through a standard EER-Text Markup Language. It can also be used as a format for data exchange between modules of UPPER CASE tools.

However, the EER diagrams generated using the EER-Text Markup Language are lengthy and not easily assimilated.

6.1 Future Work

While proposing the OB data model, there were several aspects that could not be covered. They are presented here as areas of future work in OO data modeling:

1. Query Optimization

Limited query optimization for the OB model was achieved by proposing a cache. Further, research for the optimization of queries within the different levels of the OB model needs to be developed. The different levels can be within a singular category, or between two or more related categories.

2. Dynamic Class Definition

OO DBMS based on the C++ model, require all classes to be known at com-

pile time. Dynamic class creation is required for modification of the schema, without statically recompiling the schema.

3. View Mechanisms

The mechanism for defining views on the OB Model was not covered. It is a helpful though not important requirement.

4. Access Control

Authorization or access level can be defined for different levels of the OB model:

- at the objectbase level
- at the category level
- at the object level

This feature needs to be further studied to promote the success of OB model for business applications.

5. Versioning

There is no consensus on any type of version scheme, as yet. Version support for the OB model could be provided at the the object level, composite object level. Versions of class and schema too could be maintained. These issues need to be further studied and a scheme for versioning the OB model could be proposed.

6. Distributed Objectbase Systems

Further research is necessary to determine the applicability of an OODBMS supporting the OO data model in a distributed environment.

7. Development of a visualization tool for the Internet which uses the EER-Text Markup Language to display the EER diagrams interactively

This thesis makes an attempt to impart robust data modeling capabilities to OODBMS through the Object-Base Model. By overcoming several limitations of

existing OO data models, this work attempts to bridge the gap that is highlighted currently between the well-established benefits of OO technology and traditional databases. It also introduces the OB-Tool to facilitate design of databases by leveraging the modeling capability and simplicity of EER diagrams. The OB-Tool is an user friendly prototype that translates an EER diagram to its Object-base representation and EER-Text Markup representation. In addition, the prototype tool and the robust framework proposed here also provides ample avenues for future research that will further enhance the adoption and penetration of OODBMS in the real world.

Appendix A

EER Text MarkUp Language Constructs

EER Constructs	Corresponding EER-Text MarkUp Language Constructs
Entities	<code>< ENTITY > ... < /ENTITY ></code>
Attributes	<code>< ATTRIBUTE ></code> <code>< NAME = "..." >< TYPE = "..." >< LENGTH = ... ></code> <code>< KEY = "*" >< COMPOSITION TYPE = "..." ></code> <code>< VALUE TYPE = "..." ></code> <code>< /ATTRIBUTE ></code>
Relation	<code>< RELATION ></code> <code>< NAME = "..." >< DEGREE = ... ></code> <code>< ATTRIBUTE ></code> <code>< NAME = "..." >< TYPE = "..." >< LENGTH = ... ></code> ... <code>< /ATTRIBUTE ></code> <code>< PARTICIPANTS ></code> <code>(< NAME = "..." >< CONSTRAINT = "(...,...)" >)</code> <code>(< NAME = "..." >< CONSTRAINT = "(...,...)" >)</code> ... <code>< /PARTICIPANTS ></code> <code>< /RELATION ></code>

EER Constructs	Corresponding EER-Text MarkUp Language Constructs
Dependent Entity	<pre> < DEPEND - ENTITY > < NAME = "..." > < DEPEND_UPON = "..." > < ATTRIBUTE > ... < /ATTRIBUTE > < /DEPEND - ENTITY > </pre>
Dependent Relation	<pre> < DEPEND - RELATION > < NAME = "..." > < DEPEND_UPON = "..." > < ATTRIBUTE > ... < /ATTRIBUTE > < /DEPEND - RELATION > </pre>
Link	<pre> < LINK > < ROLENAME = "..." > < /LINK > </pre>
Is-a Relationship	<pre> < IS - A > < TYPE = "..." > < PARENT = "..." > < SUBCLASS > < NAME = "..." > < NAME = "..." > ... < /SUBCLASS > < /IS - A > </pre>
Multiple Inheritance	<pre> < MULTI - INHERIT > < PARENT = "..." > < PARENT = "..." > < SUBCLASS = "..." > < /MULTI - INHERIT > </pre>

Appendix B

Function prototypes for the OB-Tool

The development of the OB-Tool was done mainly using C, Motif, XLib and Xt. The reason for using the C language as opposed to the C++ language was the absence of libraries supporting the C++ interface for Motif during the implementation of the OB-Tool. The program structure for the OB-Tool is shown in Figure 5.4 and consists of the following modules :

- main.c** The main module reads the command line, the default resource and the application resource files to initialize the OB-Tool environment. It sets up the main window and calls the various modules of the application.
- menu.c** The menu module is responsible for creating the menubar with the menu options. The appropriate functions are linked through callbacks to their respective menu options.
- diagram.c** This module is responsible for structuring the workarea of the OB-Tool. The workarea consists of the drawing-canvas, the diagram name and the palette of icons for creating and editing the diagrams. The file also defines all the callbacks and events related to the workarea.

<code>file.c</code>	The file module consists of functions related to the diagram file. Functions for providing the I/O interface for the tool, providing a choice of translations options are defined in here.
<code>create.c</code>	This module contains the functions for creating the EER constructs and defining their properties.
<code>edit.c</code>	The edit module contains the functions for editing the EER constructs and their properties.
<code>check.c</code>	This is the error handler module of the tool. It checks for the integrity of the EER diagram at the time of creation and again prior to saving it.
<code>toLocal.c</code>	This module is responsible for saving the graphical EER diagram to a predefined OB-Tool format. It also consists of functions that convert the predefined OB-Tool format to their graphical EER representation.
<code>translateEERText.c</code>	This module is responsible for the bidirectional translation of the EER diagram to the EER Text Markup language. The file is obviously an ASCII representation to promote both machine and platform independence.
<code>generateOB.c</code>	This module consists of functions for translating the graphical EER diagram to its equivalent OB constructs.
<code>Makefile</code>	This file is the input to the <i>make</i> UNIX program; the execution of <i>make</i> generates the executable binary code. As a byproduct, the file also specifies how the various modules are related.
<code>motif_env</code>	This file is an interactive <i>c-shell</i> script for setting up the Motif environment for compilation of motif programs under a different window manager, other than the Motif window manager (<i>mwm</i>). The file needs to be executed prior to the Makefile.

The function prototypes for each module are discussed below :

1. **main.c**

static void InitMainWindow(); Initialize and position the main application window on the screen after reading the relevant resource files.

void InitMessageWindow(); Initialize the message bar in the main window for displaying the selected option.

void InitDefaultResources(); Look for the specific application resource file before loading the default resource files for the Tool.

void LoadDefaultTypes(); Initialize the default data type structures and the load the OB-Tool default data types.

void InitMappingEnvironment(); Initialize and load the default data structures for the internal representation of the EER schema. This functions also calls the initialization routines for the OB schema generator and the EER Text Translator.

2. **menu.c**

static void InitMenuBar(); Create a menubar for main application window. The menubar displays the various menu names and initializes the related pulldown menus.

Widget InitFileMenu(); Create a pulldown menu for the File option displayed on the menubar. The File Menu displays the list of file options as pushbuttons. For each pushbutton-option, the corresponding callback function is registered.

Widget InitEditMenu(); Performs the same function as **InitFileMenu()** but for the Edit options.

Widget InitHelpMenu(); Performs the same function as **InitFileMenu()** but for the Help options.

Widget InitDictionaryMenu(); Performs the same function as **InitFileMenu()** but for the Dictionary options.

static void InitCreateMenu(); Initializes the Create submenu under the **InitEditMenu()**. The create submenu provides menu facilities for drawing the EER diagrams as opposed to the icon-buttons displayed on the drawing palette.

static void FileMenuCB(Widget W, char* client_data,
XtPointer call_data);

This is the callback function for the FileMenu options. It compares the option chosen under the File Menu and calls the corresponding routine for execution.

static void EditMenuCB(Widget W, char* client_data,
XtPointer call_data);

The functionality of this callback is similar to the FileMenu callback but for each of the Edit options.

static void HelpMenuCB(Widget W, char* client_data,
XtPointer call_data);

The functionality of this callback is similar to the FileMenu callback but for the Help options.

static void DictionaryMenuCB(Widget W, char* client_data,
XtPointer call_data);

The functionality of this callback is similar to the FileMenu callback but for the Dictionary option.

3. **diagram.c**

void InitWorkArea(); Initialize the working Area on to the Main Application Window. The Working Area consists of the Drawing Area that acts as the canvas for the EER diagrams, a palette of drawing icons and the diagram name bar.

Widget InitPalette(Widget parent); Creates a palette of icon pushbuttons for creating and editing the EER diagrams displayed on the canvas. Each button gets armed after selection. The palette is basically divided into two modes : the DRAW mode and EDIT mode. The icons representing the DRAW mode are displayed under the Drawing section of the palette. The DRAW MODE is an indicator for creating new EER constructs. The Editing section of the palette displays the icons for editing the previously created EER constructs. Clicking on any of the buttons under the Editing section, puts the Tool in the EDIT Mode.

void PaletteButtonCB(Widget W, char* client_data,
XtPointer call_data);
Callback function for the palette to determine the selected mode for the diagram, icon selected and then call the related function.

Widget InitDrawWin(); Creates a drawing area within the working area of the main window. The Drawing Area consists of a canvas to draw the EER diagrams, and scroll bars for horizontal and vertical panning of the canvas. The canvas is initially labeled with the diagram title which is "Untitled" and subsequently converted to the user defined filename during the file save and load operations.

void ClearDrawWin(); Clears the Drawing Area and initializes the internal structure to their defaults.

void InitGC(); Creates a default graphics context for drawing the EER diagrams. The default graphics context includes setting the drawing styles and mode for the graphic diagrams.

static void DrawEvt(Widget W, XtPointer client_data,
XEvent* evt_ptr);

This function is activated for every click of the mouse button on canvas. It determines the kind of button clicked, the kind of click, the Drawing mode selected to relate the corresponding action to be performed. For

example, the left mouse button clicked in the Draw mode calls the create routine for the the selected symbol to be placed on the canvas.

Widget LoadLabel(Widget Parent, char* LabelTitle); This is an utility routine to create a label widget, a label string with the LabelTitle and return the labeled widget to the calling function.

Widget CreateButtonRow(Widget Parent); Creates pushbuttons for the parent widget which is the palette, labels each button with their iconsymbols and register callbacks for each of them.

void UpdateDiagramName(char* filename); Change the diagram label to the new FileName. The function is invoked at Initialization, and for all file File Load and File Save operations.

4. file.c

static int InitFileSaveMenu(Widget Parent); Function to create a pop-up window for displaying the file formats and accepting the filename to save the EER diagram.

static int InitFileLoadMenu(Widget Parent); Function to create a pop-up window for displaying the file formats and accepting the filename to load the EER diagram from.

void DisplayFormat(Widget Parent, int operation); Depending upon the File Load/ File Save operation, the options for the File formats are shown as Option buttons. The File Save operation display the three options for OB-Tool format, EERText Format, and OB Schema format. The File Load operation only displays the first two options.

**static void FileIOButtonCB(Widget W, char* client_data,
XtPointer call_data);**

Call back for the operation buttons (OK,Cancel,Help) displayed on the pop-up window for the File Load/Save Menus.

**static void FileFormatCB(Widget W, char* client_data,
XtPointer call_data);**

Function for determining the Format selected for Loading/Saving and evoking the corresponding function.

5. create.c

void DrawRectangle(Point aPoint); Function to Draw a Rectangle on the canvas at location aPoint. The location position is relative to the canvas area and is given in pixels. Point is a data structure defined to represent the x-coordinate and y-coordinate position on canvas.

void DrawDiamond(Point aPoint); Function to draw a Diamond on the canvas at location aPoint.

void DrawArc(Point xPoint, yPoint); Function to draw an Arc on the canvas from xPoint to yPoint. The validation for determining the two points belong to two relevant entities is done by calling the appropriate functions from file check.c.

void DrawDblRectangle(Point aPoint); Function to Draw a Double-edged Rectangle on the canvas at location aPoint.

void DrawDblDiamond(Point aPoint); Function to Draw a Double-edged Diamond on the canvas at location aPoint.

void DrawMultiInherit(Point aPoint); Function to Draw the complex Multiple Inheritance symbol on the canvas at location aPoint.

void DrawRectDiamond(Point aPoint); Function to Draw the complex shape of a Diamond within a Rectangle symbol on the canvas at location aPoint.

void DrawTriangle(Point aPoint); Function to draw a triangle at a given location point on the canvas.

void UpdateIS-A-Class(Point aPosition); This function determines the entity selected to be a participant of "IS-A" relationship and updates

the internal structures for each selected participant. The first entity selected is the Parent, with the subsequent entities being the subclasses in the "IS-A" hierarchy.

void ComputeIS-ASpecial(IS-Aelement* aIs-Aelement); This function computes the location for drawing the "IS-A" hierarchy based on the location of the entities selected for participation.

void IS-AButtonAction(int whichButton, Point aPoint); Determines the mousebutton clicked and position of click to determine the kind of action to perform for drawing/ selecting the "Is-A" entities. The leftmost button determines the selection while the rightmost completes and draws the relationship between the participants.

6. edit.c

int GetEntity(Point aPoint); The function scans the list of entities to determine the enclosing entity for that selected point and return its identification.

int FetchEntityType(Point aPoint); Determines the Entity Type containing the selected point.

int CallAttributeMenu(Point aPoint); This Function determines the type of Attribute Menu to call depending upon the selected entity.

void GetAttributes(int entityId); This Function displays a pop-up window to display the existing attributes for a given entity. It is also used to add or update attributes and their properties.

**static void AttribCB(Widget W, XtPointer client_data,
XtPointer call_data);**
Callback for determining the operation selected from the attributes menu in the preceding function.

void WriteEntityName(String name, int entityId); Positions and displays the name for a selected entity in the EER diagram.

void DrawEndpoints(char* type); Draw Endpoints to highlight all the graphic item types selected for editing.

int GetArc(Point aPoint); This function scans the range of arcs in the diagram to returns the arc identifier for the arc selected for editing.

void WriteArcName(String name, int arcId); Display the Arc name on the midposition of the arc selected on the EER diagram.

**void EntityPropertyCB(Widget W, XtPointer client_data,
XtPointer call_data);**

The callback function for updating the attributes for an entity into the attribute list and displaying the entity name on the diagram.

void GetArcAttributes(int arcId); Function display a pop-up window to display the existing properties for a given arc. It also facilitates the modification of these properties.

**void ArcPropertyCB(Widget W, XtPointer client_data,
XtPointer call_data);**

The callback function for updating the properties for an arc into the arc list and calling the function to display the rolename on the diagram.

**Widget CreateScrolledList(Widget parent, char* name, int listType,
XtCallbackProc callback, XtPointer client_data);**

Creates and initializes a scrolled list for displaying the attributes for an entity.

void AppendToList(Widget list, char* item); Function for adding an item to the end of the list of items(attributes) displayed in the list.

**void listCB(Widget widget, XtPointer client_data,
XtPointer call_data);**

The callback function for displaying the item selected from the list in the textbox of the attribute menu.

char* GetAttributeList(int entityId); Get the list of attributes for the selected entity.

7. toLocal.c

void Writeheader(File* fp); Writes the header for the OB-Tool format.

The header consists of the filename, the format and the count for each type of construct in the EER diagram.

void WriteLocalConstructs(File *fp); Responsible for calling the various functions for writing the Internal EER Representations to the File.

void WriteDomains(File* fp); This function writes the domains of the EER diagram to a specific file.

void WriteEntities(File* fp); This function writes the entities together with their properties to the file. The entities include the graphical properties and the alphanumeric attribute descriptions.

void WriteArcs(File* fp); This function writes all the Arcs in the diagram to the file. The properties are included with their respective arcs.

void WriteIs-A(File* fp); This function writes the "Is-A" hierarchy, its properties, and the list of participants to the file.

void WriteMultiInherit(File* fp); The function writes the multiple inheritance constructs to the files.

void WriteAttributes(File* fp, int entityId); The function writes the list of attributes and properties for an entity or an arc to the file.

void Readheader(File* fp); Read the header of a pre-stored Local format file. The header consists of the filename, the format and the count for each type of construct in the EER diagram. The counts for each type of construct are updated to determine the number of records to be read for each construct.

void ReadCoddConstructs(File* fp); This function is responsible for sequencing the order in which the constructs may be read from the file.

void ReadDomains(File* fp); The Domain records are read and updated into the Domain structure.

void ReadEntities(File* fp); The File is read to update the entity structure for each record. The corresponding functions for creating the graphical symbol are called.

void ReadArcs(File* fp); The File is read to update the arc structure for each record. The corresponding functions for creating the arcs on the canvas are called for each of them.

void ReadIs-A(File* fp); The "Is-A" hierarchy records are read and updated into the respective structures. The functions for symbolizing the "Is-A" between the parent and base entities on the canvas is evoked for the set of related records.

void ReadMultiInherit(File* fp); The Multiple Inheritance structure is updated for each record representing the multiple inheritance construct. The corresponding functions for drawing the multiple inheritance symbol is evoked for each record.

void ReadAttributes(File* fp); The function reads the list of attributes and properties for all constructs of the EER diagram.

8. **translateEERText.c**

void InitTranslateEERText(); Function to initialize the Translation data structures to their default values. Load the Mapping structure to maintain a mapping between the EER Text constructs and their corresponding internal schema representations.

void WriteEERTextConstructs(File* fp); For all the internal EER Schema representations write the header for the EER text file and the remaining constructs in sequence.

void WriteTextHeader(File* fp); Write the header constructs for the file. The Header contains preliminary information of the EER-diagram document.

- void WriteTextDomain(File* fp);** The Domains for EER diagram are written in their equivalent EER Text Mark Up representation.
- void WriteTextEntities(File* fp);** This function writes the Entities and their properties in their Text Representation. Call the function to write the attributes.
- void WriteTextAttributes(File* fp, int entityId);** This function writes the Attributes and their properties for a given entity.
- void WriteTextRelationships(File* fp);** This function writes the Relationships and their properties in their Text Representation. Calls the function to write the attributes for the relationship. Calls the function to write the participants for a relationship.
- void WriteParticipants(File* fp);** This function is responsible for determining the participants of the relationship through their Identifier, and writing their corresponding name as participants.
- void WriteDependEntity(File* fp);** This function is responsible determining the dependent entities and writing their equivalent text construct.
- void WriteDependRelation(File* fp);** This function is responsible for determining the dependent entities and writing their equivalent text construct to the file.
- void WriteTextLink(File* fp);** This function is responsible for converting all internal representation of arcs with their properties to their equivalent text representation.
- void WriteTextIS-A(File* fp);** This function is responsible for converting all "IS-A" representation to their equivalent text representation. The participating entities are identified stored with their named representation.
- void WriteTextMultiInherit(File* fp);** This function is responsible for converting all Multiple Inheritance representation into their equivalent

text representation. The parent participants are converted to their equivalent name before their corresponding representation.

void ReadEERTextConstructs(File* fp); Call functions for initializing the Token stacks. Read the EER Constructs from the corresponding file. Read the file, with each line being a single file record.

void ConvertTokensToEER(char* aRecord); For each record read, convert it into a string of tokens. Each token is compared to the valid EER-Text constructs before pushing it onto the stack.

void InitTokenStack(File* fp); Initialize the Token Stack to null, and the count of elements on the stack to *nil*.

int GetElement(char* token); Get the EER Internal Representation for each token. Returns the equivalent entity identifier for the token representation.

void LoadTokenStack(char* token); Loads the Stack with the token.

char* PopTokenStack(); Pops the topmost token from the stack to return the token to the calling program.

int CmpToken(char* Token); Given a token find its corresponding token type.

void ReadTextHeader(File* fp) Reads the TextHeader for the EER Text Markup file, tokenize it and load the representation for the read information into its internal structures.

int ReadTextDomain(File* fp); Reads the Domain information for the schema from the Text File, tokenizes the read information and reads the converted information into its internal structure.

void ReadTextEntities(File* fp); This function is invoked for occurrences of the Text Markup Entity constructs. It reads, the tokenized graphical and location constructs followed by the attribute constructs for the entity. The converted information is stored into the entity structures and the

attribute structures. Depending upon the entity type, the correct routine for visually drawing the symbol on the canvas under create.c is evoked.

void ReadTextAttributes(int aEntity); Routine called by the Entity and Relationship functions to load the attribute properties for a specific entity into the attribute structures.

void ReadTextRelationships(File* fp); The Relationships are read in the textual formats and converted to their equivalent internal schema representations. All valid Relationship constructs then call the create routine for their graphical depiction on the canvas. Validation routine checks that the participating classes have been previously defined and updated to their internal representations.

void ReadParticipants(int arelation); This function is evoked from ReadTextRelationship() to convert the textual representation of the Relationship constructs to their internal representation and identifiers.

void ReadDependEntity(int aentity); This function is evoked for converting the dependent entities in the EER Text file to their equivalent internal representation.

void ReadDependRelation(int arelation); This function converts the Textual representation of each Dependent Relation to its internal schema representation.

void ReadTextLink(File* fp); The Links are Textual representation for Arcs on the EER diagram. This file reads all Textual links and converts them to their graphical and internal representations.

void ReadTextIS-A(File *fp); This function is responsible for reading the "IS-A" relationship and its properties and converting them to their equivalent internal schema representations, including updating the entity classes for which the hierarchy holds. Depending upon the kind of "Is-a", it calls the appropriate "Is-a" create routine to draw the correct representation on canvas. Validation routine includes, that the classes between which

the "Is-a" holds, have been previously read and converted to their internal representations.

void ReadTextMultiInherit(File* fp); Multiple inheritance constructs are converted into their internal representations. All parent classes and subclasses are necessarily updated prior to converting the multiple inheritance representations.

9. generateOB.c

int generateOB(File* fp); This function is invoked from the callbacks of file menu to convert the EER diagram to its equivalent OB construct.

void InitOBconstructs(); This function initializes the OB mapping constructs to its defaults on loading the Tool.

void ConvertDomains(File* fp); The function is responsible for converting the internal representation of domains to their equivalent OB representations.

void ConvertSimpleEntities(File* fp); This function locates all simple entities in the internal representation and writes their class and category definitions.

int FetchEntityAttributes(File* fp, int aentity); Fetches all attributes for a given entity and writes their equivalent membership representation for a class.

void DeclareClasses(File* fp, char* classname); Declare a class representation with a given classname.

void DeclareCategories(File* fp, char* catname, OrganizeType aot); Function declares the Category construct for a given category name and organization type for the category instances. The default Organize type is a *Set*.

void convertComplexEntities(); This function is called for converting the complex entities to their OB representations.

entitySet SearchComplexEntities(); This function locates all complex entities within the internal schema and returns their identifiers.

entitySet GetComponents(inf acomplex); Function gets the identifier components for a given complex entity.

void DeclareComplexClasses(File*fp, char* classname, Member* members);

This function declares the complex class. It is similar to the class definitions for a simple entity but additionally, includes the definition of its components as the membership constructs.

entitySet LocateIS-A(); Locates all the "IS-A" hierarchies in the internal representations and their types.

int LocateParentEntity(int ais-a); Locates the parent-identifier for a given "is-a" relation.

entitySet LocateSubEntity(int ais-a); Locates the subclass identifiers for a given "is-a" relationship.

void ConvertIS-A(int aisa-type, int parent, entitySet children);
Depending upon the "is-a" type, the corresponding class and category declarations are done.

int GetIS-AType(int ais-a); Utility function that returns the "is-a" relation type.

int LocateRelationships(); Locates all relationship constructs in the internal schemas. For each relationship construct the conversion routines are called.

void ConvertRelationships(int arelation); The conversion is done for a specified relationship.

void DeclareAssociations(int arelation); An Association construct is declared for each relationship. The Association name is same as the role-name of the relationship construct.

- relationSet LocateRelatedArcs(int arelation);** Utility function that locates the arcs for a given relationship. Each arc property identifies the rolenames of the participants, and the constraints with the participants.
- void ConvertRelationshipEntity(int arelation, relSet arelSet);** Based on the relation set, a declaration for the relationship as an entity-set is done.
- int GetMultipleInheritance();** Fetches the multiple inheritance constructs in the internal schema. Converts each such construct to its equivalent OB construct by calling the appropriate routines.
- void ConvertMultipleInheritance(int amultiplied);** Declares the multiple inheritance class for the specified identifier. It locates the parent classes for the identifier and declares them as the parent classes.
- int DeclareSubclasses();** Declares the subclasses for each "Is-a" relationship. The subclass declaration includes the parent class for its inheritance declarations.

References

- [ABD⁺90] Atkinson, M., Bancilhon, F., DeWitt D., et al. The object-oriented database system manifesto. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*. North-Holland, 1990.
- [BMS84] Brodie, M., Mylopoulos, J., and Schmidt, J. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Springer-Verlag, New York, 1984.
- [BPR88] Blaha, M., Premerlani, W., and Rumbaugh, J. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414-427, 1988.
- [Cat91] R.G.G. Cattel. *Object Data Management*. Addison-Wesley, Reading, Mass., 1991.
- [Che91] P. Chen. The entity relationship mode - towards a unified view of data. *ACM Transactions on Database Systems*, 1(1), March 1991.
- [DNF79] Dos Santos, C., Neuhold, E., and Furtado, A. A data type approach to the entity-relationship model. In *Proceedings of the International Conference on Entity-Relationship Approach*, 1979.
- [Gal92] L. Gallagher. Object sql: Language extensions for object data management. *ISMM Conference on Knowledge and Information management*, November 1992.

- [GH91] Gogolla, M. and Hohenstein, U. Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems*, 16(3), September 1991.
- [HM78] Hammer, H. and McLeod, D. The semantic data model: A modeling mechanism for data base applications. In *Proceedings of the ACM SIGMOD*, 1978.
- [Kim90a] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, Mass., 1990.
- [Kim90b] W. Kim. *The Object Model, Interoperability and Beyond*. ACM Press, AW, New York, New York, 1990.
- [Kim90c] W. Kim. Object-oriented database: Definitions and research directions. *IEEE Transactions on Knowledge and Data Engineering*, December 1990.
- [Kul94] K. Kulkarni. Object-oriented extensions in sql3: A status report. In *Proceedings of the ACM SIGMOD*, page 478, Minneapolis, Minnesota, 1994.
- [Lan73] B. Langefors. Theoretical analysis of information systems. *Student Literature, Lund.*, 1973.
- [Lin85] T. Ling. A normal form for entity-relationship diagrams. In *Proceedings of the International Conference on Entity-Relationship Approach*, pages 24–35, Chicago, 1985. IEEE Computer Society Press.
- [LZ92] Loucopoulos, P. and Zicari, R.(eds). *Conceptual Modeling, Databases and CASE : An Integrated View of Information Systems Development*. John Wiley & Sons, 1992.

- [NC83] Navathe, S. and Cheng, A. *A Methodology for Database schema mapping from Extended Entity-Relationship models into the Hierarchical model*. Elsevier North-Holland, New York, 1983.
- [NH89] Nijssen and Halpin. *Conceptual Schema and Relational Database Design - A fact oriented Approach*. Prentice-Hall, 1989.
- [NNJ93] Narasimhan, B., Navathe, S., and Jayaraman, S. On mapping er and relational models into oo schemas. In *Proceedings of the International Conference on Entity-Relationship Approach*, pages 402–413. Springer-Verlag, 1993.
- [Pet93] J. Petras. Modeling and implementation in an object-oriented database management system. In *MCS Thesis*, Carleton University, Ottawa Canada, 1993.
- [RBP⁺91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [SS77] Smith, J. and Smith, D. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), June 1977.
- [SSW79] Scheuermann, P., Schiffner, G., and Weber, H. Abstraction capabilities and invariant properties modeling within the entity-relationship approach. In *Proceedings of the International Conference on Entity-Relationship Approach*, 1979.
- [Str91] B. Stroustrup. *The C++ Programming Language, 2nd ed.* Addison-Wesley, Reading, Mass., 1991.
- [TGBK89] Teorey, T., Guangping, W., Bolton, D., and Koenig, J. Er model clustering as an aid for user communication and documentation in database design. *Communications of the ACM*, 32(8):975–987, August 1989.

-
- [TYF86] Teorey, T., Yang, D., and Fry, J.,. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2), June 1986.
- [Web81] N. Webre. An extended er model and its use on a defense project. In *Proceedings of the International Conference on Entity-Relationship Approach*, pages 175–194, Washington D.C, 1981. North-Holland.
- [Weg90] P. Wegner. Concepts and paradigms of object-oriented programming languages. *OOPS Messenger*, 1(1):25–27, August 1990.