APPLYING THE OBJECT-ORIENTED PARADIGM TO DATABASES BY EXTENDING THE RELATIONAL MODEL

by

HASHIM EL AMIN MUSTAFA

Thesis submitted in accordance with the
requirements of the University of Liverpool
for the degree of Doctor of Philosophy.

Department of Computer Science
University of Liverpool

December 1990.

CONTENTS

Acknowledgement

I would like to sincerely thank my supervisor Professor Michael Shave for his continued and invaluable advice, help and encouragement throughout the period of this work.

I am also grateful to all my colleagues and all the members of staff in the department of computer science at Liverpool University for their assistance.

I would like to acknowledge the funding support given to me by University of Khartoum, Sudan.

Chapter 1

Introduction

1.1 Motivation

1.2 Semantic Data Models

1.3 Object-oriented Paradigm

1.4 The Relational Data Model

1.5 OQUEL System

1.6 Outline of the Thesis

Chapter 1

Introduction

1.1 Motivation

The concept of a database occurs now to some extent in almost every area of information processing. However, the extension of database technology to new applications like engineering design, office automation, software engineering environments, AI applications, scientific applications, image processing promotes the need for more expressive models and new functionalities [Su86], [Stal86], [Wied86], [Karl88], [Gibb83], [Woel86], [Hitc87], [Gray88], [Pato88], [Moha88], [Jaga89]. These new emerging applications require management of databases that are more complex than those in traditional business applications such as banks and airline reservation systems. The requirements of these applications include (i) support for complex objects (ii) support for abstract data types (iii) support for higher-level interfaces (iv) support for integrity (v) schema evolution and version control (vi) complex relationships (i.e. spatial, temporal, procedural relationships (vii) types which describe behaviour rather than purely structure.

The relational data model is well known but over the last fifteen years there has been considerable research to replace the relational model with one having additional semantic constructs. Some of these constructs can be listed as follows:

(a) enriched collection of objects

entities,attributes and relationships [Chen 76]

classes [Hamm81] roles [Bach77]

set-valued attributes [Zani83]

unnormalized relations [Lum85]

aggregation [SmSm77]

molecular objects [Bato84]

(b) types of relationships

is-a hierarchies [SmSm77]

part-of hierarchies [Katz85]

convoys [Hamm81]

referential integrity (inclusion dependencies) [Date81]

(c) other constructs

ordered relations [Ston83b]

long fields [Lori83]

hierarchical objects [Lori83]

multiple kinds of nulls [Zani84]

multiple kinds of time [Snod85]

parameterized versions [Bato85]

table names as a data value [Lohm83]

recursion or at least transitive closure [Ullm85]

universal relations [Kort84]

unique identifiers [Codd79]

From the above list the following conclusions are evident: (1) there is a large collection of constructs, each relevant to one or more application specific environments. (2) the union of these constructs is impossibly complicated to understand and probably infeasible to implement with finite resources. The next generation database management system should aim to provide a support system that simulates the most important constructs.

Two developments which have attempted to resolve these difficulties are the semantic data models and the object-oriented paradigm.

1.2 Semantic Data Models

Semantic data models contain some general semantic constructs for defining the structural relationships, constraints, as well as behavioral aspects of data. They allow database designers and users to more explicitly define the semantic properties of their data. They provide a higher level of data representation than normalized relations in the relational model. A DBMS which uses this type of model, will be able to make use of the semantics of data captured in the schema of the database to behave more intelligently such as automatically resolving ambiguities in user queries, providing explanation to the user, and enforcing the constraints automatically to keep the database in a consistent state.

1.3 Object-Oriented Paradigm

There are several different programming paradigms, procedure-oriented, object-oriented, logic-oriented, rule-oriented, and constraint-oriented. The maturation of software engineering has led to the development of object-oriented analysis, design, and programming methods, all of which address the issues of programming-in-the-large. The object model provides the conceptual framework for object-oriented methods and it encompasses the following features:-

1.3.1 Object identity

Objects exist and can be uniquely identified independent of their representations or any of their properties. This implies a requirement for system-generated unique object identifiers. Pointers or array subscripts are inadequate as object identifiers because they force certain representation. "Primary keys" as used in relational database systems whose values are object properties, are inadequate as well because they are subject to changes by users. "Surrogates" as defined in RM/T [Codd79] and database systems that implement entity-oriented data models provide satisfactory

identification mechanisms.

1.3.2 High-level data abstraction

Complex objects found in engineering design and manufacturing tasks, for example, can be explicitly defined in terms of other objects. For example, a design object such as an automobile can be defined in terms of objects such as chassis, body assembly, engine assembly, wheel and brake assembly and so forth. The object, such as the body assembly, can in turn be defined by such objects as top, doors, interior, and so forth. The user of an object-oriented system is able to address the components as independent objects as well as the complex object defined by the top level of abstraction. Engineers can directly define and manipulate objects they deal with in their applications instead of dealing with the implementation representation of objects such as relations in a relational DBMS.

1.3.3 Uniform representation and communication

All things of interest in a complex application such as physical devices (tools, machines, computers, manufacturing parts etc.), and software entities (control programs, procedures, functions, and low-level system entities such as buffers, stacks, and queues) can be uniformly treated and represented as objects. Each object has its own private memory and public interface. Communication among objects is by message passing, but the procedure call mechanism of traditional programming languages can be used.

1.3.4 Separation of object specification and definition

In an object-oriented system, the specification of an object class in terms of an object class name, superclass and subclass relationships, instance variables, operations and parameters, and so on are all separated from the class definition, which contains the actual procedures and functions (i.e. methods) for executing the operations. This separation allows not only the ease of changing the implementation strategies (i.e. the procedures and functions) without affecting the user's view of the object classes, but it also provides the next very important property of the object-oriented paradigm.

1.3.5 Information hiding

The abstraction mechanism of the object-oriented approach provides an external view of objects and operations while hiding their representation and implementations. For complex objects, it provides a view of the object in terms of its aggregate characteristics while hiding the internal structure of the object in terms of its components. The ability to describe objects at multiple levels of abstraction encourages the decomposition of a problem into independent subproblems by hiding the details of lower level implementation (storage structures and algorithms). In order to make use of an object, the user (human user or program) only needs to know the specification part of the object class and not the inner workings of procedures and functions of that class. The user does not need to know in which programming languages the procedures and functions are written and in what hardware system the program runs. This information hiding property not only simplifies user programming tasks, but also allows accessing and sharing of the procedures and functions implemented by different people and organisations.

1.3.6 Inheritance

In the object-oriented paradigm, the objects of a subclass can automatically inherit the attributes and operations of its superclasses. This is similar to the concepts of generalization and generalization hierarchy introduced by Smith and Smith [SmSm77]. This concept can be extended so that constraints and knowledge rules associated with objects and object classes can also be inherited. If a new object class can be inserted into a proper place in an object class hierarchy, the structure properties, operational characteristics, and knowledge rules associated with its superclass can automatically be inherited without tedious respecification and definition. Furthermore, the implemented procedures and functions for performing the operations and for enforcing constraints and rules of these superclasses can also be used. This leads to the next very important property of the object-oriented systems.

1.3.7 Reusable codes

Programs written for an object-oriented systems are broken down into independent modules with well-defined interfaces. These modules are treated as objects and can be activated by passing the proper messages to them. They can be reused and incorporated into other programs since the functionalities and interfaces are well-defined. Thus, in a sense, programming in an object-oriented system involves composing a new program using the existing coded modules and adding new codes. Also, programs coded for an object class can be inherited by its subclasses. Thus, recoding of these programs for the subclasses is not necessary.

1.3.8 Polymorphism

An operator, function, or procedure in an object-oriented system can be implemented in different codes and can operate differently depending on the type of object to which the message is sent. For example, an operation PRINT can be implemented by two different procedures to print a character string or an integer, depending on whether the object to be printed is a character string or an integer. Instead of naming the procedures differently, they are given the same name (operator overloading). The proper procedure of the PRINT operation is dynamically bound to an object at run-time once the type of the object is known. This feature gives the user greater flexibility in naming operations associated with object classes and relieves the user from the burden of remembering different operations for different objects.

1.4 The Relational Data Model

Although the relational model has some limitations in supporting non-traditional applications, it has the following desirable features:-

1.4.1 Data independence

The independence of users and programs from details of the way the data is stored and accessed is critically important for at least two reasons: (1) it is important for application programmers because, without it, changes to the structure of the database would necessitate corresponding changes to applications programs. (2) it is important for end-users because, without it, direct end-user access to the database would scarcely be possible.

1.4.2 Sound theoretical base

Relational systems are based on a formal theoretical foundation. As a result, they behave in well-defined ways; and users have simple model of that behaviour in their mind that enables them to predict with confidence what the system will do in any given situation. This predictability means that the user interfaces are easy to learn and use.

1.4.3 Small number of concepts

The relational model is notable for the small number of concepts it involves. There can be little doubt that relations are easy to understand. The basic data construct is one, namely the relation or table; all information in the database is represented using this one construct, and more over this construct is both simple and highly familiar since people have been using tables for centuries.

1.4.4 Relational languages

The relational languages have the following desirable features:-

1.4.4.1 Set-level operators

Relational data manipulation operations such as SELECT, UPDATE, etc., in SQL (or RETRIEVE, REPLACE, etc., in QUEL) are set-valued operations. They are declarative and provide control abstraction. This fact means that users simply have to specify what they want, not how to get to what they want.

1.4.4.2 Closure property

The result of any operation of the relational algebra, or equivalent language, is itself a relation, which allows us to write nested relational expressions.

1.4.4.3 Symmetric exploitation

Relational languages generally provide symmetric exploitation, i.e., the ability to access a relation by specifying known values for any combination of attributes, seeking the values for its other attributes. Symmetric exploitation is possible because all information is represented in the same uniform way.

1.4.5 Compatibility

It is obvious that the user wants to view the same entity at sometime object-wise and at other times relational-wise. For example a design object which lends itself to object-oriented view needs storage of traditional business data for cost evaluation.

1.4.6 Relationships

Most object-oriented approaches support only one type of relationship among objects, IS-A, relating object sub-types to their super-types. Several other relationships have been proposed for engineering applications, including COMPONENT-OF, INSTANCE-OF, VERSION-OF [Bato85]. Instead of building in a few fixed relationships, systems must be capable of supporting user-defined relationships.

1.5 OQUEL System

Neither the relational nor the O-O approach is best for all applications. A DBMS is needed which provides several styles for describing and manipulating data.

We want to argue for combining features from three areas; semantic models, object-oriented paradigm, and the relational data model in a new model called OQUEL since it extends QUEL with object-oriented features.

OQUEL has the following features:-

It extends the data structures and operations of the relational data model and provides the desirable features of the OOPL paradigm and semantic data models such as improved semantics, data abstraction, reusability of data structure and code, extensibility, complex object support, schema evolution and ADT domains.

OQUEL introduces the object identifer (OID) to improve the semantics and reduce the space to store the database. OID captures the uniqueness of entities in the real world and allows modeling of complex objects. OID may be used to refer to an object instead of copying it.

OQUEL extends the relational model by inheritance to allow for sharing of data structures and operations and this improves the productivity of the programmer.

OQUEL provides two access modes for the database. One through the relational interface which helps to formulate unpredictable queries and provide flexible selection of a target list through different combinations of attributes - this is due to the no-information-goal-dependency of the relational formalism -. The other is a method-based interface which improves reliability by providing semantic integrity, managing complex objects and propagating updates through different semantic references, providing information hiding and operations for naive users.

OQUEL provides structuring for complex objects through specialized attributes

and provides operations through methods to enforce the integrity of the complex objects. Abstracting complex objects through tuples and relations provides for organisation of data by object or relation.

OQUEL provides extension for the domains of the relational model which simplifies queries and provides a more natural interface for spatial and temporal data.

OQUEL provides operations to manipulate evolution of structure as well as content. These make use of Ingres and C++ [Stro86], which provides object-oriented extensions to C language. We have provided an interface between C++ and Ingres.

1.6 Outline of the Thesis

Current database management systems and their data models which are discussed in *chapter two*, are limited in their support for these nontraditional applications. The requirements for these applications are discussed in *chapter three* as well as the limitations of the current data models and their implementations.

Motivated by the need for more powerful data models to capture and control more of the meaning of data stored in the database a number of semantic data models have been proposed. Most semantic data models were influenced by semantic networks and they are generally object oriented. These data models contain some general semantic constructs such as generalization (is-a), classification (instance-of), aggregation (part-of), for defining the structural relationships, and constraints. These models are discussed in *chapter four.*

The object-oriented paradigm, which was introduced in programming languages such as Simula, Smalltalk, C++, etc., can be extended to provide the basis for the development of more powerful database management systems. Several key concepts of the object-oriented paradigm are important and useful for a wide variety of database applications. These concepts are discussed in *chapter five.*

In *chapter six* we discuss the different approaches which include extensions to data models or systems in databases or object-oriented programming languages (OOPLs) to provide structural, behavioral, or full object-oriented databases.

In *chapter seven* we discuss the design and implementation of OQUEL model which combines features from semantic models, object-oriented paradigm and the relational model.

In *chapter eight* we illustrate through examples from the geological domain the applicability of OQUEL. The conclusions and suggestions for further research are discussed in *chapter nine*.

OQUEL architecture and OQUEL listings are contained in appendices 7A and 7B respectively.

Chapter 2

Basic Concepts and Characteristics

Chapter 2

Basic Concepts and Characteristics

2.1 Introduction

We begin this chapter by giving a brief history and evolution of databases from the early file structures. Then we discuss the classical data models , Hierarchical, Network, and Relational Data Models. We discuss their power and limitations which motivate the needs for more advanced models which will be considered in chapters *four* and *six*. Then we discuss three approaches to data modeling from a philosophical perspective.

2.2 Pre-Database Models

When the automated data processing era began in the 1950s and early 1960s, many organisations started transferring their manual operations to computerized systems that offered economical, high-speed, accurate data processing.Initially these were based on fragmented application dependent filing systems. A need for integrated access to the information gave rise to the concept of a generalized database management system. The DBMS is hence a general-purpose software system that facilitates the process of defining, constructing, and manipulating a database. A number of characteristics distinguish the database approach from the traditional approach of programming with files.

2.2.1 Self-contained Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database. In traditional file processing, data definition is typically part of the application programs themselves. Hence, these application programs are

constrained to work with only one specific database whose structure is declared in the application programs.

2.2.2 Insulation between Programs and Data

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require changing all programs that access this file. By contrast, DBMS access programs are written independently of any specific files. The structure of data files is stored in DBMS catalogues which are separate from the access programs.

2.2.3 Data Abstraction

The DBMS should provide users with a conceptual representation of data that eliminates most of the details of how the data is stored. A data model is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, which may be easier for most users to understand than computer storage concepts [Shav81]. Hence, the data model hides storage details that may not be of interest to most database users.

2.2.4 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files.

The DBMS has the following intended uses and advantages (1) Controlling Redundancy (2) Sharing of Data (3) Restricting Unauthorized Access to Data (4) Providing Multiple Interfaces (5) Representing Complex relationships (6) Enforcing Integrity Constraints (7) Providing Backup and Recovery (8) Reduced Application

Development Time

2.3 Classical Data Models

Classical data models include hierarchical, network, and relational data models. These models are discussed in sections 2.3.1-2.3.3.

2.3.1 Hierarchical Data Model

The hierarchical data model is the oldest traditional data model. Most of the model's development and implementation has been done by IBM in its information management system IMS software package. This model was developed to model the many types of hierarchical organisations that exist in the real world [Tsic76]. Hierarchies in the physical and natural world have been recognized and analysed for centuries. Humans have used hierarchical organisation for information to help them better understand the world. There are many examples such as classification schemes for species in the plant and animal world and classification schemes for libraries and governmental hierarchies.

2.3.1.1 Hierarchical Data Model Structures

The hierarchical model captures the relationships in terms of hierarchical definition trees. Figure 2.1 illustrates a possible hierarchical definition tree for the university example. The relationship between two entities is denoted by the parent-child arc. The parent-child relationship automatically supports referential integrity in the sense that no instance of a child record can exist without the existence of a parent record.

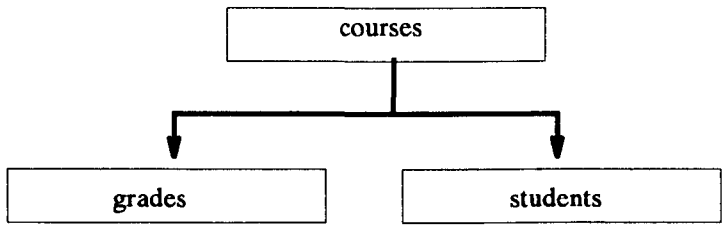


Figure 2.1 Hierarchical Definition Tree

2.3.1.2 Operations in the Hierarchical Data Model

The manipulation of data in the hierarchical model is record-at-a-time, unlike the relational model which is set-at-a-time. This model makes extensive use of the parent-child relationship and the ordering of subtrees for navigation.

2.3.1.3 Limitations

The hierarchical view leads to an asymmetrical perception of the universe in two respects: (1) The tree structure implies the relationship between entities have direction. The concept of father and son enforces this. (2) Things of the real world are classified as entities or attributes. There would appear to be no rationale for this distinction. Consider the colour of a plant. The colour is of as much interest to an artist as the plant to a gardener. The asymmetry of the view leads to asymmetric models and, to asymmetric implementation of access paths through data models. For example, deleting a parent record instance automatically deletes all corresponding child-record instances. Depending upon the application being developed by the hierarchical model, the data contained in the child-record may or may not need be retained. In addition the handling of some queries can be difficult if the queries do not conform to the hierarchical structure.

2.3.2 Network Data Model

The network model uses additional pointers to add flexibility to the hierarchical model. In its most general form, a network is a collection of nodes with links possible between any nodes. These links can be assigned meanings and the creation of links between nodes can be constrained in various ways. The hierarchical model for instance, is a special case of the network model where each node is linked to a parent node. In a pure hierarchy, each node may have only one parent although it may itself be the parent of more than one lower-level node. In a hierarchy, there is only one path between any two nodes, whereas in a network, there may be a number of paths. Figure 2.2 illustrates a network model for the university example.

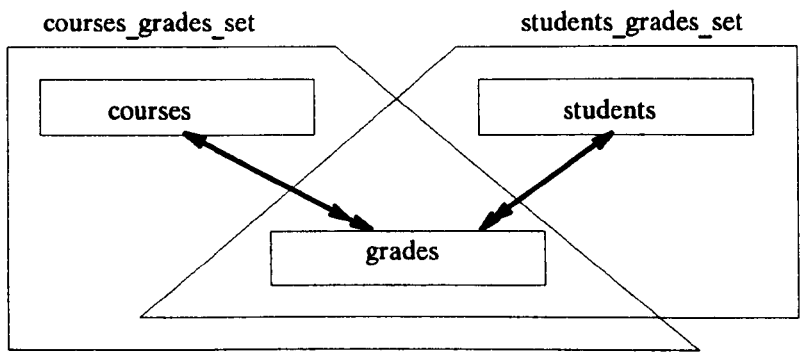


Figure 2.2 Illustrates a network model with two sets for university example

2.3.2.1 Network Model Data Structures

The basic data structures in the network model are records and sets.

Record, Record Types, and Data Items

An analogy can be drawn between network terminology and the relational model terms as follows :

Logical Record Format	Relational Scheme
Logical Record	Tuple
Logical Record Type	Relation Name
Data Item	Attribute

However, there is an important distinction between tuples of relations and records of a record type. In the value-oriented relational model, tuples are nothing more than the values of their components. Two tuples with the same values for the same attributes are the same tuples. On the other hand, the network model is object-oriented, at least to the extent that it supports object-identity. Records of the network may be viewed as having an invisible key, which is in essence the address of the record i.e. its object identity. This unique identifier serves to make records distinct, even if they have the same values in their corresponding fields. The reason it makes sense to treat records as having unique identifiers, independent of their field values, is that physically, records contain more data than just the values in their fields. In a database built on the network model they are given physical pointers to other records that represent the relationships in which their record type is involved. These pointers can make two records with the same field values different, and we can not make this distinction if we thought of only of the values in their fields. The distinction between tuples and records tells us a great deal about the ways in which each of the two data models - relational and network - excels. The relational model gives us the ability to use component values in arbitrary ways, whether or not they are the ways that were expected by the database designer when the scheme was first created. However, in the network model, where languages only allow us to follow links, there is no convenient way to express some queries.

Since the result of an operation on relations is a relation, we can build a complex expression of relational algebra easily. However, the result of operations on

networks is not generally a network, or even a part of one. It has to be that way, because the invisible pointers and unique identifiers for records cannot be referred to in a network query language. Thus a new network cannot be constructed by queries; they must be constructed by the data definition language.

2.3.3 Relational Data Model

The objective of the relational model was (1) to provide a high degree of data independence, (2) to provide a simple community view of the data, so that a wider community of users in an enterprise ranging from the most computer naive, to the most computer sophisticated can interact with a common view, (3) to introduce a theoretical foundation into database management, (4) to lift the database application programming to a new level- a level in which sets are treated as operands instead of being processed element by element.

2.3.3.1 Relational Model Concepts

Relations, tuples, attributes

The fundamental structural concept of the relational model is the table. A relation is a two_dimensional table of values where the columns of the table are called attributes and the rows are called tuples. Classical data-processing terms are similar, substituting data elements for attributes, records for rows, and flat files for relations. In Codd's view, a relational database system must support several structural features. The first feature is relations of degree n, that is tables with some arbitrary number n of attributes, in which the ordering of rows or columns is not relevant to the model. Tables are of four types.

Base Tables

Query Tables

View Tables

Snapshot Tables

Base tables are the actual data stored in the database, while query tables are the result of any query, which may at the user's option be saved in the database for

further operation. A relational database system is thus a mathematically closed system, i.e. all operations on relations in the database produce more relations. View tables are the dynamic result of one or more relational commands operating on the base tables to produce another table,called the view. These are virtual tables that do not actually exist in the database although their definition is stored in the system catalog tables but are produced upon request by a particular user, at the time of request. Snapshot tables are the resultant evaluation of view table invocation materialized at any particular time.

All information in a relational database is represented explicitly at the logical level and in exactly one-way by values in tables. That is why the relational model is called value-oriented compared with object/entity-oriented.

Domain

Another structural concept is the domain, which is the set of values from which individual columns (attributes) can be defined. The domain concept is important because it allows the user to define in a central place the meaning and source of values that attributes can take on. As a result more information is available to the system when it undertakes the execution of a relational operation, and operations that are semantically meaningless can be excluded. An example of a nonsensical operation is a join between the weight of a car and the age of a person.

Key

Another fundamental structural concept is the key, which provides associative access to records (tuples) in a relation without specifying the actual method by which such access is physically accomplished. The primary key is a combination of attributes whose values uniquely address each record in a relation. There should be a primary key for each relation in the database. Most existing products do not

support the primary key concept. A foreign key is an attribute in one relation table which can serve as a primary key into another table. Support for these concepts allows for all the associativity of the network data model independent of implementation.

Integrity

In the view of most information managers, data integrity is of pre-eminent importance in database management systems, and the more facilities that can be embedded in the DBMS product the better the guarantee of good data quality. There are three kinds of integrity :

Entity Integrity

Referential Integrity

User-Defined Integriy Constraints

Entity integrity guarantees the existence of a primary key for each record in a table. For example, if Social Security number is a primary key in a personal record, there can not be a record without a value for the Social Security number.

Referential integrity guarantees the existence of references to any foreign key. For example, if in a personnel record, an employee works in the payroll department, there must be a payroll department record in the table of departments.

User-defined integrity constraints are those specified by the users or database administrators. Having constraints stored in the DBMS catalog has the advantages of centralized control and enforcement. In the event of a constraint change, the change and enforcement will be performed in the catalog, without having to rewrite any of the applications. Also, because all constraints are stored only once, there will be no chance of having multiple versions of constraints, each enforced by different application programs.

2.3.3.2 Operations in the Relational Data Model

There are two rather different kinds of notations used for expressing operations on relations:

1. Algebraic notations, called relational algebra, where queries are expressed by applying specialized operators to relations, and

2. Logical notations, called relational calculus, where queries are expressed by writing logical formulas that the tuples in the answer must satisfy. One of the interesting facts about these notations for relational databases is that they are equivalent in expressive power, that is, each can express any query that the other can express.

Using selection and projection many natural questions can be asked about single relations. Many more queries are expressed by navigating among relations, that is, by expressing connections among two or more relations. It is fundamental to the relational model that these connections are expressed by equalities (or sometimes inequalities) between the values in two attributes of different relations. It is both the strength and weakness of this model that connections are expressed this way. It allows many varied paths among relations to be followed than in other data models, where, particular pathways are favoured by being built-in to the scheme design, but other paths are hard or impossible to express in the languages of these models.

2.3.3.3 Normalization

There are guidelines in the relational theory for database design that govern the "well-formedness" of relation schema, the so-called normal forms. These normal forms are designed to prevent data duplication, inconsistency, and update anomalies. Normalization is a step-by-step process for converting data structures into standard form relational tables. This standard form then satisfies the following constraints :

1. Each entry in a table represents one data item (no repeating group)

2. All items within each column are of the same kind

3. Each column has a unique name

4. All rows are unique (no duplicates)

5. The order of viewing the rows and columns does not affect the semantics of any function using the table.

2.3.3.4 Merits of the relational model

Declarative Languages:- The relational model gets its popularity perhaps from the way it supports powerful, yet simple and declarative languages with which operations on data are expressed. We may trace these capabilities to the fact that, unlike competing models, the relational model is value-oriented. That fact, in turn, leads to our ability to define operations on relations whose results are themselves relations. These operations can be combined and cascaded easily using relational languages. This property is lacked by Object-Oriented models because the result of a useful operation is often a new type. Such a type needs to have operations defined for it, so it can not become immediately the operand of another operation.

Data Independence:- In physical data independence we have the important notion of separation of the physical storage and performance aspects from the logical structures of data that an application program sees. Application programs remain logically unimpaired whenever any changes are made in either the storage representation or access methods. If, for example, the database administrator decides to drop an index on a column in a table , computer programs that access that table should continue to run without recompilation or any modification whatsoever. In many non-relational systems, a change to physical structure brings to a halt the working of applications programs that operate on the data. Thus, database modifications on such systems to tune

performance cause enormous overhead in requiring consequent application modifications to adjust to changes. Physical data independence allows database administrators full freedom to tune performance of the RDBMSs and to rearrange the physical organisations such as clustering tables or creating different kinds of indexes etc., without affecting running applications.

Logical data independence provides yet another level of insulation of application programs from the structuring of data. Under this concept, even the logical structure of the database may be altered, if done in such ways so as to preserve all information, without affecting the running applications. Key to this concept is the view structure which hides the logical structure alteration. For example, if two tables are combined without information loss, views can be defined which are projections of the combined table, leaving the appearance to the user with the impression that no change has taken place.

2.3.3.5 Limitations

The major contribution of the relational model lies in the elimination of data dependency and the successful formulation of high-level logical languages, both deriving from the uniform representation of information in terms of attributes. But the uniformity also implies the elimination of ways to express explicit relationships among objects which are so essential to the meaning of the world of objects, hence this is an overkill of data dependency. The semantic difficulty of classical models will be discussed in *chapter three.*

2.4 Philosophical Perspective to Data Modeling

The following data modeling approaches have been identitified for their importance in the data modeling field:-

Entity-based approach

Rule-based approach

Frame-based approach

2.4.1 Entity-based approach

"Entities are a state of mind, no two people agree on what real world view is;" [Metaxides]

Entity-based approaches to data modeling are so called because of the primacy given to the notion of an entity. That is, the central core of these approaches is the entity and its associated concepts. An entity, in this context, is the representation of an object of the real world, about which there is a desire to record information. Its associated concepts are the attributes and relationships, where the former describes properties possessed by entities and the latter association between entities. This category includes the semantic models. The entity-based approach to data modeling tends to follow in the steps of objectivist tradition. Under this tradition a data model is like a mirror or picture of reality. Reality is given "out there" and is modeled by entities. Entities have properties. Both entities and their properties have an objective existence. Entity-based approaches implement a picture theory of meaning i.e. data corresponds to facts, and it is these that entity-based approaches seek to model.

The entity-based approach can be classified into two broad approaches; the Entity-attribute-relationship (EAR) approach which includes ER model [Chen 76] and the Binary Relationship approach (BR). There are three general types in the BR approach; (a) entity-relationship, (b) entity-attribute, and (c) entity-function. Category (a) approaches function with the two primitives entity and relationship; examples include SDM [Hamm81], SAM [Su83]. Category (b) approaches possess

the primitives entity and attribute, an example of which is the relational model [Codd70]. Category (c) approaches are a variant of the other two using the primitives entity and function. An example of this category is the functional data model [Ship81]. The importance of entity, function and binary relationships for the conceptual schema has also been emphasised in [Shav81].

Although entity-based approaches have been widely used in information system development they have been criticized by the following (1) if there is a unique and objectively given reality there should be a unique way of modeling it. Yet the proponents of the entity-based approaches have not agreed upon one. For instance, in one model relationships are allowed to have attributes; in others it is illegal. (2) In the entity-based approaches, the meaning of each instance of an entity, attribute or relationship class is defined by a reference to the object or property it is to depict. However, defining meaning in terms of reference is inadequate. (3) In the realist theory the meaning of a sentence is said to be correct if it corresponds to an actual state of affairs. However, correspondence would have to be established through the social uses of words and symbols. This approach is disscussed in *chapter four.*

2.4.2 Rule-based approach

The rule-based approaches to data modeling are heavily influenced by the subjectivist tradition. Their proponents see the main task of data modeling as formalizing the meaning of messages and actions to be followed among a professional community. The expression of meanings must follow socially determined rules which facilitate the comprehension of what is communicated. Data can at best convey meaning from someone to someone, but they cannot have any objective meaning.

2.4.3 Frame-based approach

Central to the object-oriented approach is the concept that the objects of interest in the real world can most effectively be modeled through recording them together with the operations which are permitted upon them. In this way a move is made from representing purely structural aspects towards more integrated behavioural view combing both structure and operations. This approach is represented by SmallTalk [Gold83]. This approach can be used to implement both entity-based and rule-based approach. This approach is discussed in *chapter five.*

Chapter 3

The Need For a New Approach

Chapter 3

The Need For a New Approach

3.1 Introduction

Data is at the heart of any information system and as computing becomes more widely available, each discipline will come up with its unique requirements. Thus the potential for the application of database technology is expanding continually. In this section we wish to identify two major categories of applications outside the realm of traditional business type applications that are presenting good opportunities as well as a great challenge to database technology. The special modeling requirements of each application area will be highlighted and then the shortcomings of the classical models and their implementation are discussed. In chapters *four* and *six* some previous proposals for addressing these problems will be discussed while in *chapter seven* our approach is described in detail.

3.2 Requirements of Non-Traditional Applications

3.2.1 Computer_Integrated_Manufacturing (CIM)

Existing data models - such as relational, network, and hierarchical models - and the commercially available DBMSs based on these models are not entirely suitable for managing databases in a CIM environment. The reason is that these models and systems are designed mainly for managing business-oriented databases. As discussed later, many data types and semantic properties useful in CIM applications are not considered in these models. This inadequacy has motivated much research work in semantic data models in recent years, including the work by Chen [Chen76], Smith and Smith [SmSm77], Su [Su 86], Wiederhold [Wied84], Codd [Codd79], Hammer and McLeod [Hamm81], Shipman [Ship81], Brodie [Brod81a]. In the following we discuss the various modeling needs for CIM :-

3.2.1.1 Complex data types

Perhaps the most obvious inadequacy of existing business-oriented DBMSs for CIM applications is their very limited data typing capability. Only a few basic data types such as integer, real, character are recognized. More complex data types such as vector, matrix, set are not handled by the model or the DBMS, but by the application programs through some host programming language. Data with these data types is very common and is treated as a collection of basic data objects in CAD/CAM applications. For this reason the data should be processed by the DBMS directly using operators proper to these data types. For example, a vector of three elements determines the coordinates of a position in the working space of a robot. It should be allowed to be stored as the value of an attribute having a data type Vector_3 in a record, and an operator should be introduced to test the value of a coordinate to determine if the record should be retrieved or processed.

3.2.1.2 Complex relationships

These include temporal, positional, and procedural relationships. Modeling temporal dimensions in database systems has drawn considerable attention in database research for example in work by Bubenko [Bube77], Anderson [Ande82], Snodgrass [Snod85], Navathe [Nava88a], Tansel [Tans86], Ariav [Aria86], Ahn [Ahn86], Gadia [Gadi88]. In design and manufacturing environments, the order in which operations or activities take place is extremely important, since it can represent the temporal, positional, or procedural relationships of the data objects. For example, a workpiece passing through a number of workstations is operated on and modified by the equipment at those workstations. The data that describes the workpiece is continually changing. The sequence of these workpiece descriptions is important to a parts inspection system that attempts to identify the time, place, or step at which a defect occurs. Temporal, positional, and procedural relationships need to be captured not only at the low-level data elements by domains that contain ordered sets of integers or names, but also at high-level data elements by ordered sets of records, files, or subdatabases, which record data generated or processed by some ordered events or operations.

3.2.1.3 Complex objects

Many of the new applications deal with highly structured objects: parts in a part hierarchy may be composed of other parts, complex geographic features may be composed of other features, documents are composed of sections and front pages, etc. Facilities are needed for representing the internal structure of a complex object, while at the same time permitting the manipulation of the entire object as a whole. Facilities are also needed for capturing complex constraints and interrelationships among the components of a complex object (e.g. rules for mapping operations on

the object into operations on its individual components without violating any constraints on their interrelationships).

3.2.1.4 Schema evolution

Since CAD applications are such that they vary in their structure and content, the model must support an evolving design data. It is also necessary to keep old versions and create new versions of the same object.

3.2.2 Knowledge_Based Systems

Computers are tackling new and exciting tasks such as diagnosing, planning, scheduling, monitoring processes, etc. These applications have transformed the way we think about computing and brought a new perspective to programming - a knowledge perspective. They introduced a new type of database, an *expert* database - that stores not only values but "chunks" of knowledge. Knowledge can be embodied in a program as a procedure or as a data structure. This distinction corresponds to the philosophical difference between knowing *how to* and knowing *that*.

The primary use of knowledge in knowledge-based systems is reasoning, broadly defined as the accessing of information implicit in the knowledge base. The module that does the reasoning is known as the inference engine.

Knowledge bases may not seem to differ from databases. Both are data structures, or sets of data structures, in which information is stored and from which information is retrieved. Yet intuitively, knowledge bases and databases are not quite the same. There is no intrinsic difference between a knowledge base and a database. The difference is one of perspective. To view a set of data structures as a database is to be concerned with data-level issues: What data structures are used to store information ? What mechanisms are used to manipulate and access these structures ? Are they efficient ? Do they meet performance standards ? How is the data's integrity maintained ?

To view the structures as a knowledge base is to be concerned with knowledge-level issues: What do the data structures tell us about the world ? What kinds of knowledge are represented ? What does the system know at any time ? What follows from what it knows ?

Traditional databases, however, are primitive when viewed from the knowledge-level. Their expressive powers are weak; they are limited to positive instances of predicates and relations of individuals, and they exclude disjunction and

negation. Their inferencing capabilities are restricted to retrieval and the elements of reasoning are numerical comparisons and counting.

From the database perspective, however, knowledge bases are primitive. They are not particularly large, and they tend to be bound to applications; therefore they are not as public nor as persistent as databases.

Database and knowledge-based technologies can no longer ignore each other. Knowledge is information at a higher level of abstraction, typically generated by experts in some domain of expertise. The knowledge is used to define, control, and interpret data.

DBMSs provide two kinds of knowledge representation capabilities: extensional (explicitly stored facts in the database) and intensional (defined as views or by queries over the stored database).

Extensional knowledge includes the stored facts (specific facts about individual objects, e.g. airplane no. KRT123 consumed 3000 gallons of fuel on May 17); and metadata (schemas that describe general facts about classes of objects, e.g. airplanes consume fuel on missions). Extensional knowledge is represented using the data model of the DBMS.

Intensional knowledge is a collection of constraints (e.g. an airplane can be scheduled on a mission only in its range) and rules for deriving new data (facts) from stored data. In existing DBMSs, these rules are expressed as view definitions in the query language. The key problem in knowledge representation in DBMSs is that existing languages are too weak to capture all the essential intensional knowledge. Specifically, since existing query languages are first order, they are incapable of expressing recursion. As a result, this knowledge has to be embedded in application programs, and is unavailable to the DBMSs for optimization. On the other hand, logic-based AI languages such as PROLOG do permit recursive definition in rules.

AI applications requires not only the representation of complex objects such as

lists, trees; but also the representation of frames which incorporate both structural and procedural knowledge. They need the storage of facts and rules so as to be shared. They require the ability to introduce new object types, add new attributes and methods to existing objects and also add new specialization to existing types. They require access to objects from their key properties and access from object to related objects [Gray88].

As computers tackle new tasks in new domains, a new way of thinking about databases is required. A shift from the view of data as values - sets of uniformly formatted data types - to a view of data as chunks of knowledge. In object-orientation, (see *chapter five*) the chunks are viewed not as passive objects merely to be stored, retrieved, and manipulated, but as active objects. The stored object has declarative and procedural knowledge. The database user makes requests of an object, and it responds on the basis of its knowledge.

3.3 Limitations of the Classical Models and their Implementations

3.3.1 Record-Orientation

Record structures reflect the attempt to find efficient ways to process data. They do not reflect the natural structure of information. Historically, database systems evolved as generalized access methods. They address the issues of enabling independent programs to cooperate in accessing the same data. As a result, most database systems emphasize the questions of how data may be stored or accessed, but they ignore the question of what the data means to the people who use it. The record-based approach is such an ingrained habit of thought that most of us fail to see the limitation it forces on us. It didn't matter in the past, because business applications was record processing almost by definition. Much of the meaning of a record is supplied by the mind of the user, who intuits many real world implications which naturally follow from the data. If we look at the semantics which inherently reside in the record construct, we find the following presumptions about the nature of information:

* Any thing has exactly one type -- because a record has exactly one record type. We are not prepared for multiple answers to " what kind of thing is that?"

* All things of the same type have exactly the same naming conventions and the same kind of attributes -- because all records of the same type have the same fields.

* The kinds of names and attributes applicable to an entity are always predictable and don't change much -- because our systems presume stable record description in the catalog or dictionary.

* There is a natural and necessary distinction between data and data descriptions. We are accustomed to having record descriptions in catalogs and programs quite separate and different from data files.

* The name of the relationship occurring between two entities is not

information, since it does not occur in the data file.

* A record, being the unit of creation and destruction, naturally represents one entity. Any thing not represented by a record is not an entity.

* Such entities are the only things about which we have data. The key field of a record identifies one such entity; all other fields provide information about that entity and not about any other entity.

* All entities have identifiers. Or at least, all entities are distinguishable from each other. For any two entities we must know some fact which is different about them, which we can use to tell them apart.

* Each kind of fact about an entity always involves entities or attribute values of a single type. We don't expect different kinds of entities to occur in the "employer" fields of two people's records; the record system does not have any way of telling us which type is occurring in that field for a particular record occurrence.

* The entities or attribute values involved in a given kind of fact all have the same form of name representation. We do not have selfdescribing records which tell us which data type or format is being used in this particular record occurrence.

* There is an essential difference between entities and attribute values, and between relationships and attributes. The difference seems to correlate with the things which are or aren't represented by records. If there is a record, then the thing it represents is an entity, and a reference to it in a field comprises a relationship. But if there is no separate record for the thing, then a reference to it involves neither an entity nor a relationship; it is simply an attribute value.

* Relationships are not distinct constructs to be represented in a uniform way. Many-to-many relationships are usually entities in their own right. And the associations implied by multivalued attributes are also entities, even though they are not relationships. This all follows from being represented by distinct records. But one-

to-many relationships are usually not entities.

3.3.2 Type Definition

Most database systems supply a fixed set of primitive types such as integers, reals, and character strings and perhaps a few specialized types such as date and money. However, the lack of abstract data types in the current systems is perhaps the most significant type of weakness. There is no way to define a new abstract type in terms of its abstract properties and then define its implementation in terms of existing types. This kind of abstraction becomes even more important for complex objects that represent spatial and temporal data.

The constructors for higher-level types are also limited. The relational model supports tuples and sets of homogeneous tuples (relations). The hierarchical model supports segments and trees of segments, the network model has records and owned lists of records (CODASYL set). A value in a record itself can not be a structured data item , except for limited support of repeating fields.

The operations for higher-level types are induced by the type constructors and can not be extended. The operators are similar in the models: access or set a field in a tuple, segment or record; traverse a relation, tree or list in some order; select a record from a structured data item based on boolean condition. In addition, the relational model supports operations on entire relations such as project or join. Even so the set of operations can not be augmented within the database systems.

Database systems do not separate cleanly the type definition from data declaration. For example, in relational systems, it is seldom to define a relation type (scheme) independently of declaring a relation to be of that type. Thus redundant specification is necessary to declare several relations as instances of that type.

3.3.3 Structural Limitations

The data structuring capabilities of current database systems do not adequately support the complexity and variation that occur in the real world. Records of a given type must be identical in the structure. Every record of a given type must have the same fields and a field must draw its values from the same type in each record. At best there is allowance for null values or missing fields.

3.3.4 Modeling Power

Whenever data structures in a database system will not support the actual structure of information in the real-world - then the form of the real-world information gets over-simplified in the database, or it must be encoded into the available data structures. If the structure of the real-world is over-simplified, the utility and reliability of data is compromised. For example, if a database scheme only allows for a single name middle name, two people who are distinguished by middle name might become indistinguishable in the database. When information is encoded, such as flattening a set-valued field into several tuples, applications programs must deal with the encoding. Encoding information also means that a database needs extra integrity constraints to ensure that only legitimate encodings appear.

In the relational model we have ambiguity of the concept of key. The concept of key which plays the double role of identifying and describing entities is based upon two premises, that is, first an entity is identified by a selected set of attributes, and second, the selection can be made not for each individual entity but for type of entities.

In the relational model, by definition, relations represent relationships among domains. This means that, at instance level, tuples assert facts about attribute values. A close look at the relational calculus reveals that tuples are actually treated as objects rather than facts as firstly, they maintain identities through changes of

nonkey attribute values, and secondly, quantifications are applied not to attribute but to tuple variables.

Commercial database systems have not supported a hierarchy of types. We can not exploit the similarities between say, employees and managers so as define common operations. Type hierarchies are common in AI systems as have been suggested for data modeling by Smith and Smith[SmSm77].

Another problem with current systems is that update commands are machine-oriented commands which insert, delete, and modify parts of data items. Such changes do not necessarily correspond to any possible change in the real world, such as changing an employee's birthdate. Changes in the real world typically involve updates to several database objects. Hiring an employee could involve insertions in several relations. Being able to model real world changes is a powerful capability for a database system.

3.3.5 Separation of Languages

In the past, computer systems have placed more emphasis on programs than data. That focus manifests itself in the design of traditional programming languages, where data that exists for the life of the program (variables) is treated differently from data that persists after execution (files). Files generally provide for much less data structure that program variables, requiring user-generated encodings for structured values written to files. In the database world, data manipulation languages do not support arbitrary computation on database objects, necessitating an interface to a general-purpose programming language. One language must be embedded in the other. The problem with having two languages is impedance mismatch. One mismatch is conceptual - the data language and the programming languages might support widely different programming paradigms. One could be declarative language while the other is procedural. The other mismatch is structural - the languages do

not support the same data types, so more structure is reflected back at the interface. For example, we can access a relational database using QUEL from C, but when the time comes to do some computation, C can only operate at the tuple level. The relational structure is lost.

Chapter 4

Semantic Data Models

Chapter 4

Semantic Data Models

4.1 Introduction

Early database research concentrated on the physical structure of the database. Little attention was given to the user's perception of the data. The hierarchical and network models offer the user the means to navigate the database at the record level. The relational model adds a data structure level, eliminating the necessity of performing record level manipulation of the database. Modeling capabilities with these classical models are still closely related to the record structure of the database.

In the middle of the seventies, researchers attempted to simplify the design and use of databases by providing modeling structures that were capable of supporting the user's view of the data [Chen76], [SmSm77]. In response, a wide literature on semantic data modeling has arisen and several specific semantic data models have been proposed. Many of these models were introduced as schema design aids. For this reason the predominant emphasis was placed on providing explicit representation for much of the meaning associated with data in a database. A schema could first be designed in a high-level semantic model and then translated into one of the traditional models for ultimate implementation [MacG85], [Lyng86], [Lyng87], [Rumb88]. The emphasis of the initial semantic models was to provide mechanisms and constructs that mirror the prevalent kinds of relationships that arise frequently in typical database applications. Semantic models provide a richer set of data abstraction primitives for specifying databases than record-oriented models. Thus, semantic models simplify database conceptual design in that they allow the designer to easily capture more of the meaning or semantics of an application environment. This is where the name "semantic modeling" comes from.

Four principles of semantic database modeling can be identified. The most

basic is that data about objects, and relationships between them, should be modeled in a direct manner. As first introduced by the Entity Relationship Model [Chen76], such entity-based modeling allows the database designer and users to think in terms of their objects without the indirection resulting from the symbolic identifier necessitated by records and pointers. As highlighted by the Functional Data Model, a second basic precept of semantic modeling is that many relationships recorded in a database are functional in nature. Such relationships have also been termed "has_attributes". A third precept is the significance of the IS_A relationships, which specify the fact that one set of objects must be a subset of another set of objects as illustrated in figure 4.1.a. The final precept of semantic data modeling is to provide hierarchical mechanisms for building object types out of other object types. Perhaps the best known of these are aggregation (i.e. cartesian product) and grouping or association (i.e. finitary power set). For example automobile might be the aggregation of motor, body, wheels; and motor might be the aggregation of engine, radiator and battery, as illustrated in figure 4.1.b. As an example of grouping or association we have the means of transport of Paul's family as illustrated in figure 4.1.c.

The field of semantic modeling has begun to draw interest as more than just a conceptual design and documentation tool and is now being studied all along the research spectrum from theoretical analysis to physical design [Chan82], [King84], [Fram85], [Nixo87], [AbHu87]. A Database Management System based on a semantic model will facilitate the design and use of databases by producing modeling structures that are capable of supporting the user's view of the data. However, one problem inherent in modeling any subset of the real world is the difference between the human's perception of the enterprise and the computer's need to organize the structure in a particular way for efficient storage and performance.

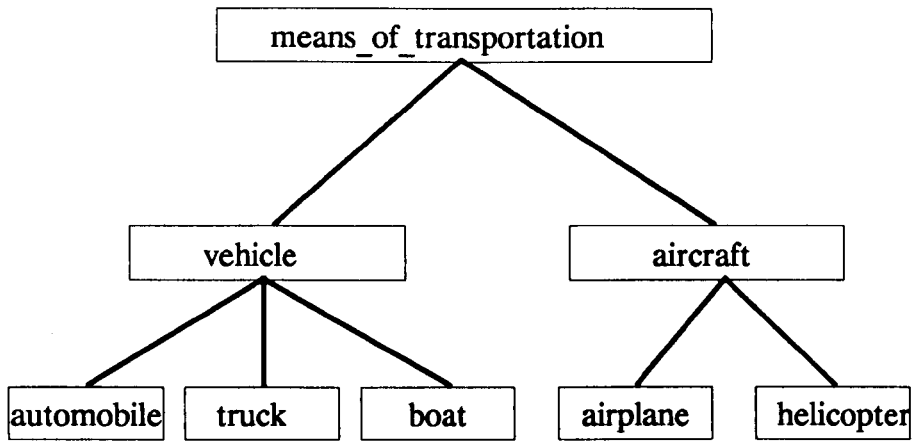


Figure 4.1.a Generalization Hierarchy

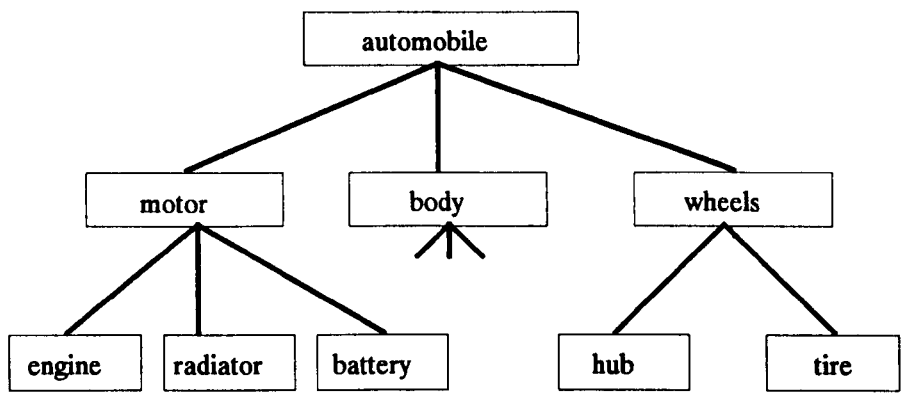


Figure 4.1.b An Aggregation Hierarchy

4.2 Semantic models and AI formalisms

Semantic modeling has its roots in semantic network research in artificial intelligence. A semantic network models knowledge as a collection of objects. The objects are interrelated by three types of relationships (is-a, is-instance-of, is-part) representing subtypes, memberships, and attributes. The major differences between semantic networks and semantic data models are that semantic networks mix schema and data and they do not provide a convenient way of abstracting the structure of data from the data itself. In a database a separate schema is necessary for

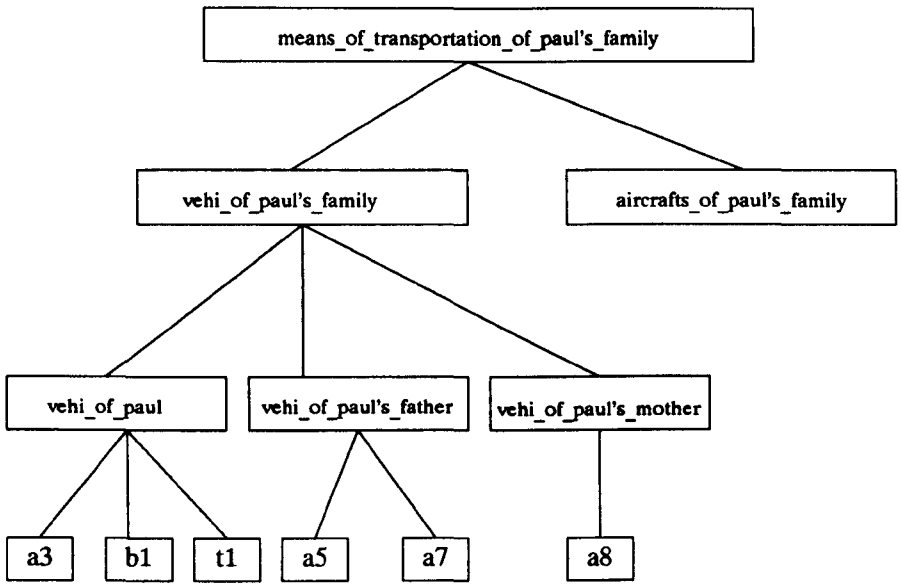


Figure 4.1.c An Association Hierarchy

handling a large amount of similarly structured data. Another difference is that a semantic network generally does not embody a data manipulation language.

4.3 Semantic models and OOPL

Semantic modeling relates to work in abstract data types and object-oriented programming. Semantic models encapsulate structural aspects of objects, whereas object-oriented languages encapsulate behavioral aspects of objects. Object-oriented languages are characterized by three principal features. (1) Explicit representation of object classes. Objects are identified by surrogates rather than their values. (2) Encapsulation of operations within objects. (3) Inheritance of methods from one class to another. The major difference between object-oriented models and semantic models are : (1) object-oriented models do not have the rich type constructors of semantic models. (2) the inheritance of methods is different from the inheritance of attributes. Semantic modeling differs from abstract data types in that an abstract data type operator is typically associated with only a single

type of object. A semantic model, on the other hand, typically supports operators which may be used directly to relate objects of different types.

4.4 Advantages of semantic data models

Semantically based data models and systems provide the following advantages over traditional, record-oriented systems.

(1) Increased separation of conceptual and physical components

In the record-oriented models the access paths available to end users tend to mimic the logical structure of the database schema directly. In the relational model a user must simulate pointers by comparing identifiers in order to traverse from one relation to another. In contrast the attributes of semantic models may be used as direct conceptual pointers.

(2) Semantic overloading

Semantic models provide several constructs for representing data interrelationships whereas record-oriented models provide just two or three constructs for representing such interrelationships. As a result, constructs in record-oriented models are semantically overloaded. In the absence of integrity constraints the data structuring primitives of the relational model are not sufficient to model the different types of commonly arising data relationships accurately. A primary objective of many semantic models has been to provide a coherent family of constructs for representing in a structural manner the kinds of information that the relational model can represent only through constraints. Indeed, semantic modeling can be viewed as having shifted a substantial amount of schema information from the constraint side to the structure side.

(3) Abstraction mechanism

Semantic models provide a variety of convenient mechanisms for viewing and accessing the schema at different levels of abstraction. One dimension of abstraction provided by these models concerns the level of detail at which portions of a schema can be viewed. On the most abstract level, only object types and is-a relationships are considered. At a lower level the structure of complex objects is considered. At further lower levels details about attributes and derivation rules are considered. A second dimension of the abstraction provided by semantic models is the degree of modularity they provide. Information about a given type, its subtypes, and its attributes can be easily isolated. Semantic connections can be easily followed to find closely associated object types. The above mechanisms are very useful in schema design and browsing. A third dimension of abstraction is provided by derived schema components, that permit the user to identify a specific subset of the data.

Examples in figure 4.1.d and figure 4.1.e summarize the abstraction concepts and provide an example based on the integrated abstraction concepts respectively.

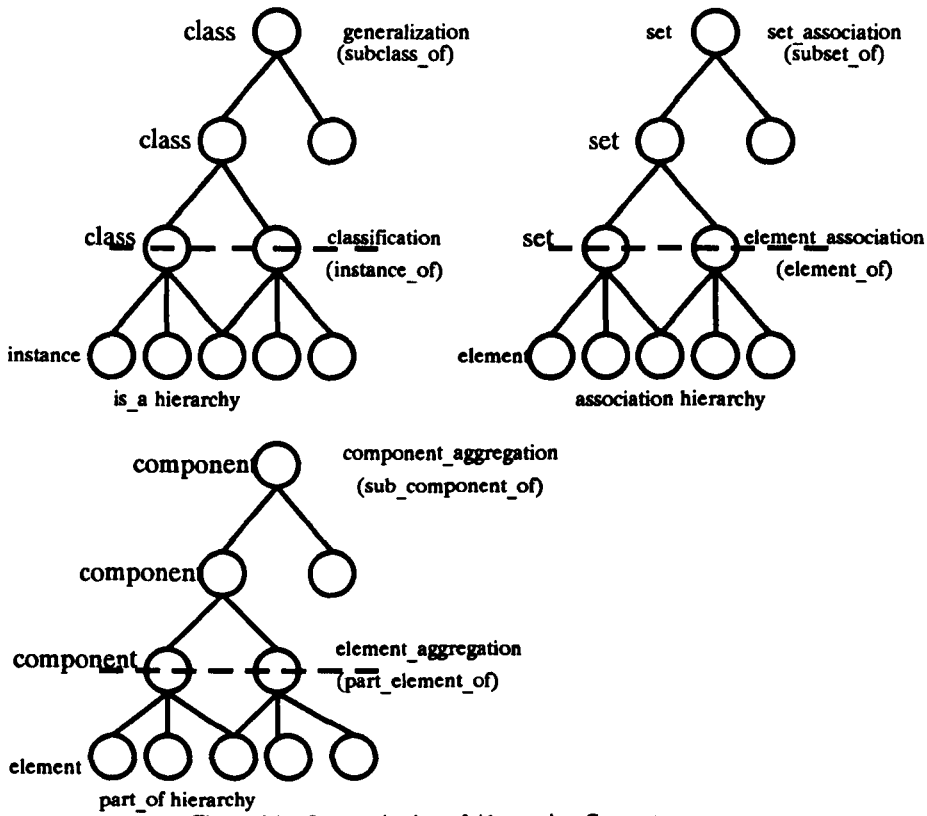


Figure 4.1.d Summarization of Abstraction Concepts

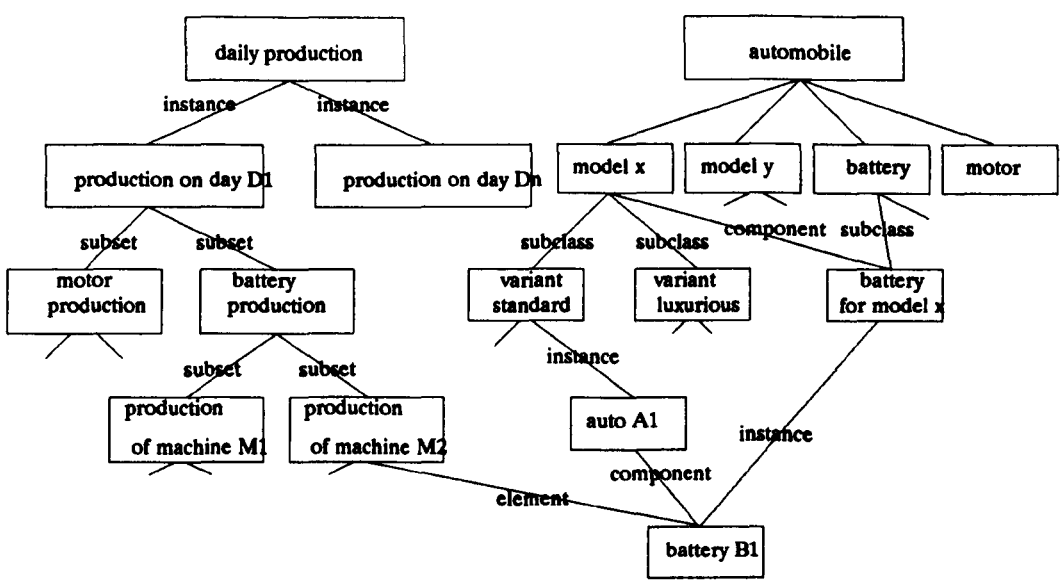


Figure 4.1.e Integrated Abstraction Concepts Example

4.5 Prominent Semantic Models

Semantic models can be classified into the following categories:-

1) Extensions of the relational database model

2) Binary database model

3) Functional database models

4) Entity/Association database models

5) Behavioral database models

4.5.1 Extensions of the relational database model

4.5.1.a RM/T

RM/T (Tasmanian model) [Codd79] is an extension of Codd's relational model [Codd70], attempting to capture more meaning in a conceptual model through the introduction of relationships and integrity rules. The relational model provides a tabular conceptual model in which all relationships between the tables are dynamically formed on the basis of data values in the tables. RM/T represents a means of enhancing the semantic expressiveness of the relational model while maintaining its fundamental character. For example, a type is represented by a unary relation that contains a symbolic unique identifier for every type member. Attributes of type members are represented in a separate n-ary relation that relates every unique identifier with a set of values for its attributes.

The orientation of this model is model is slightly different from that of other semantic models. RM/T arose out of the desire to handle database inconsistencies arising from insertion and deletion of tuples connected through interrelational

dependencies. Thus the model is information structure oriented, as is the relational model. Most other semantic data models provide similar modeling abstractions, but at the higher, conceptual level distinct from the underlying information structures. RM/T, however, still qualifies as a semantic model since the definition of these semantics not only gives more meaning to these relationships, but provides the data structures necessary to utilize the ususal data modeling abstractions.

This model, however, lacks the encapsulation of structure and behaviour, and modeling of complex objects with unpredictable structure.

4.5.1.b GEM

This model support entities and relationships as well as subtyping and non_atomic attributes. GEM [Zani83][Tsur84] is an attempt to provide a semantic front end that is compatible with existing relational DBMSs and to assess the usefulness of relational database machines in supporting semantic models. Like RM/T, this model lacks treatment of complex objects with unpredictable structure as well as encapsulation aspects.

4.5.2 Binary database models

The binary models attempt to supply a small universal set of constructs that are used to build more powerful structures. These models are based on the notion of "object-relationship-object". The structure of such models contains a collection of nodes and a set of binary links connecting them. Binary database models represent and treat data and the description of data uniformly. The binary relationship approach is appropriate to logic-based and fact-based systems as well as functional database systems. Interest in a binary approach arose from an awareness of the inadequacies of the record oriented approach. These inadequacies include the

following three points: (1) The record-oriented approach attempts to represent non-homogeneous objects (entity sets and relationship sets) in a homogeneous structure, the record or relation. (2) The relational approach represents a single type of object - the relationship - in various ways e.g. as an attribute, as a relation, and as a foreign key. (3) A particular object may be represented in one way as an attribute in one context, and in another way as an entity in some other context. The binary relationship approach does not have these limitations; its primary advantage is its generality and flexibility. Many things are simpler when we can deal with things pairwise, two at a time. Binary relations fit a simple linguistic model, two objects connected by a relating verb: people own cars, employees are assigned to departments, etc. They also lend themselves to a natural picture; a line connecting two points or nodes

People .___own_____. cars

However,the binary approach forces everything into this pattern, even when more than two things are involved. This exploits the fact that relationships are themselves entities, which can then in turn be related to other things. So, if three things are involved, we first link two of them to generate a new entity, which then gets linked to the third thing. There happen to be three ways of doing this. And the structures get more complex than they need to be, especially if even more than three things are involved. Another disadvantage with this approach is the difficulty of implementing efficient systems to manage data represented in this way.

The representative example in this family is SBDM [Abri74].

4.5.3 Functional database models

The functional data model DAPLEX was developed by Shipman [Ship81]. It is based on the notion of entities and functions from entities to entities. The significant contribution of the model is that it combines the computational power of application programming with the data definition and abstraction facilities provided by the database languages. The ideas of Shipman are based on the artificial intelligence notion of a semantic net. This is a structure which is used to represent association between objects. For each object of a given type there is a corresponding collection of functions which are applicable to it. Some of these provide a single value, but the results of others are found by following arcs in the net, which connect objects to other objects of various types. Functions can be applied in turn to these objects, thus exploring a network of associations. Usually the network is represented as a list structure with many pointers. The essential difference from relations is that instead of a set of explicit object data values we have a set of nameless objects whose properties may be given either as data values or as pointers to other objects. The properties can be obtained only through function application and if the results point to other objects, then further functions must be applied, thus resulting in a composition of functions.

The notion of a function is basic in mathematics. Functions may take objects of one type and produce a result of another type. The functional model generalizes the idea that the result is functionally dependent on the arguments, but it does not specify the precise representation of the arguments or results, only their types. The functional model works in two classes of items - entities and scalar values. Scalar values are single atomic values which have a literal representation. An entity is some form of token identifying a unique object in the database and usually representing a unique object in the universe of discourse. Thus two entities with identical component values can still be distinguished by having distinct references. Functions

defined over entities may return scalar values, entities or set entities. The data language DAPLEX [Ship81] was the first data definition and access language formulated entirely in the high-level terms provided by an object-oriented semantic database model. The most appealing property of the functional data model is its attractive and simple query facility based on functional composition. ADAPLEX [Chan83] is an experiment in efficiently implementing a semantic database using general-purpose operating system files and in embedding a semantic database language DAPLEX in the programmming language Ada.

The functional model is related to abstract data types. The notion of abstract data types is that a type is defined implicitly by the operations on it; only these operations are allowed to be applied to instances of that type, and only these operations have access to its inner structure and know the details of its representation. The functional data model uses this approach to define an entity. Instead of saying that it is represented by a record with certain contents, or by a tuple in a B-tree, it says which functions are defined on it. The functional model supports generalization which gives a hierarchy of subtypes and aggregation.

The functional model removes the sharp distinction between data and program and especially between objects in secondary storage and objects created in memory during the running of a program. We have just two kinds of functions : computable functions, like sum, max, etc which are better not tabulated, and stored functions, like Name(Person), which are not easily calculated. The functional model is related to the binary model, and thus allows one to add extra binary relations, representing new entities or relationships after the database has been in use for some time and hence providing extensibility. The EFDM [Kulk86] used a persistent language (PS-Alogl) as database implementation tool for an extended functional data model. DAPLEX has served as the basis of several object-oriented data models including IRIS [Fish86], Exodus [Care88], PDM [Mano86], which will be discussed in *chapter*

six.

4.5.4 Entity/Association database models

ER, SDM, and SAM* data models define a database as a collection of entities and relationships among entities. Typically a diagrammatic technique represents database schemas, and contains different sets of nodes and arcs corresponding to entities and kinds of interrelationships. Certain abstraction and information encapsulation primitives are utilized to organize database schemas, to support conceptual database design, or to provide a meaning_based user interface.

4.5.4.a Entity-Relationship Model

The Entity-Relationship (ER) Model,was proposed by [Chen76] as a schema design aid, permitting users to design schemas using a high-level object-based approach. The primary modeling constructs are the entity and the relationships. Schemas of this model have a graph-based representation and consist of types and relationships interconnecting these types, along with printable attributes of the types and relationships. The model provides strong support for a multiplicity of constraints (1:1,many:1,many:many). The ER schemas are translated into either the relational or the network model. The only abstraction directly supported in the original ER model is aggregation, although there are proposals to extend the model to include generalization [Teor86], categorization [Elma85], Object-Orientation [Nava88b].

4.5.4.b SDM

SDM [Hamm81] introduces a single abstraction, the class, to incorporate a wide range of modeling constructs. The intent is to permit the database designer to express the meaning of the database clearly with mechanisms designed to map directly onto the designer's concepts. Where most other semantic models provide primitives from which the designer can construct more conceptual objects, SDM attempts to offer a full set of modeling facilities. Classification and association have greater emphasis in SDM than aggregation and generalization. An SDM database is a collection of instances organized into classes or types. The designer defines classes and within this framework specifies member and class attributes, interclass connections, and derivations. SIM is a subset of this model which was implemented recently by Unisys. This model, however, lacks behaviour and encapsulation aspects.

4.5.4.c SAM*

SAM* [Su 86] is a semantic model designed originally for scientific -statistical databases and later extended to explicitly support computer-integrated manufacturing applications. In the SAM* the database can be modeled by a network of interrelated concepts. There are two types of concepts: atomic and non-atomic. An atomic concept, which can not be decomposed, is an observable physical object, abstract object, event, or any data element that the database user regards as a fundamental information unit. That is, its meaning is assumed to be commonly understood, and thus, it does not need to be defined in terms of other concepts. A nonatomic concept is a physical object, an abstract object, or an event whose meaning is described or defined in terms of other atomic or nonatomic concepts. To meet the requirements of CIM the following types and their operations are built into SAM* : (1) sets (ordered and unordered), (2) vectors and matrices, (3) time and time-series, (4) text, (5) G-relations (generalized relations).

The G-relation is an extended relation whose attributes may be of any valid SAM* types, including relation. Seven association types are distinguished in SAM*.

(1) Membership association (M):- used to define a domain of values, i.e. to group together a set of similar atomic concepts. The types of these concepts can be simple or complex.

(2) Aggregation association (A):- used to define an object type (atomic or nonatomic) as a grouping of a set of characteristic attributes. Each attribute is usually defined by an M-association. It can also be any other association type.

(3) Interaction association (I):- if a set of events or facts are the result of some interactions (actions or relationships) among the occurrences of independent entity types defined by an aggregation association, then the entity types can be grouped together to form an interaction association.

(4) Generalization association (G):- used to group objects together according to their generic nature to form a more general concept type in a similar way to the generalization abstraction mechanism.

(5) Composition association (C):- a grouping together of similar or dissimilar object types to form a more general concept type. It is used to model the semantics of IS-PART-OF, i.e. aggregation abstraction mechanism. The composition association groups together concept types, while the aggregation association groups together attributes.

(6) Crossproduct association(R) :- a grouping of some concept types to form a new concept, the occurrences of which are the occurrences of the cross-product of the component concept types.

(7) Summarization association (S):- essentially useful in statistical databases, it is used in conjunction with the cross-product association. This model also lacks encapsulation aspects.

4.5.5 Behavioral database models

4.5.5.a The Event Data Model

The Event Model [King82] is a semantic data model that defines the concept of 'events' to model database dynamics. This model provides integrated facilities for database specification, design, and manipulation. A subtype relationship is used to organize the static schema into a set of hierarchies. Membership in a subtype is defined using predicates evaluated on attributes. This model is not appropriate for applications in which the flow of information is not fixed or routine.

4.5.5.b SHM+

The Extended Semantic Hierarchy Model (SHM+) [Brod84] is a behavioral database model addressing the problem of modeling the static and dynamic portion of an application. Specification of data objects and associated transactions are performed using an abstract data type philosophy. The basic modeling constructs of SHM+ are primitive objects and operations, composition rules for hierarchically forming more complex objects and operations, and constraints to be applied to all primitives, composition rules, and hierarchies. The most important aspects of SHM+ are its contribution to dynamic data modeling and a consistent modeling methodology for both dynamic and static schemas.

4.5.5.c TAXIS

TAXIS [Borg85] is a language for the design of interactive database systems that places emphasis on classification and generalization/specialization hierarchies. The data model combines ideas from programming languages and database theory in order to support the following capabilities: (1) data encapsulation (2) semantic data modeling. Specialization is extended from the static to dynamic

portion of the database definition. Transactions are modeled as classes within the same taxonomic framework as object classes. Transactions are therefore closely related to the abstraction hierarchy of the data with refinement by specialization.

4.6 Summary

Although there are thus many conceptual modeling approaches they agree that the main objective of semantic data modeling is to facilitate the modeling of and use of the database. They also agree that two important steps towards this objective are that a semantic model should provide relationships between objects that support the manner in which the user perceives the real world, and that, for these relationships, a semantic model should contain semantics that specify the acceptable states, transitions, and responses of the database system.

The model developers vary in their perceptions of the following:

(i) whether models should be application independent or targeted toward specific environments, such as SAM*.

(ii) whether the relationships should be highly developed packages with all semantics built in (insertion/deletion constraints, cardinality constraints, etc.) as in SAM* or whether the database designer should have the option to specify the semantics of each relationship in an explicit manner as in ER.

(iii) whether relationships should be complex or primitive in their structure such as binary models.

(iv) whether relationships are distinguished from entities at the conceptual level. In ER models relationships act as primary modeling elements with semantics which differ from those of entities. On the other hand, in the functional model both entities and relationships are represented as functions.

(v) which abstraction (e.g. classification, aggregation, generalization) should be

emphasised. Certain models stress one or two abstraction as their primary modeling tools. SDM for example makes heavy use of classification and aggregation, whereas TAXIS stresses generalization hierarchies.

(vi) what approaches to dynamic modeling should be followed. SAM* provides abstract data types, in contrast to the generalization hierarchy approach in TAXIS and the flow of information approach of the event model.

Object-oriented database models which will be discussed in *chapter six* are fundamentally different from semantic models in that they support forms of local behaviour in a manner similar to object-oriented programming languages. This means that a database entity may locally encapsulate a complex procedure or function for specifying the calculation of a data operation. This gives the database user the capability of expressing, in an elegant fashion, a wider class of derived information than can be expressed in semantic models. On the other hand, object-oriented models are similar to semantic models in that they provide mechanisms for constructing complex data by interrelating objects.

Chapter 5

Object-Oriented Concepts

Chapter 5

Object-Oriented Concepts

5.1 Introduction

Object-Oriented models have appeared under many different guises. They have evolved independently in the areas of programming languages, database systems, and knowledge representation [Gutt80], [Card85], [Stef86], [Pete87], [Schr87], [Stro88], [Meye88], [Thom89], [Maie86], [Bloo87], [Kim89], [Atki89], [Sheu88], [Pate90]. Only in the past few years have researchers in these areas recognized the similarities and differences among object-oriented paradigms.

In sections 5.2-5.4 we present the evolution of object-oriented models in each of these computer science disciplines including a discussion of unique features and limitations. Then in section 5.5 we discuss the basic concepts of the object-oriented paradigm and the implications of introducing these concepts in the database field. Section 5.6 concludes the chapter by commenting on the limitations of the object-oriented paradigm.

5.2 Object-Orientation in Programming Languages

Statements, modules, and types are associated with three distinct paradigms of computation, namely the state transition, communication, and classification paradigms. The state transition paradigm views a computation as a sequence of states that are transformed by computation. The communication paradigm views the world as a collection of communicating agents. The classification paradigm views computation as a classification process that progressively refines the class in which a value must lie. Object-Oriented programming languages (OOPL) are characterized by their methods for structuring and processing data. Class data structures are the main data type, and hierarchies of classes and subclasses are constructed using language

primitives. Classes are instantiated to produce specific instances of class objects. A goal of OOPL, derived from the study of data abstraction, is to manage class objects as self contained entities or objects. Objects interact with each other through well-defined interfaces. A class is defined by its own attributes and also inherits attributes from its superclasses. Likewise, subclasses inherit procedures for manipulating instances. The internal structure of objects, and methods for processing objects are hidden within an object's definition, realizing the concept of abstract data types (ADT).

Simula, developed in 1967 as an extension of Algol-60, is one of the pioneer OOPL. The successor to Simula and purest OOPL is Smalltalk [Gold83]. Unlike its predecessor which includes traditional data types as integers, reals, strings and arrays, the only data types which Smalltalk supports are classes and instances. The internals of an object, namely its properties and processing routines, are hidden from other objects. In Smalltalk all computations are performed by message transmission, therefore the message-passing paradigm has come to be closely associated with OOPLs.

An OOPL entails other features such as overloading, late-binding, and interactive interfaces. These capabilities however further describe the functionality of an OOPL, they are not requisite definitional properties such as object-identity, data abstraction, property and method inheritance, and message-passing.

5.3 Object-Orientation in Knowledge-Based Systems

Knowledge-Based systems allow us to organize the explicit knowledge of experts about a given universe of discourse not only to retrieve the explicit knowledge, but also to extract implicit knowledge by means of an inference mechanism.

The general notion of an object in the sense of modeling real-world

phenomena, then , clearly applies to knowledge-based systems as well. Experts identify the entities of interest in the universe of discourse, and state facts and general rules about them.

According to Mylopoulos [Mylo84], knowledge representation considers the world as a collection of individuals and as a collection of relationships that exist between them. The collection of all individuals and relationships between them at any one time constitutes a state, and there can be state transformations that cause the creation and destruction of individuals or that change relationships between them. Depending on whether the starting point for a representation scheme is individuals/relationships, true assertions about states, or state transformations, we have a network, logical, or procedural scheme.

Logical representation schemes employ the notion of constant, variable, function, predicate, logical connectives to represent assertions about states as logical formulas. An important advantage of these schemes is the availability of inference rules in which one can define proof procedures for information retrieval, semantic constraint checking, and problem solving. The most serious drawback is the lack of organisational principles which keeps structure from being imposed on the set of axioms, so that a flat knowledge base of several thousands axioms becomes unintelligible to a reader.

Procedural representation schemes view a knowledge base as a collection of active agents or processes. In doing so they deal with two issues : the activation mechanism offered for processes, and the control structures for them. Thus the knowledge base contains assertions and a collection of demons that watch over it and are activated whenever the database is modified. The control structures have to deal with search and backtracking strategies. As a draw back , procedural schemes are difficult to understand and modify.

A semantic network represents knowledge in terms of a collection of objects

(nodes) and binary assertions (edges), where the former stands for individuals or concepts, and the latter stands for binary relationships over them. Many of the concepts like classification, generalization, inheritance have arisen in semantic networks so as to provide organisation for knowledge.

Frame-based representation languages result from research into semantic networks. A frame is a complex data structure for representing a stereotypical situation. The frame has slots for the objects that play a role in the stereotypical situation, as well as relations between these slots. Attached to each frame are different kinds of information such as how to use it, what to do if something unexpected happens, and default values for slots. What differentiates frame languages from database languages is inheritance. What also distinguishes frame languages from database languages is that certain inference mechanisms can be associated with frames, and automatic classification of new objects may be provided.

Frames [Mins75] represent a way to combine declarations and procedures within a knowledge representation environment. The fundamental organizing principle underlying frame systems is the package of knowledge. In packaging and representing knowledge, frames provide five major functions. These functions are: (1) Naming: A unique name is assigned to each object. (2) Describing: The body of an object is composed of a number of attributes that have values. The attributes describe properties of the object or link different objects together. (3) Organizing: Each object except the top-level in the hierarchy has one or more parents, providing an inheritance mechanism. (4) Relating: the values of object attributes may be other objects. Objects may thus be related by having one object as the value of an attribute in another object. Objects may also be related to rules. (5) Constraining: Each attribute in an object may have attached predicates, which are invoked whenever the attribute is read or modified.

5.4 Object-Orientation in Database Systems

In the field of databases "object-oriented" used to be synonymous with "entity-oriented" and is best contrasted with the relational model. In the relational model data organisation is based on the mathematical definition of a relation : the cartesian product of two or more domains. A database relation modeling a real world situation contains a subset of the crossproducts of domain values of relevant attributes. Each element of the subset corresponds to a relation tuple. Data items or tuples are accessed primarily by the relation name and secondarily by values of attributes within the relation. Tuples in a single relation can only be distinguished by values of the composite attributes. To retrieve a complete description of an entity may require accessing many relations and selecting only the tuple whose values correspond to some key for the entity in question.

An entity-oriented database, however, associates a unique identifier with a real world entity. Data retrieval is based primarily on object identity. Once an entity is accessed, attribute values and relationships components can be selected.

The Entity-Relationship model was originally developed as a database design tool to model reality in terms of entities and relationships among entities. Although the original goal of the ER model was to conceptually unify the hierarchical, network, and relational models [Chen76], the ER model has gained its own recognition and is the foundation for many object-oriented database management systems (OODBMS) [Ditt87], [Nava88b].

Three approaches to object-orientation in databases are identified. These include structural, behavioural, and full object-orientation. These approaches are characterized as follows:- (1) In structural object-orientation more complex data structures are available to model more complex structures of objects in applications. Applications manipulate data structures and are sensitive to changes in the structures used to implement the application. (2) Behavioural object-orientation allows

applications to be expressed in terms of operators which are semantically meaningful to the application domain, but without benefit of complex data structures that might improve the efficiency of the application. Applications can be designed in object-oriented fashion, yet implemented with methods that map to conventional data storage such as files and relational data structures. Such applications are relatively data independent, and enjoy most of the benefits of object-orientation: ease of development, extensibility, maintainability, adaptability, etc. They can be migrated to more efficient implementation, including various forms of structural object-orientation as they become available, in object-oriented fashion by altering the methods that implement the operators. (3) Full object-orientation combines the structural and behavioural approaches. These approaches will be discussed in *chapter six.*

5.5 Basic Features

In this section we discuss the fundamentals concepts of object-orientation and the implications of introducing them in the database field.

5.5.1 Object Identity

A fundamental and powerful object-oriented concept is object identity. Object identity has long existed in programming languages. The concept is more recent in databases [Khos86]. An identity is a "handle" which distinguishes one entity from another. In a model with object identity, each object will be given an identity that will be permanently associated with the object, immaterial of the object's structural or state transitions. For example, each of us as a person undergoes structural or state transitions. We grow older, we graduate from several schools and then start a professional career. We acquire new attributes such as a spouse, children, or excess weight. We might change our name. Yet, no matter how many additional attributes we acquire, modify or drop, there is presumably something unique about each one of us that is permanently associated with us.

Object identity brings these characteristics of the real world to languages and computation. Without object identity it will be awkward if not impossible to assign attributes or instance variables to a self-contained object. Also without object identity it will be awkward to make the same object a component of multiple objects if required.

Most programming and database languages use variable names to distinguish objects. This confuses addressability and identity. Addressability is external to an object. Its purpose is to provide a way to access an object within a particular environment and is therefore environment dependent. Identity is internal to an object. Its purpose is to provide a way to represent the individuality of an object independently of how it is accessed. Address-based identity in programming

languages compromises identity.

5.5.1.1 Identity in the Relational Model

In the relational model an identifier key is some subset of the attributes of an object that is unique for all objects in the relation. For example, year and model of a car. Using identifier keys as object identity confuses identity and data values or object state. There are three problems with this approach:

Modifying identifier keys:

Identifier keys will not be allowed to change, even though they are user-specified descriptive data. For example, a department's name may be used as the identifier key for that department and replicated in employee objects to indicate where the employee works. But the department name may need to change under a company reorganisation, causing a discontinuity in identity for the department as well as update problems in all objects that refer to it.

Non-uniformity:

The choice of which attributes to use for an identifier key may need to change. For example, Company X may use employee numbers to identify employees, while Company Y may use social security numbers for the same purpose. A merger of the two companies would require one of these keys to change, causing a discontinuity in identity for the employees of one of the companies.

Unnatural join:

The use of identifier keys causes joins to be used in retrievals instead of path expressions, which are simpler. For example, suppose we have an employee relation Employee(EmpName,SS#,Salary,CompName) and a company relation Company(Name,Budget,Location) and the CompName attribute establishes relationships between an employee and a company. Using identifier keys, CompName would have as its value the identifier key of the company, for example, Name. A retrieval involving both tuples would require a join between the two tuples.

For example, in SQL to retrieve for all employees the employee name and the location the employee works in, we would use

SELECT Employee.EmpName, Company.Location

FROM Employee, Company

WHERE Employee.CompName = Company.Name

The point here is that joins are unnatural; in most cases what the user really wants instead of the CompName is the actual company tuple. With normalization the user is restricted to a fixed collection of base types and is not allowed to assign and manipulate tuples, relations, or other complex object types of attributes. Hence, normalization loses the semantic connectives amongst objects in the database. In fact, relational languages such as SQL incorporate additional capabilities like foreign key constraints to recapture the lost semantics.

Non-First Normal Form Models allow a more direct and intuitive representation of object spaces. Consider the normalized representation of persons with spouses, education, and children as described in figure 5.2.1.a. If we allow relation-valued attributes, we can have a more compact representation for Education, which becomes a nested relation [Scho86]. This is illustrated in figure 5.2.1.b. However,

the children are still represented separately, since each child pertains to both parents. Furthermore, the spouses refer to each other through a foreign key.

Persons	Name	Age	Address	Spouse
	John	35	20 Park Road	Mary
	Mary	32	20 Park Road	John

Education	Person	Degree	University	Year
	John	M.Sc.	Leeds	1981
	John	Ph.D.	York	1985
	Mary	M.Sc.	Manchester	1979
	Mary	Ph.D.	York	1986

Children	Person	CName	CAge
	John	Tim	5
	John	Jane	3
	Mary	Tim	5
	Mary	Jane	3

Figure 5.2.1.a Normalized representation of persons with spouses, children and education

In the vast majority of cases the user wants to directly reference the Children, Spouses and Education of a person. The first normal form constraint of the relational model forces the programmer to normalize everything. Non-First Normal Form models provide a partial solution. What is needed is the ability to share objects. The concept of sharing is somewhat confusing and has been used in

Persons	Name	Age	Address	Spouse	Education		
	John	35	20 Park Road	Mary	Degree	University	Year
					M.Sc.	Leeds	1981
					Ph.D.	York	1985
	Mary	32	20 Park Road	John	Degree	University	Year
					M.Sc.	Manchester	1979
					Ph.D.	York	1986

Children	Person	CName	CAge
	John	Tim	5
	John	Jane	3
	Mary	Tim	5
	Mary	Jane	3

Figure 5.2.1.b NFNF representation of persons with spouses, education and children

different connotations by AI, Object-Oriented programming languages, and database communities.

In a database framework, sharing relates to synchronizing concurrent access to objects to ensure the consistency of information shared in the database. In an object-oriented framework, sharing relates to the support and maintenance of the references of the shared objects. A reference to an object implies shared ownership by all referencing objects.

So unlike the database perspective, where the users of an object could be thought of as concurrently executing transactions, the users in the object-oriented world are themselves objects owning or referencing the same shared entity.

Object identity supports and enables the referential sharing of objects. The

persons example with object identity is illustrated in figure 5.2.1.c. (Note o1 and o2 stand for objects and * stands for sets). Thus arbitrary graph structured object spaces are easily represented.

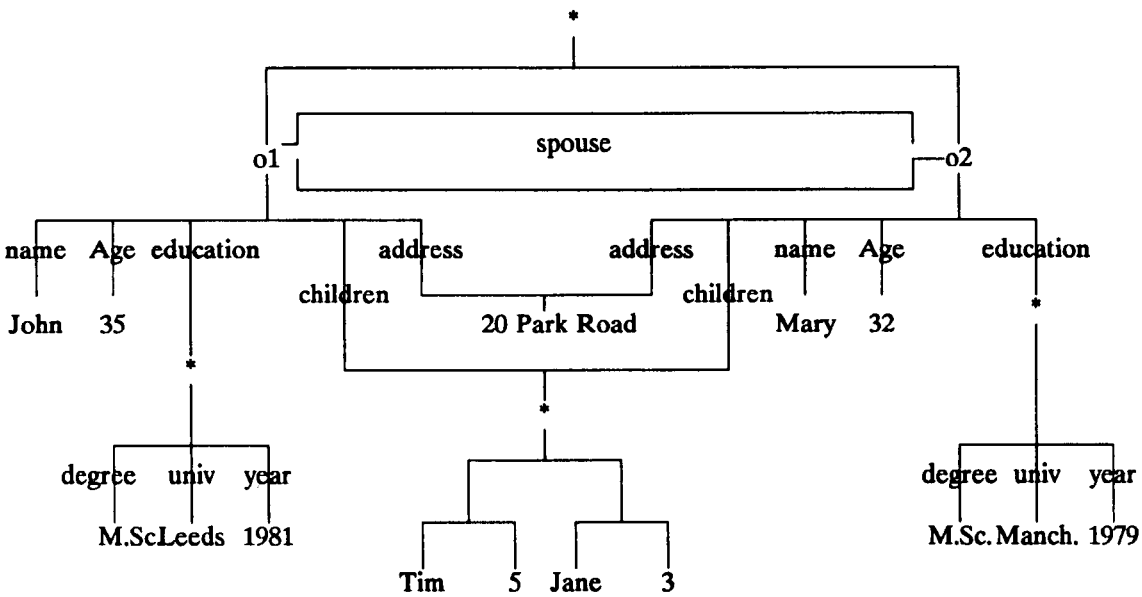


Figure 5.2.1.c Representation of persons with spouses, education and children using object identity

5.5.1.2 Identity in the Network Data Model

The idea of identity in object-oriented systems is shared with the earlier network data models (e.g. CODASYL). However, there are many other features that are required for object-oriented database systems that are not present in network models. For example, network models have not considered support for behavioural encapsulation or for any sort of inheritance or type hierarchy.

5.5.1.3 Identity in Temporal Models

Several researchers have argued for a temporal model to support historical versions (e.g.[Clif83], [Khos86]). The reason is that most real-world organisations deal with histories of objects, but they have little support from existing systems to help them in modeling and referencing historical data. Strong support of identity is important for temporal data models, because a single retrieval may involve multiple historical versions of a single object. Such support requires the database system to provide a continuous notion of identity throughout the life of each object, independent of any descriptive data or structure that is user modifiable. This identity is the common thread that ties together these historical versions of an object.

5.5.2 Encapsulation

The early development of the concept of an object can be traced to the programming language Simula where an object was thought of as a self contained program having its own data and procedures. In modern programming languages such as Smalltalk an object contains both a private memory and the set of operations which can be validly applied to the contents of that memory. Encapsulation collects the private data and the set of operations into a single entity. Furthermore, it hides the private memory and the internal procedures from the external world so that the only way of communicating with the object and influencing the state of the memory is by way of external methods. In other words, encapsulation prevents objects from being manipulated except via its defined external operations.

The technical reasons for encapsulation are quite obvious : if one is only interested in behaviour the representation can be ignored, and by divorcing the two the representation may be freely modified and adjusted to factors such as better algorithmic solutions or new hardware technology.

However, prespecifying all operations by a fixed set of methods is a rigid

constraint for evolving databases. In some cases users may want to access the representations (i.e. attributes of tuples).

Implications of encapsulation

Object-oriented systems emphasize object independence by encapsulation of individual objects. Database systems, on the other hand, emphasize data independence by separating the world into two independent parts, namely the data and applications operating on them. The independence boundary is between the database and the rest of the world. The separation serves many purposes, but the most important effect is that the responsibility of the database system is well defined, and consequently the database interface can be relatively simple. Object independence is fundamental to object-oriented systems, just as data independence is fundamental to database systems. If database techniques are to be relevant to object management, we must ask whether these two principles can co-exist?

Consider, for example, the design of a large application. If we believe in data independence then we must split our application into persistent, uninterpreted data, and the programs that manipulate and share them. An object-oriented approach, however, would lead to a design composed entirely of objects. Manipulation of objects is via their interfaces, and sharability is implicit in the message-passing paradigm. It seems clear that the two approaches lead to very different designs.

Relational systems allow many relationships, but they are completely static, based on contents and operations like join. Contents, however, are generally hidden in objects due to encapsulation. Also in the relational model, tuples do not have a visible identifier. They are identified by their contents, via primary or secondary keys. If an object's contents are properly encapsulated, they cannot be expected to provide a means for identification. Databases traditionally provide operations based on selection by contents. This is especially true in relational systems, when all

relationships between entities are represented by contents, and all operations are based on contents. In an object-oriented system an object's contents are typically encapsulated. We are not supposed to know the values of an object's variables. Even when objects advertise visible attributes, we may not know whether they are real, or virtual attributes computed by the object upon request. How can we take advantage of existing indexing mechanisms and content-oriented selection in database systems for object selection?. Since objects encapsulate behaviour, they should also be selectable in terms of their behavioural aspects.

5.5.3 Types/Classes

A type, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. It has two parts: the interface and the implementation. Only the interface part is visible to the users of the type, the implementation of the object is seen only by the type designer. The interface consists of a list of operations together with their signatures (i.e. the type of the input parameters and the type of the result).

The type implementation consists of a data part and an operation part. In the data part, one describes the internal structure of the object's data. Depending on the power of the system, the structure of this data part can be more or less complex. The operation part consists of procedures which implement the operations of the interface part.

In programming languages, types are tools to increase programmer productivity, by ensuring program correctness. By forcing the user to declare the types of the variables and expressions he/she manipulates, the system reasons about the correctness of programs based on this typing information. Types are mainly used at compile time to check the correctness of programs.

The notion of class is different from that of type. Its specification is the same as

that of a type, but it is more of a run-time notion. It contains two aspects: an object factory and an object warehouse [Cox86]. The object factory can be used to create new objects, by performing the operation *new* on the class. The object warehouse means that attached to the class is its extension, i.e. the set of objects that are instances of the class. The user can manipulate the warehouse by applying operations on all elements of the class. Classes are not used for checking the correctness of a program but rather to create and manipulate objects.

According to the principle of classification objects which have the same properties i.e. the same kind of private data and operations, are collected into an object class which corresponds to a specific type. Objects which belong to an object class are called instances of this class. An Object class actually serves two purposes : it functions (a) as an abstract data type for its instances and (b) as the object which represents the set of its instances.

When regarded as an abstract data type (ADT) an object class defines the private structure and the operations of its instances by instance attributes and instance method specifications. In addition it defines a public interface and therefore determines the behaviour of all its instances which can be described together at a higher level. Unlike instance attributes which may take different values in different instances, an ADT also specifies all the attributes which are constant for all instances. These attributes are called class attributes and should be located for economical reasons at the object class.

In addition, an object class will itself be treated as an object that represents the set of all its instances. As such it has its own set-oriented attributes and operations. Treating an object class also as an object leads to a very homogeneous model. Object class objects differ from elementary objects in carrying type information. Or in other words, an object class object additionally has a special kind of property representing the abstract data type information.

5.5.4 Inheritance

If you describe a tiger, you can say it is a big cat with stripes. You do not think of it as individual ears, wiskers, paws and a tail. While these features may be important they are not things which distinguish it from a cat next door. Looking at the world in this sort of way is one of the bases of object-oriented programming. The object-oriented approach to writing software is described as being closer to reality than traditional methods, which work at the kit-of-parts rather than the hierarchy of cats level.

Inheritance has two advantages: it is a powerful modeling tool, because it gives a concise and precise description of the world and it helps in factoring out shared specifications and implementations in applications.

An example will help illustrate the interest in having the system provide an inheritance mechanism. Assume that we have Employees and Students. Each Employee has a name, an age above 18 and a salary, he or she can die, get married and be paid. Each Student has an age, a name and a set of grades, He or she can die, get married and have his or her GPA computed.

In a relational system, the database designer defines a relation for Employee, a relation for Student, writes the code for the *die*, *marry* and *pay* operations on the Employee relation, and writes the code for the *die*, *marry* and *GPA computation* operations for the Student relation. Thus the application programmer writes six programs.

In an object-oriented system, using the inheritance property, we recognize that Employees and Students are Persons; thus, they have something in common (the fact of being a Person), and they also have something specific. We introduce a type person, which has attributes name and age, and we write operations *die* and *marry* for this type. Then, we declare that Employees are a special type of Persons, who inherit attributes and operations, and have a special attribute, salary, and a special

operation *pay*. Similarly, we declare that a Student is a special kind of Person, with a specific set-of-grades attribute and a special operation *GPA computation*. In this case, we have a better structured and more concise description of the schema(we factored out specification) and we have only written four programs (we factored out implementation). Inheritance helps code reusability, because every program is at the level at which the largest number of objects can share it.

The types of inheritance include: substitution inheritance, inclusion inheritance, constraint inheritance and specialization inheritance.

In substitution inheritance, we say that a type t inherits from a type t', if we can perform more operations on objects of type t than on objects of type t'. Thus any place where we can have an object of type t', we can substitute for it an object of type t. This kind of inheritance is based on behaviour and not on values.

Inclusion inheritance corresponds to the notion of classification. It states that t is subtype of t', if every object of type t is also an object of type t'. This type of inheritance is based on structure and not on operations. An example is a square type with methods *get, set(size)* and a type filled-square, with methods *get, set(size),* and *fill(colour)*.

Constraint inheritance is a subcase of inclusion inheritance. A type t is a subtype of a type t', if it consists of all objects of type t which satisfy a given constraint. An example of such inheritance is that a teenager is a subclass of person: teenagers don't have any more fields or operations than persons but they do obey more specific constraints (their age is restricted to be between 13 and 19).

With specialization inheritance, a type t is a subtype of a type t', if objects of type t are objects of type t' which contains more specific information. Examples of such are persons and employees where the information on employees is that of persons together with some extra fields.

Inheritance encourages the development of small, reusable classes that become

building blocks for more sophisticated classes. This approach results in less code to maintain and test and more rapid development from prototype to final application.

There is reason to believe that the use of hierarchies of data abstractions will lead to a system with improved performance over systems using more traditional DBMSs (e.g. relational). Firstly, all the physical aspects of representation of information by data are user-transparent in the object models. This creates greater potential for optimization; more things may be changed for efficiency considerations, without affecting the user programs. The relational model has more data independence than the older models. The object models have even more transparency. Secondly, in the object models, the system knows about the meaning of the user data and about the meaningful connections between such data. This knowledge can be utilized to organize the data so that meaningful operations can be performed faster at the expense of less meaningful or meaningless operations. Traditional DBMSs typically support a simple static data structure such as a table. When they are used in an application, the data from the application must be forced into this structure whether appropriate or not. By using hierarchies of data abstractions, the data can be organized into meaningful groups for the application system. Thus data that is likely to be used together will be stored together and can be retrieved with a single operation.

5.5.5 Overloading & Late Binding

There are cases in which the user wants to use the same name for different operations. Consider, for example, the display operation: it takes an object as input and displays it on the screen. Depending on the type of the object, we want to use different display mechanisms. If the object is a picture, we want it to appear on the screen. If the object is a person, we want some form of a tuple printed. Finally, if the object is a graph, we will want its graphical representation. Consider now the

problem of displaying a set, the type of whose members is unknown at compile time. In an application using a conventional system, we have three operations: display-person, display-bitmap and display-graph. The programmer will test the type of each object in the set and use the corresponding display operation. This forces the programmer to be aware of all the possible types of the objects in the set, to be aware of the associated display operation, and to use it accordingly. We could represent this by the pseudo-code:

```
for x in X do
    begin
        case of type(x)
            person: display(x);
            bitmap: display-bitmap(x);
            graph: display-graph(x);
        end
    end
```

In an object-oriented system, we define the display operation at the object type level. Thus, display has a single name and can be used indifferently on graphs, persons and pictures. However, we redefine the implementation of the operation for each of the types according to the type (i.e. overriding). This results in a single name (display) denoting three different programs (i.e. overloading). To display the set of elements, we simply apply the display operations to each one of them, and let the system pick the appropriate implementation at run-time. We could represent this new approach by the much more concise pseudo code:

```
for x in X do display(x)
```

However, the use of object-dependent binding complicates the analysis of the ultimate behaviour of a method when it is called, and this makes optimization of methods a complex task.

5.5.6 Composite Objects

This is the ability to define new composite objects from previously defined objects in a nested or hierarchical fashion.

Many application domains, including CAD/CAM, Spatial Applications, Office Automation and Knowledge Bases, require direct representation and efficient manipulation of arbitrary complex objects.

Most database data models do not have the capability to directly represent and manipulate composite objects e.g. the relational model. Relational systems impose the first normal constraint. Thus the objects must be mapped onto a collection of flat relations. With this approach much of the inherent semantics of complex object composition is lost.

In the relational model we have tuple and set constructors. Sets are important because they are a natural way of representing collections from real world. Tuples are critical because they are a natural way of representing properties of entities. However, the model lacks the list or array constructor to capture the order, which occurs in the real world, and which also arises in many scientific applications, when people need matrices and time series. The constructors of the relational model are not orthogonal, because the set construct can only be applied to tuples and the tuple construct can only be applied to atomic values. There have been several attempts to address this issue. Some extend existing database systems [XSQL] [Lori83] , others extend relational models by relaxing the first normal form constraint [NF2-relations] [Dada86], other extend semantic models [Damokles][Ditt87]. These approaches will be discussed in *chapter six.*

5.6 Limitation of Object-Oriented Definition

The problem of defining real world entities in all their sufficient and necessary attributes and operations is well known in philosophy. This problem faced Plato when trying to define a table in terms of its necessary and sufficient attributes and he contended with the definition of an ideal table. This problem occurs because of the existence of natural kinds of objects which can not be defined in terms of other things. However, it is practical to define our technical objects in terms of partial views.

Another problem is that objects are configured according to one particular view, which renders sharing of information complicated. Indeed, it can be very hard for another application to deal with an object format that is not adequately structured. The second application would have to recognize, extract, and reformat appropriately the information in such an object base.

Thirdly, queries generally follow predefined links and tend to operate on individual objects. Thus, the processing of queries involoving large and arbitrary sets of data is not well supported.

Chapter 6

Object-Oriented Data Models

Chapter 6

Object-Oriented Data Models

6.1 Introduction

[Ditt86] and [Ditt88] introduce a classification of object-oriented data models: if a model supports complex objects, it is called structurally object-oriented; if extensibility is provided, it is called behaviorally object-oriented; a fully object-oriented model has to offer both features. While the record-oriented approach to data management provides simple objects and generic operations, the object-oriented approach - influenced by database systems and programming languages - provides three variants as illustrated in figure 6.1.

The central notion in structural object-orientation is that of a complex object [Lorie and Plouffe 1983] [Lori83] or of a molecule [Batory and Kim 1985] [Bato85], reflecting the fact that objects in the world of interest are composed of parts that among themselves undergo a variety of other relationships. Some researchers start from the relational model and extend it to impose clustering on the set of tuples. One way in which this is done is by introducing an abstract pointer mechanism for interconnecting the tuples from one or more relations that together make up a cluster object [Lori85]. Another approach avoids the pointers by directly building the tree structure into a relation: in contrast to the standard relational model where attribute values are atomic, these values may themselves be relations. Nesting of relations may thus occur to an arbitrary depth. This model is called NF2 (Non-First-Normal-Form) [Dada86]. Other researchers start from the entity relationship model. From a technical perspective this model could hardly be classified as object-oriented because all it does is to introduce basic building blocks called entities and establish relationships between them. Its success as a method for database design is due to the naturalness with which many real-world phenomena can be

mirrored as entities and relationships. By grouping several entities, and the relationships among them, into a cluster and treating the cluster again as entities one obtains a fairly general structurally object-oriented model. Two examples are the complex entity-relationship model (CERM) [Ditt87], and molecular objects [Bato84].

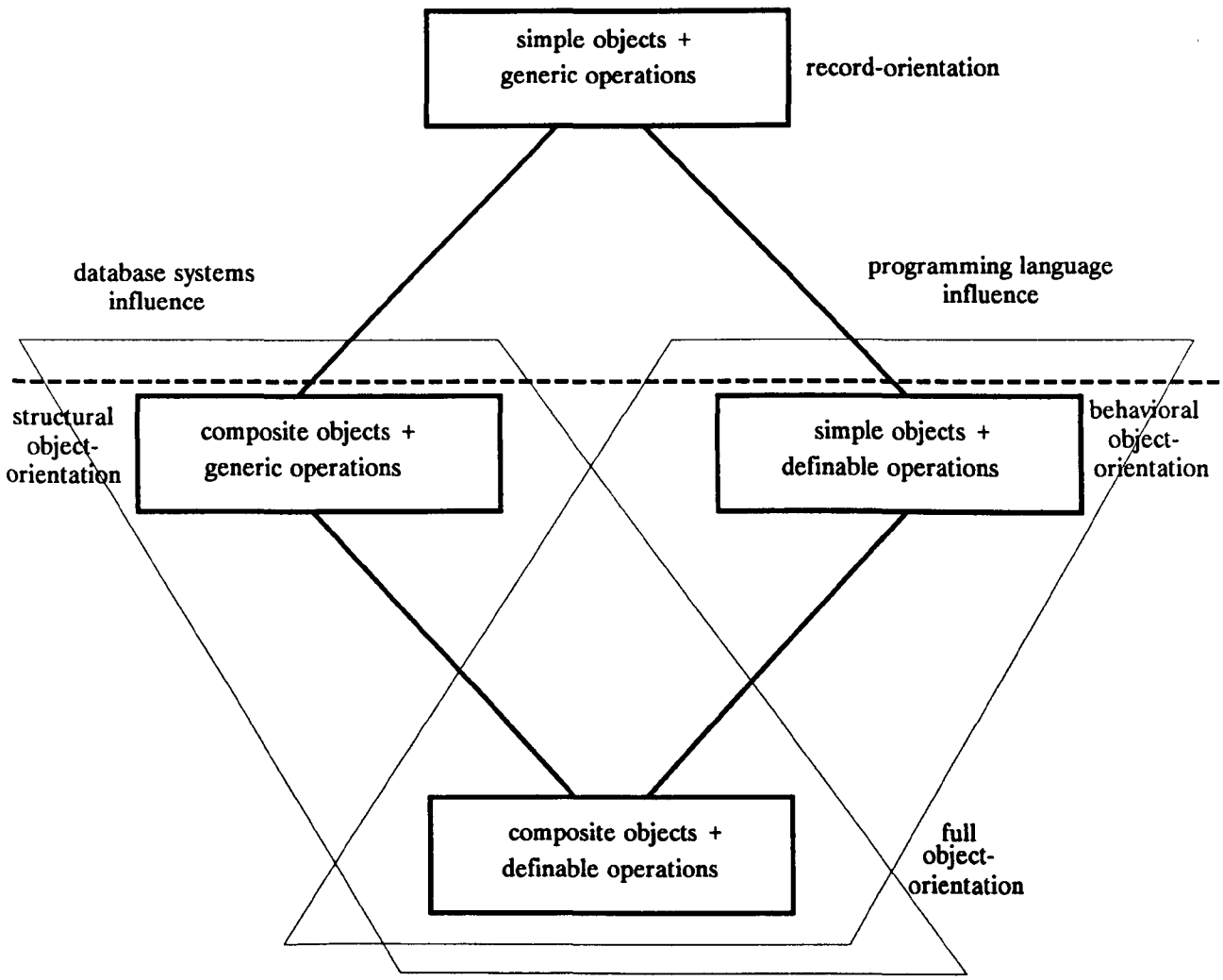


Figure 6.1 Approaches to object-orientend data management

6.2 Structural Object-Orientation

Models exhibiting this form of object-orientation fall into two categories: Those based on data model extensions (described in sections 6.2.1-6.2.2) and those based on systems extensions (section 6.2.3). We will use the cell description in figure 6.2 for comparing the different approaches. Note that CELL3 is composed of four instances of CELL1 (i.e. I1-I4), one instance of CELL2 (i.e. I5), and six paths (i.e. P1-P6). The path P3 in turn is composed of three segments (i.e. S1-S3).

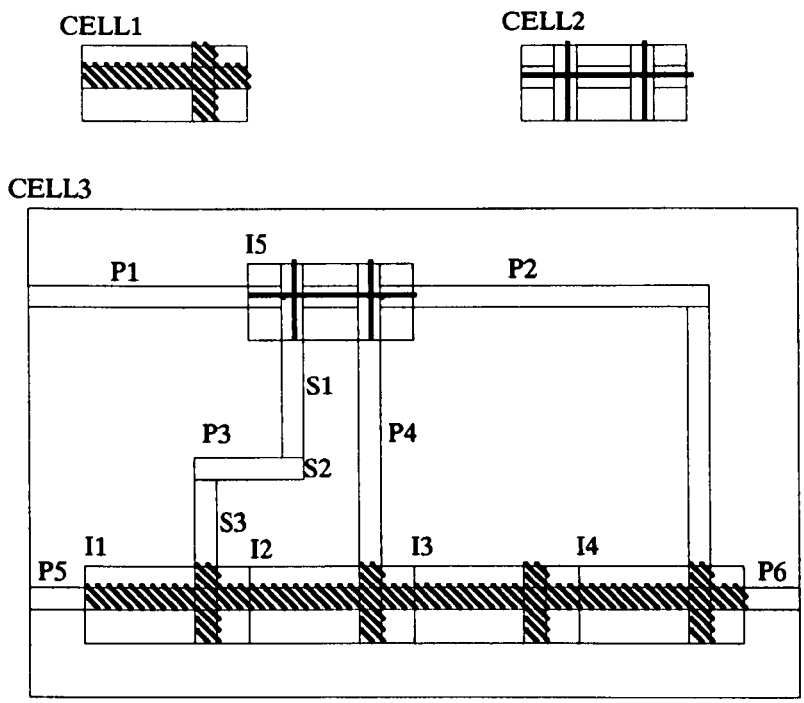


Figure 6.2 Cell description

6.2.1 Relational Model Extensions

Database systems that recognize objects have the advantage of allowing object operations to be specified more easily and executed more efficiently. However, the introduction of objects into the relational model if not done properly will destroy the symmetry, simplicity and elegance of the relational model. Should objects be treated as full members of the relational model with the same status as tuples and relations?

Answering yes implies that a complete set of operations must be provided to manipulate objects. One reason for the popularity of the relational database systems is their guarantee that any information stored in the database can be extracted using a nonprocedural query. This property should not be sacrificed by the introduction of objects.

Answering no implies that relational database objects can treat those objects as a useful luxury. The system will be complete because all the data in a database is stored in tuples and relations, and the traditional relational algebra can be used to manipulate the data. Objects will exist but only as abstract collections of tuples and relations.

The introduction of objects means that data can be organized and stored by relations or objects. When data is organized by relations, global searches of all tuples in a relation are easy to specify and more efficient to execute. Organizing data by objects simplifies specification of operations that address a particular object and improves their efficiency. In the following sections we discuss the different relational model extensions to provide structural object-orientation:

(a) The NF2 Data Model

The NF2 (non-first-normal form) model , as introduced by [Sche86] , is based on the nonnormalized relational model. AIM-P [Dada86] is an implementation of the NF2 model that is being developed at the IBM Scientific Center, Heidelberg. It supports composite attribute types, which can be either tuple valued, that is, one tuple, or relation valued, that is a set of tuples. A composite attribute could also be a list of possibly composite elements. All these structures can be arbitrarily nested. The NF2 data model implicitly incorporates references to tuples of different relations. Thus it is really a hybrid of the relational and the hierarchical data model. Again we note that there is a problem with data redundancy. The extended query language is nontrivial because of its nested nature. This model does not provide for application-specific operations. An example of NF2 representation for figure 6.2 is shown in figure 6.3.

(b) QUEL as a data type

Stonebraker [Ston84] has proposed representing an object as a QUEL query stored in the field of a relation. This query, when executed, retrieves all the data in the object. Thus, the object is stored conceptually in a field, but physically as a set of tuples in one or more separate relations. Figure 6.4 shows the cell relation for the database in figure 6.2 using Quel queries to represent objects.

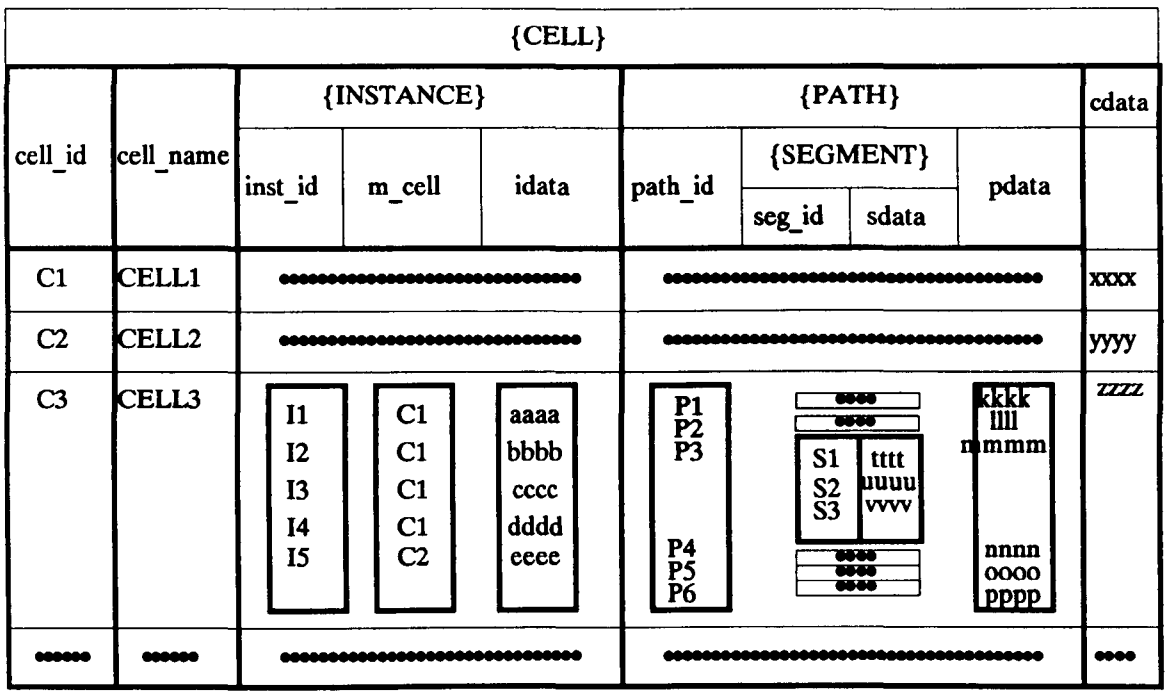


Figure 6.3 NF2 Representation of the CELL data

(c) Relational/Network Model

Haynie's model [Hayn81] allows an attribute of a relation to contain a pointer to another relation. This scheme divides a database into many small relations, each of which contains part or all of the data in an object. For example figure 6.2 would be changed as shown in figure 6.5.

CELL			
cell_name	cdata	INSTANCE	PATH
CELL1	xxxx	@10	@20
CELL2	yyyy	@30	@40
CELL3	zzzz	@50	@60

Figure 6.5 The CELL relation using the Relational/Network Model

CELL

cell_id	cell_name	instances	paths	cdata
C1	CELL1			xxxx
C2	CELL2			yyyy
C3	CELL3	retrieve (INSTANCE.all) where cell = "C3"	retrieve (PATH.all) where cell = "C3"	zzzz

INSTANCE

inst_id	cell	m_cell	idata
I1	C3	C1	aaaa
I2	C3	C1	bbbb
I3	C3	C1	cccc
I4	C3	C1	dddd
I5	C3	C2	eeee

SEGMENT

seg_id	path	seg#	cdata
S1	P3	1	tttt
S2	P3	2	uuuu
S3	P3	3	vvvv

PATH

path_id	cell	segments	pdata
P1	C3	retrieve (SEGMENT.all) where path = "P3"	kkkk
P2	C3		llll
P3	C3		mmmm
P4	C3		nnnn
P5	C3		oooo
P6	C3		pppp

Figure 6.4 The CELL relation using QUEL/POSTQUEL as a data type

(d) ADT-Ingres

ADT-Ingres was proposed by Stonebraker et al. [Ston83a] and implemented as an experimental prototype on top of existing DBMS Ingres. In this proposal the object is represented in a field of a tuple as an abstract data type supplied by a conventional programming language. Operations for the type are implemented as procedures within the programming language and can be called by the DBMS.

ADT-Ingres provides a novel way of specifying new data types and corresponding operators in a database management system and these operators can be arbitrarily complex. However, the penalty of this flexibility is that the user has to be familiar with two quite different systems (1) the database language Quel and (2) the programming language C. Another limitation of this approach is that the internal representation of ADT must fit in the length of the field provided by Ingres DBMS. In this approach each ADT is mapped completely into an attribute and the internal representation of the object does not reflect the external structure of the object. Furthermore, it lacks support for the hierarchical data structure that occurs frequently in object-intensive applications. ADT-Ingres provides some facilities for behavioral object-orientation by allowing the database users to define application specific ADT operations. However, these operations are quite tedious to implement because the model is not structurally object-oriented. Figure 6.6 shows the ADT-Ingres representation for the cell data.

CELL			
cell_name	cell_data	INSTANCE	PATH
CELL1	xxxx	*ADT*	*ADT*
CELL2	yyyy	*ADT*	*ADT*
CELL3	zzzz	*ADT*	*ADT*

Figure 6.6 The CELL database using abstract data types

In this approach the hierarchy of ADT is not considered and the inheritance of common operations and use of overloading concepts are not considered either.

6.2.2 ER Model Extensions

(a) Complex-entity-relationship model (CERM)

This model has been realized in the DAMOKLES database system prototype [Ditt87]. DAMOKLES has been developed as the basic information management component for a software engineering environment. CERM is an extended entity-relationship data model which, since it is no longer just a semantic framework, includes a full-fledged set of operators. The main structural features of CERM include the following:

- molecular entities (complex objects) which may be built recursively and which may also overlap (i.e a subentity is part of two or more superentities)(note figure 6.7.a),
- versions of structured objects,
- arbitrary n-ary relationships between objects of any structure or versions. Figures 6.7.b and 6.7.c show alternative models for the cell description using CERM.

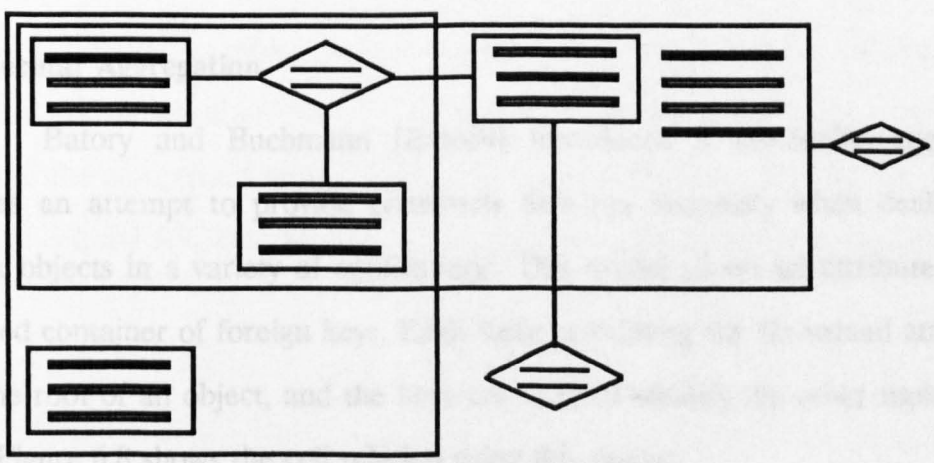


Figure 6.7.a

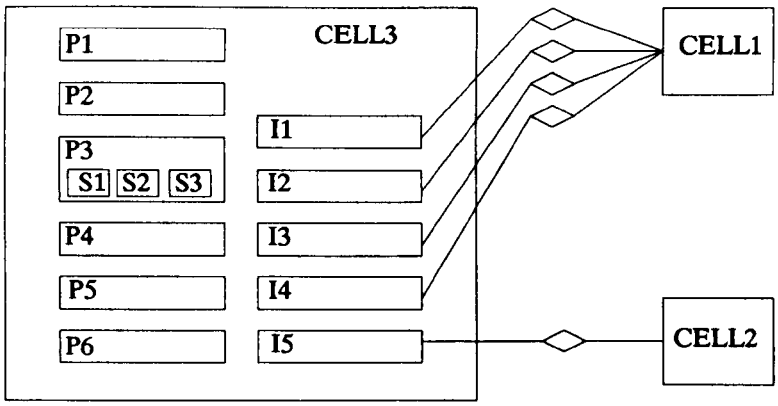


Figure 6.7.b

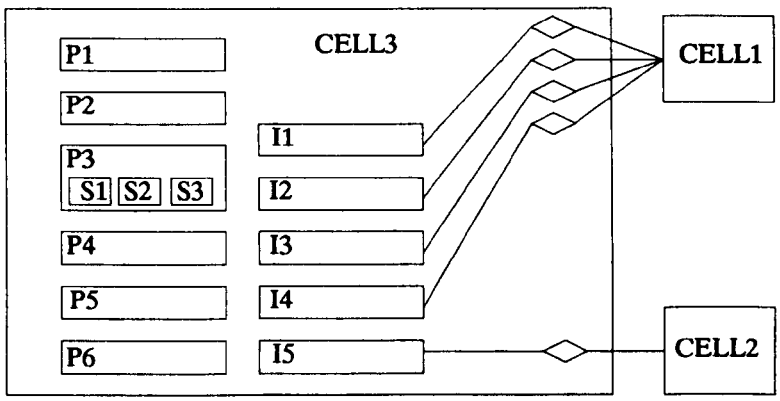


Figure 6.7.c

(b) Molecular Aggregation

Batory and Buchmann [Bato84] introduced a molecular aggregation model as an attempt to provide constructs that are necessary when dealing with complex objects in a variety of applications. This model allows an attribute to be a set-valued container of foreign keys. Each tuple containing the set-valued attribute is made the root of an object, and the keys are used to identify the other tuples in the object. Figure 6.8 shows the cell relation using this model.

CELL			
cell_name	cell_data	INSTANCE	PATH
CELL1	xxxx	••••••••••	••••••••••
CELL2	yyyy	••••••••••	••••••••••
CELL3	zzzz	{@123000301, @123000302, @123000303, @123000304, @123000305}	{@123000306, @123000307, @123000308, @123000309, @123000310, @123000311}

Figure 6.8 CELL objects using Molecular Object Model

6.2.3 System Extensions

(a) The Complex Object Model (System R extension)

This is one of the extensions to enhance System R [Astr76] to support technical applications. In Lorie's proposal [Lori83] a complex object is a hierarchical cluster of tuples of different relations logically connected together. It corresponds to a 1:N relationship. The hierarchical relationship between tuples is expressed by attributes of type "component-of". General N:M relationships are expressed by attributes of type "reference". The main difference between the two reference types is that the data model provides built-in support to access all tuples belonging to a component-of relationship by physically clustering the data and manipulating pointers to the component tuples. The association of tuples is achieved via surrogate attributes. Figure 6.9 illustrates the Complex Object Model representation for the cell data.

CELL			
PID	TID	cell_name	cdata
@000000000	@123000100	CELL1	xxxx
@000000000	@123000200	CELL2	yyyy
@000000000	@123000300	CELL3	zzzz

INSTANCE			
PID	TID	m_cell	idata
@123000300	@123000301	@123000100	aaaa
@123000300	@123000302	@123000100	bbbb
@123000300	@123000303	@123000100	cccc
@123000300	@123000304	@123000100	dddd
@123000300	@123000305	@123000200	eeee

PATH		
PID	TID	pdata
@123000300	@123000306	kkkk
@123000300	@123000307	llll
@123000300	@123000308	mmmm
@123000300	@123000309	nnnn
@123000300	@123000310	oooo
@123000300	@123000311	pppp

SEGMENT		
PID	TID	sdata
@123000308	@123000312	qqqq
@123000308	@123000313	rrrr
@123000308	@123000314	ssss

Figure 6.9 Complex Object Model description of the CELL

6.3 Behavioral Object-Orientation

Behavioral object-orientation of database systems has mostly originated from the programming area, with an attempt to extend programming languages with persistency. Examples of this category include *O2*[Banc88][Lecl88][Lecl89], *CACTIS*[Huds86][Huds87], *Trellis/Owl*[O'Br86], *OOPS*[Schl88]. Early approaches stuck closely to Smalltalk and hence lack the clustering concepts; one such development is Gemstone [Cope84].

6.3.1 OOPL Model Extensions

(a) GemStone

The OPAL language that constitutes both the data definition and data manipulation of GemStone is very similar to Smalltalk. OPAL introduces a selection block construct for special types of objects called constrained collection. Any instance of Collection is a collection of objects such as a set or a bag, or a dictionary. A collection that is a set is similar to a table in a more conventional database query language such as SQL. However, one immediate distinction between a language such as SQL and a novel untyped language such as Smalltalk is that the former specifies a particular type for each of its columns that corresponds to the instance variables of the elements of the collection in Smalltalk. In Smalltalk, corresponding instance variables can assume different types and the same instance variable can be updated to indicate a different type of object. SQL implementations make heavy use of the type constraint in storage organisation, query optimization, and indexing. OPAL introduces constrained collections for similar reasons. OPAL violates encapsulation in two respects (1) accessing instance variables directly through dot notation, (2) specification of indexes explicitly on instance variables. More recent approaches adopt many of the ideas from Smalltalk but combine these with some clustering facilities [Banc88],[Bane87a]. However, clustering is essentially restricted to assembling objects into sets so that much of the expressiveness provided by relationships is still missing.

6.4 Full Object-Orientation

This approach combines the merits of both behavioral and structural object-orientation. Some extend the structural model with behavioral features [Kemp87], others extend the behavioral model with structural features [Bane87a]. However, some extend classical models with both structural and behavioral features [Ston86c].

6.4.1 Relational Model Extensions

(a) POSTGRES Data Model

This data model was proposed by Stonebraker and Rowe [Ston86c][Rowe87] as a successor of Ingres. The goals of the POSTGRES project are the following :- (1) provide support for complex objects, time varying data, (2) user extensibility for data types (3) facilities for active databases (i.e. alerters, triggers and rules), (4) make as few changes to the relational model as possible. The POSTGRES data model is a relational model extended to support semantic modelling constructs:- (1) abstract data types [Ston86d] (2) data of type procedure [Ston87] (3) rules [Ston88]. Features of the POSTGRES Data Model include the following:- (1) support for primary keys (2) inheritance of data and procedures (3) attributes that reference tuples in other relations.

POSTQUEL is the POSTGRES query language and differs from Quel in :- (1) from-clause to define tuple variables, (2) relation-valued expressions may appear any place that a relation could appear in Quel, (3) transitive closure and execute commands have been added, (4) set operators have been included.

An important aspect is **complex object support.** An ADT facility proposed by Stonebraker [Ston83] meets the needs of a variety of object management applications. However, it fails in three important situations: (1) objects with many levels of subobjects (2) objects with unpredictable composition (3) objects with shared

subobjects.

Consider an application which stores data about a particular building in a database. An object in such a database might be an office desk. However, the desk is in turn constructed of subobjects (drawers), which are in turn constructed of subobjects (e.g. handles). This "part-of" hierarchy is prevalent in many non-traditional applications. A user often wishes to "open up" an object and access specific subobjects. For example, he might want to find the handle on the lower left-hand drawer. The ADT proposal would force a user to write an operation for each such access he wanted to perform. A very large number of operations would result that would be exceedingly hard to use. In summary, a user wants the query language to assist with "opening up" complex objects and searching for qualifying subobjects; he does not want an operator for each particular search.

The second problem concerns the unpredictable composition of objects. Suppose the database contains objects that are on top of desks in the example building. In particular, some desks have flowers, some have phone sets. In this case a subobject of a desk may be one or more objects from a large set of possible desk accessories. It is unreasonable to require a user to write an operation to extract any object from such unpredictable collections.

The third problem concerns shared subobjects. Consider a heating duct in the building that is accessible from several rooms in the building. One would want to store the duct once, and then have it as a shared subobject in higher level objects (rooms). The ADT proposal has no ability to share subobjects in this fashion.

The Postgres proposal for storing Postquel as a field in the relation solves these problems but has its limitations as discussed in the data model comparison section (See section 6.4).

(b) R2D2 (Relational Robotics Database System with Extensible Data Types)

R2D2 [Kemp87] lies in the category of full object-oriented database systems. This is achieved by integrating the concept of abstract data types into the data definition and manipulation language of a structurally object-oriented DBMS. Thus the database user can define data types that correspond to application-specific objects. The structural basis for R2D2 is formed by an extended NF2 data model, which allows nested relations whereby hierarchical relationships among subobjects can be modeled.

6.4.2 Functional Model Extensions

(a) IRIS

The IRIS [Fish86] data model is based on a semantic data model that supports ADT. It is based on the DAPLEX data model. It has much in common with the PROBE data model (PDM)[Mano86] (see subsection (b) below). The IRIS data model contains the main construct objects, types, and functions. Objects in IRIS are used to represent entities and concepts. Types have unique names and are used to categorize objects into sets that are capable of participating in specific functions. Objects serve as arguments to functions and may be returned as results of functions. A function may only be applied to objects that have the types required by the function. Types are organized in a cyclic type graph that represents generalization and specialization. A type may be declared as a subtype of another type. Attributes of objects, relationships among objects, and computation on objects are expressed in terms of functions. Unlike mathematical functions IRIS functions may have side effects. The IRIS model clearly separates the concepts of objects, types and functions. This is reflected in the following: (1) objects may acquire and lose types. (2) objects of a given type are not required to participate in every function defined on

that type. (3) functions over a given type may be created and destroyed at any time. This results in schema evolution without affecting existing applications.

(b) PDM

PROBE Data Model (PDM)[Mano86] is an extension to the DAPLEX data model. It incorporates multiargument functions and computed functions into the model, and extensions to the model to deal with objects having spatial and temporal semantics. As we have pointed out in section 4.5.3 the functional model has many features associated with object-oriented data models. These include (1) the concept of an entity or object that has existence independent of any properties or relationships with other entities. (2) objects and set-valued properties allow the complex objects to be modeled. (3) an entity class or type concept together with generalization hierarchy. (4) functions provide a method for incorporating behaviour and derived properties. The cell description using the functional approach is illustrated in figure 6.10.

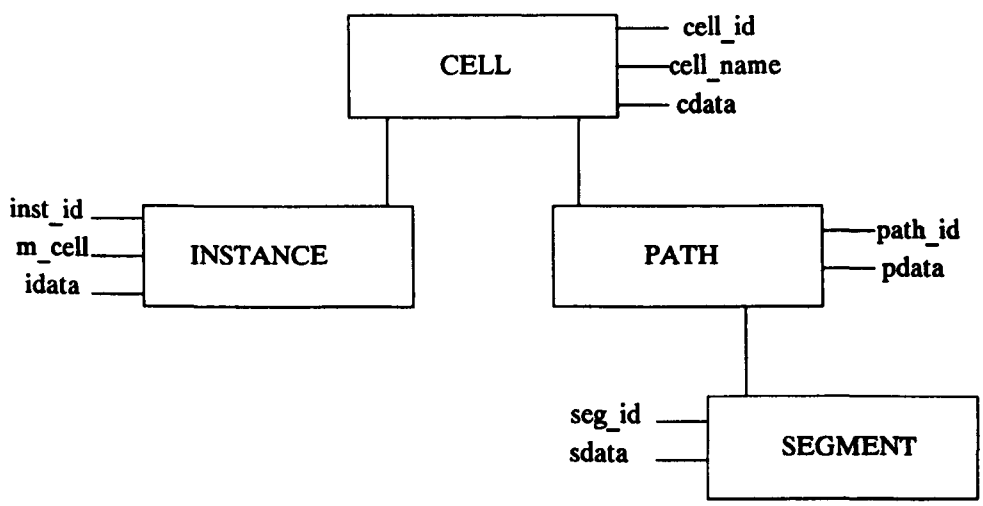


Figure 6.10 Cell description

6.4.3 OOPL Model Extensions

(a) ORION

ORION [Bane87a] extends the Smalltalk concept by adding composite objects. In ORION, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft. Following the paradigm of encapsulation, an object consists of some private memory of instance variables that holds its state , and a set of methods that manipulate or return the state of an object. The domain of an instance variable is a class. Methods are part of the definition of the object. Objects communicate with one another through messages. Similar objects are grouped together into a class, and classes are organized into class hierarchies. Inheritance involves the instance variables and methods specified for a class. In ORION a class can have more than one superclass, generalizing the class hierarchy to a lattice. For clustering, ORION introduces the notion of composite object that captures the is-part-of relationship between an object and objects it references. A composite object has a single root object, and the root references multiple children objects each through an instance variable. Each child object can in turn reference their own children objects, again through instance variables. A parent object exclusively owns children objects, and as such the existence of children objects depends on the existence of their parent. However, the model and implementation of composite objects as presented in [Kim87] and implemented in ORION, suffer from the following shortcomings:

First, the model restricts a composite object to a strict hierarchy of exclusive component objects; that is a component object is only part of one composite object. This is certainly the right model for a physical part hierarchy, in which an object can not be part of more than one object. However, this is not acceptable for a logical part hierarchy; for example, an identical diagram or illustration may be a part of two

different books.

Second, the model forces a top-down creation of a composite object; that is, before a component object may be created, its parent object must already exist. This prevents a bottom-up creation of objects by assembling already existing objects.

Third, the model requires that the existence of a component object depends on the existence of the parent object; that is, if an object ceases to exist, all its component objects are also deleted. This feature is sometimes desirable, since it frees the applications from having to search and delete all nested components of a deleted object. Sometimes, however, it impedes reuse of objects in complex design environments.

6.5 Comparison of the Data Models

We use the following metrics to compare the different data models. We have not tried to identify all possible metrics, but have included those that we feel are important and that allow direct comparison between the data models. The metrics fall into four classes: (1) implementation metrics (2) semantics metrics (3) representation metrics, and (4) structure metrics

6.5.1 Implementation metrics

The implementor of a relational database system for objects should be able to choose whether to cluster tuples into objects or relations. Organisation by relation will make relation-oriented operations more efficient, while organisation by object will make object-oriented operations more efficient. Both organisations are possible for the complex object model, molecular aggregation. A by-relation organisation of the complex model is trivial. A by-relation organisation of the molecular aggregation object model can be made using link relations to replace the set-valued attributes in the root tuple of an object. A by-object organisation of both models is possible in a database system such as System R that allows tuples of one relation to be stored with the tuples of another relation.

A by-relation implementation of Haynie's relational/network model [Hayn81] is possible through collecting similar small relations into one large relation using an additional attribute to identify the small relation to which each tuple belongs. A by-object implementation of the QUEL/POSTQUEL as a data type model [Ston84]/ [Ston86c], is more difficult because the component tuples of an object can only be identified by executing the query that defines the object. Thus, when a new tuple is inserted into a database these queries may have to be executed to find the object that owns the new tuple.

The object-in-a-field or ADT scheme described by Stonebraker and R2D2 uses

abstract data types from a programming language to represent an object. Since the data in an object must be accessed via this programming language, it must be clustered and stored in that object, making a by-relation organisation difficult to implement.

In the NF2 scheme the data in an object is described by a tree of tuples. A by-relation organisation of this model must provide access to each of the tuples via their relation and via the object. This organisation is difficult to implement efficiently. A by-object organisation of this model, by contrast, needs only to provide access to the tuples in an object via the tuple containing that object in one of its fields. Hence, a by-object organisation is much easier to implement. Databases designed round the Smalltalk paradigm like GemStone, Orion use by-object organisation. The functional model extensions like PDM, IRIS can be implemented as traditional databases using binary relation or CODASYL style sets. However, unless a distinction is made between functions that connect data in the same object and functions that connect independent objects in relationships, this scheme is difficult to implement as a database organized into objects. For example, in the functional model the department of the designer of a cell might be accessed using an expression such as cell.designer.department. In this expression, department and cell could be disjoint objects but the notation does not show that a boundary of any object has been crossed. Therefore, unless there is a method for making this distinction, the system will not know that department values should not be clustered around cell objects.

6.5.2 Semantics metrics

At least three types of semantic information can be identified: application semantics deals with the meaning of the data as used by the application programs, structural semantics deals with the integrity of the data structure used to implement the object, and operational semantics identify the valid operations that may be applied to an object.

Usually application semantics is handled by the application programs that use the database. As a result, important considerations for application semantics are the interface between application programs and the database systems, and the work required to retrieve an object from the database. As discussed in Zaniolo [Zani85], there is often an impedance mismatch between programming languages that deal with one record at a time and an object database that deals with an object (multiple tuples) at a time. Sometimes objects must be retrieved tuple by tuple, degrading performance and making the database system hard to use. The abstract data type approaches of Stonebraker and R2D2, and Smalltalk extensions like Orion, GemStone are best at avoiding these problems because they are extensions of programming languages that include the concept of an object. The Complex Object, Relational/Network, Molecular Aggregation schemes are poor. They offer very little more than the traditional tuple-at-a-time interface of a relational database system. With the NF2 scheme an object can be retrieved at once, but a special programming language is required to manipulate the object.

Structural semantics is included to varying degrees in all the data models. The ADT approach uses it perhaps the most since a user can not directly modify the structure of an object once created and stored in a field. Smalltalk extensions like Orion, GemStone are also good; the operation for creating and modifying an object can include code to verify that the integrity of the object structure is maintained. The functional model extensions like PDM, IRIS appear to be the weakest since

they do not clearly define the boundary, and hence the structure of an object. Therefore, it may be possible to connect two values and form an illegal object.

The data models vary in the degree to which they allow expression of operational semantics. All of the models allow operations for objects to be expressed. However, transaction modeling is needed to handle complex precedence relationships between operations and it is not discussed in these models except in Orion and PDM.

6.5.3 Clarity of representation

This concerns the ability of a data model to precisely define objects and relationships between them. These metrics measure the ease with which a user can determine the meaning of the object in the different models.

Some mechanism must be used to represent arbitrary relationships between objects in a database. In addition, a mechanism is needed to represent the connection between a root tuple of an object and its components. If the same mechanism is used to do both things, then it may no longer be clear which connection represents relationships and which represents the components of an object. In this case the boundary of the object may not be clear.

In a complex object database, the root of an object is connected to its components using tuple identifiers (TIDs). If the TID of a tuple is its only key, a relationship between this tuple and another tuple must be modeled by storing the TID of this tuple in the other tuple. Hence the distinction between component objects and other relationships to tuples in other objects may be weak. To control integrity, Lorie has recommended that the TIDs used to link the tuples in an object be hidden from the user. Hence, the user will have to give each tuple another key. This eliminates the potential for confusion, but the system must now provide the user with a function for finding the tuples within an object, implying that a complex object

database must contain operations that are not in the basic relational algebra.

The object-in-a-field models (ADT-Ingres, R2D2) connect tuples in the same object by placing them in the same abstraction. This technique does not use foreign keys, so there is no potential for confusion. The relational/network model makes the boundary of an object clear, because all the data in a small relation belongs exclusively to an object. The Quel/Postquel-as-a-data-type model provides the fuzziest connection, because a Quel query must be executed to find the tuples in an object. The functional model extensions (PDM, IRIS) are weak in distinguishing the boundary of each object, because they make no distinctions between connections that link objects to attributes and connections that link objects to objects. Smalltalk paradigm extensions (GemStone, Orion) arrange data into objects. Therefore, the boundary of an object is clear if a single object is used to represent each application object. In practice, however, the Smalltalk applications often represent large objects as a network of smaller objects, and then the boundary of the application object is less clear.

6.5.4 Object structure

(a) Generalization/Aggregation

Complex object model, NF2, molecular aggregation, and relational/network models have only one abstraction mechanism. Therefore, the distinction between an abstraction that is an aggregation and an abstraction that is generalization may sometimes be fuzzy. On the other hand, a single definition can be used to represent an abstraction that has both aggregate and generic qualities, meaning that a data structure may be completed with fewer definitions.

In functional model extensions (IRIS, PDM) a subtyping mechanism exists for defining generalization hierarchies. Therefore, the two types of abstractions can be

distinguished if functional connections are used to model aggregation only and subtypes are used to model generalizations.

ADT-Ingres and R2D2 abstract data type schemes use abstract data types from programming languages to represent objects, so they can represent generalization using a variant data structure.

Smalltalk extensions (PDM, IRIS) allow generalization to be defined using object type hierarchy. Each component of the generalization is defined as a specialization of the more generic object. An attractive feature of this approach is that the properties and operations of the generic object are inherited by the specialization unless they are overridden by a new property or operation. Aggregation is defined in the Smalltalk paradigm by using the instance variables within an object to reference component objects. The Smalltalk paradigm makes a clear distinction between the two types of abstraction while allowing both to be mixed in the definition of one object.

(b) Disjoint/Overlap

In all the schemes, overlapping objects can be modeled by creating separate subobjects for the parts of the objects that overlap. The overlapping objects can then refer to these subobjects using foreign keys. In the complex object model, the foreign keys are the TIDs of the objects. All the models except the ADT in a field approach are also able to represent overlapping objects using the same technique used to represent a disjoint object. For example, a component tuple in a molecular object could be linked to two root tuples.

(c) **Flexibility**

Flexibility to adapt to changing requirements is an important feature for data models. All data models allow some flexibility. The ADT-in-a-field is relatively inflexible with respect to alterations. If an aggregate domain is expanded to include new attributes, all instances of data items in that domain must be found and expanded. The functional model may be the most flexible; new data and relationships can be added without affecting the existing sets and relationships.

Chapter 7

DESIGN AND IMPLEMENTATION OF OQUEL

7.1 OVERVIEW OF OQUEL

7.2 TYPES IN OQUEL

7.2.1 Philosophies of Type

7.2.2 Primitive Components of OQUEL Model

Objects

Classes

Methods

Inheritance

Relationships

7.3 TRANSFORMATION OF AN OBJECT SCHEMA TO A RELATIONAL SCHEMA

7.3.1 Identification

7.3.2 Classification

7.3.3 Generalization/Specialization

7.3.4 Association

7.3.5 Aggregation

7.3.6 Methods and the Behaviour of Objects

7.1 OVERVIEW OF OQUEL

Neither the relational nor the O-O approach is best for all applications. For example, a design object is seen by an engineer in an object-oriented fashion while it is viewed by an accounting department in a traditional relational form. A DBMS is needed which provides several styles for describing and manipulating data. We propose a model which combines the features of OOPL, semantic data models and the relational data model. Since it extends INGRES/QUEL with object-oriented features we called it OQUEL. OQUEL has the following features:-

* It extends the data structures and operations of the relational data model and provides the desirable features of the OOPL paradigm and semantic data models such as improved semantics, data abstractions,reusability of data structure and code, extensibility, complex object support, schema evolution and ADT domains.

* OQUEL introduces the concept of an object identifier (OID) to improve the semantics and reduce the space required to store the database. OID captures the uniqueness of entities in the real world and allows for modeling complex objects. An OID may be used to refer to an object instead of copying it.

* OQUEL extends the relational model by inheritance to allow for sharing of data structures and operations and this improves the productivity of the programmer.

* OQUEL provides two access modes for the database. One is through the relational interface which helps to formulate unpredictable queries and provide flexibility in the selection of a target list through different combinations of attributes - this is due to the no-information-goal-dependency of the relational formalism. The other is a method-based interface which improves reliability by providing semantic integrity, managing complex objects and propagating updates through different semantic references, providing information hiding and operations for naive users.

* OQUEL provides structuring for complex objects through specialized

attributes and provides operations through methods to enforce the integrity of the complex objects. Abstracting complex objects through tuples and relations provides for organisation of data by object or relation.

* OQUEL extends the domain of the relational model by making provision for abstract data types (ADTs) which simplify queries and provide a more natural interface for spatial and temporal data.

* OQUEL provides operations to manipulate the evolution of structure as well as content.

* We have also provided an interface between C++ and Ingres to provide freedom for the implementation of any required methods.

* We can summarize by saying that OQUEL incorporates the concepts of object identity, complex objects, data abstraction/encapsulation, types/classes, inheritance, extensibility, overloading, computational completeness, schema evolution and ADT domains. We have not dealt with concepts such as multiple inheritance, upward inheritance, and complex object versions due to time constraints. The interface between C++ and Ingres, however, is an additional new feature of the implementation.

7.2 TYPES IN OQUEL

7.2.1 PHILOSOPHIES OF TYPE

Programming languages, database systems, and artificial intelligence systems all have the notion that entities can be classified into types. As might be expected, however, the usage of the notion of types is not the same through or even within these areas. The way in which entities are classified into types depends on the purpose of the type classification. In a broad sense, all entities belonging to a type share a common set of properties. In particular, the set of properties that can be used to distinguish types is restricted (a) to fit the purpose for which types were introduced and (b) to allow simple testing for membership in a type.

Types in programming are generally used and thought of as a means of characterizing values that arise dynamically in the course of computation. For instance, a value that is to be computed by a program may be represented by a name or an expression, and although the particular value to which this expression refers may not be known in advance, other information concerning the value might be available. This information could be an indication of the meaning of the expression or it might be a constraint describing a property that the value must have.

The algebraic approach to types is set oriented; in an algebra, a type is a set of individual elements upon which operations of the algebra are defined. A typing system can help support efficiency objectives, provide a language framework capable of guiding a system designer's conceptualizations, and verify the descriptive information provided explicitly and implicitly by a program.

The concept of type plays different roles and is perceived differently by different people. The following is collection of some definitions :-

* System evolution (object-oriented) view :

Types are behaviour specifications that may be composed and incrementally

modified to form new behaviour specifications. Inheritance is a mechanism for incremental behaviour modification.

* System programming (security) view :

Types provide a filter for the interpretation of raw data. They are a suit of clothes that protects raw information from unintended interpretations.

* Type checking view :

Types impose syntactic constraints on expressions so that operators and operand are compatible.

* Type inference view :

A type system is a set of rules for associating with every subexpression a unique most general type which reflects the set of all possible contexts in which the subexpression may be interpreted.

* Application programmers (classification) view :

Types organize values into classes with common attributes and operations.

* Verification view :

Types determine behavioural invariants that instances of the type are required to satisfy.

* Implementation view :

Types specify a storage mapping for values.

Despite the apparent variety, the above mentioned views are in fact complementary since they view the same thing at different levels of abstraction. The purpose of a data abstraction is to model a "thing" without specifying more of its structure and properties than should be externally visible. A "thing" can be viewed as having two aspects: (a) an identity which is invariant over time and which allows one to investigate the same thing at different points in time and (b) a substance which may be investigated. Substance includes both content and structure. Typically the means

of investigating the substance of a "thing" is to perform some applicable operation on the "thing" or to send it one or more messages and to interpret the responses that are returned.

The solution to the overspecification implicit in data models representation (e.g. relational) is to augment the chosen representation with a set of integrity constraints. The notion of constraint is very much the same as the notion of invariants in the implementation of data abstractions in programming languages. Both constraints and invariants define a subspace of the modeling space which is sufficient to represent the entities being modeled. Transaction is the analogue of the data abstraction operation. Given this similarity, it makes sense to encapsulate the representation in the data model with a set of operations that define data abstraction whenever possible.

To preserve the capability of handling unanticipated queries, this encapsulation need not be total. To ensure the constraints are satisfied, it is only necessary that all operations that change the state of the representation be part of the encapsulation. It is possible and appropriate to let some or all of the standard data model operations for querying show through the encapsulation in a semi-transparent way. Semi-transparency means that some of the operations applicable to the representation of a data abstraction are exported directly as operations applicable to the data abstraction itself.

Both objects and types provide a uniform framework within which to understand and manipulate entities.

7.2.2 PRIMITIVE COMPONENTS OF THE OQUEL MODEL

Objects

Objects represent entities and concepts from the application domain being modeled. They are unique entities in the database with their own identity and existence, and they can be referred to regardless of their attribute values. Each object has an assigned, system-wide, unique object identifier, or OID. This supports referential integrity and is a major advantage over record-oriented data models in which the objects, represented as records, can be referred to only in terms of their attributes. The creation of objects is considered in sections 7.5 and 7.8.

Classes

The process of classification involves classifying similar objects into object classes. We can now describe the classes rather than the individual objects themselves. Beside the function of a class to define structure and behaviour of objects, the following property is important : a class represents the set of objects of a type (the instances) stored in the database. Instantiation is the inverse of classification. An object instance is related to its object class by a relationship that may by called "instance-of" relationship. Classes in OQUEL are discussed in sections 7.4 and 7.5.

Methods

The behaviour of an object is encapsulated in methods. Methods consist of code that manipulate or return the state of an object. Methods are part of the definition of the object. However, methods and attributes, are not visible from outside the object. Objects can communicate with one another through messages. Messages constitute the public interface of an object. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a

message by executing the corresponding method, and returning an object. (See Section 7.6).

Inheritance

Objects are classified into classes as first order classification and classes are classified into subclasses as second order classification. Objects in subclasses inherit attributes and methods from their superclasses, but may have additional attributes and methods (horizontal inheritance) or specialized attributes and methods (vertical inheritance). Inheritance facilitates incremental definition and reusability of code. Subtypes (is-a) are defined such that if

B subtype of A,

instances of B are also instances of A. (See Section 7.4)

Relationships

In addition to the classification, instantiation, and generalization mentioned above, there are other important relationships such as aggregation and association. Aggregation is an abstraction concept for building composite objects from their components objects. This relationship can be called is-part-of. An association relates two or more independent objects. An association may have one or more attributes and methods. This relationship can be called is-associated-with. (See Section 7.8).

7.3 TRANSFORMATION OF AN OBJECT SCHEMA TO A RELATIONAL SCHEMA

In this section we will discuss the following abstraction concepts which are used to form an OQUEL schema, their mapping to the relational schema and the necessary relational structures to help the mapping. These abstraction concepts include :-

7.3.1 Identification

7.3.2 Classification

7.3.3 Generalization

7.3.4 Association

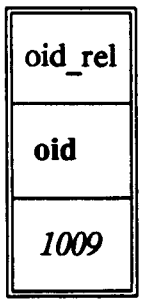
7.3.5 Aggregation

7.3.6 Methods

7.3.1 Identification

This is different from the other abstractions in that, while the other concepts of aggregation, generalization, classification and association are used to build a schema by grouping the various object classes, the concept of identification is used to uniquely identify the object structures that have been formed. In order to enable the user to uniquely identify an instance of a class, the primary key concept of the relational model is supported, i.e. an attribute whose values uniquely identify the instances of the class is designated as the users object's identifier. However, internal to the system, an OID is generated by the system for every object to maintain its unique identity globally. An object can play different roles and therefore can appear as an instance of many classes. However it will be supplied with a unique object identifier (OID) when it is first introduced to the system. This object identifier allows objects to be shared, and association among objects can be modeled by relating the corresponding objects. The following relation, 'oid_rel', was used to maintain

the object identification. It has one attribute which holds the current value of the last generated object identifier.



7.3.2 Classification

Found in almost every existing data or knowledge representation, classification is the most important and the best understood form of abstraction. It is achieved by grouping objects that have common properties into a new object for which uniform conditions hold. Describing a model in terms of specific factual information is however hardly satisfactory. Frequently it might be important to abstract the details of each object and to treat them in a more generic form, without having to worry about the specific values of each property. Classification provides an important means for introducing such generic information by allowing the modeler to refer to the class as a representative or prototype of its instances.

Classification is based on the notion of a set. Including this concept explicitly in data modeling forces a distinction to be made between individual instances and a class of those instances. Therefore, if there are certain properties that are applicable to the group as a whole, they can be modeled as attributes of the class, called class attributes. This abstraction can be used in two ways. In the first case if the constituent objects are individual instances, they can be abstracted to form a set or class of those instances with the aim of focusing on the common properties shared by all

those instances. On the other hand if the constituent objects are themselves classes, this abstraction gives us the ability to model the important notion of a metaclass i.e. a class all of whose instances are classes.

OQUEL divides the database into classes, each class representing some properties and behaviour common to a set of real world objects. Unlike object-oriented programming languages which recognize only object classes and class hierarchy, OQUEL explicitly distinguishes several structural relationships such as generalization, aggregation, and association. Each of these structural relationships has different semantic properties associated with itself.

An OQUEL class is defined by the following:-

(1) Name :- this is the name of the class or the object type it represents.

(2) Structure :- this is defined by the association of the class with other classes.

(3) Operations :- these are valid operations or methods that are associated with the class or the instances of the class.

A class in OQUEL serves the following roles :- (1) it captures the definition of a prototypical object and (2) it refers to a set of instances of real world objects that conform to the prototypical definition. Each instance is therefore a representation of a real world object and consists of the actual values of properties modeled by the class, along with the global unique identifier of the object. The inverse of classification, instantiation, can be used to obtain objects that conform to the constraints associated with the properties specified by the class. Classification object classes can interact with each other by means of aggregation, generalization, association or classification. A classification object class can be formed from constituent object classes which can be aggregation, generalization, association, classification or any combination of these four. Aggregate operations like count, average, sum, min etc. can be formed for the class attributes of classification object classes.

By inserting objects and expressing the belief that they are instances of any classes, the system has the capability to reason exactly about the internal structure of these objects. For example, by expressing that a particular object is an instance of an oil_well, the model can infer that this object has location, drilling_company, depth, etc. It might be important to point out here this view of classification and instantiation does not correspond to the one supported by conventional database models. In database systems, the belief that an object is an instance of a class is not explicitly represented in the model, and can not for that reason, be queried.

Mapping of object classes

1) For each class C in the OQUEL schema we create a relation R that includes all simple attributes of C and object identifiers for composite attributes.

2) We add an object identifier (OID) as an additional attribute to identify every instance of the class.

3) The following relation, 'cl_cattr', has been used to hold information about the class attributes and their domain to help query translation. This will be discussed later.

cl_cattr		
cl_name	**cattr_name**	**cattr_type**
person	*oid*	*100*
person	*name*	*c20*
person	*dob*	*date*

7.3.3 Generalization/Specialization

Two or more object classes can be generalized to form a higher level object class, or one can use the inverse of generalization, namely, specialization, whereby new object classes can be defined to be subclasses of one or more object classes. The two are complementary to each other in the sense that generalization is a bottom up abstraction process with the object structure inherited up the hierarchy, while specialization is a top down abstraction process with object structure being inherited down the hierarchy. The construct used to model it is the object class of the type superclass or subclass. The set/subset constraint between each defining object class and its subclass is implicit in the model.

The most commonly used form of generalization involves a generic class that has subsets which are non-overlapping (N) and total (T) and as such would be labeled (N,T) as in figure 3.1. Other variants of generalization are non-overlapping and partial (N,P), or overlapping and partial (O,P) as in figure 3.2. and figure 3.3 respectively.

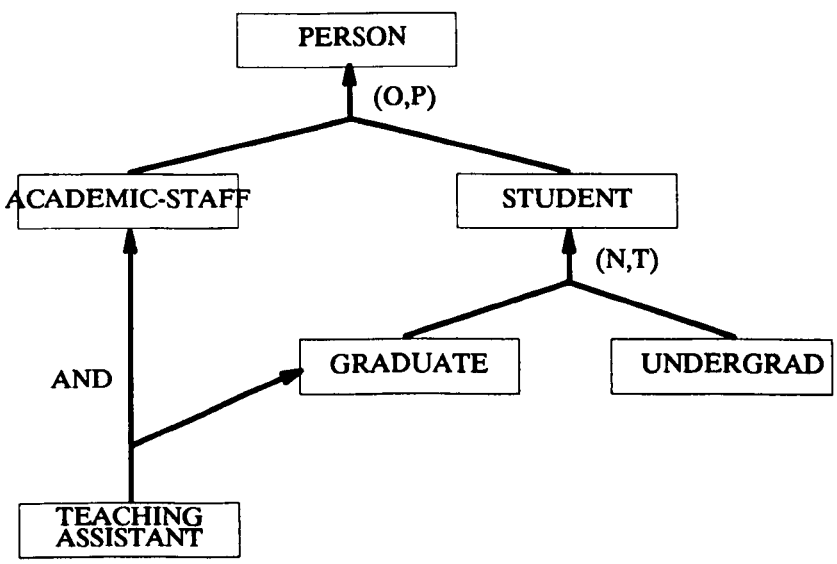


Figure 3.1

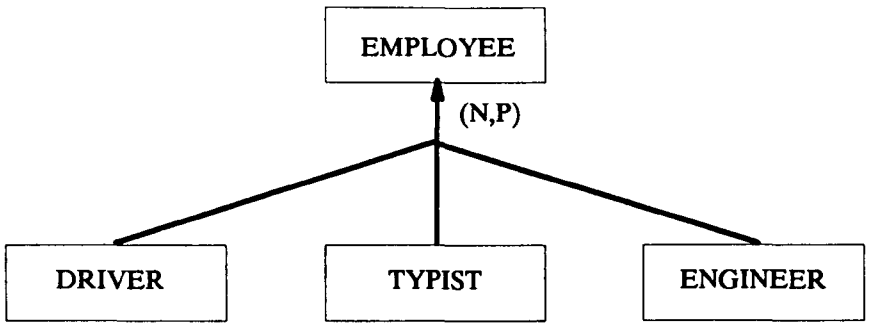


Figure 3.2

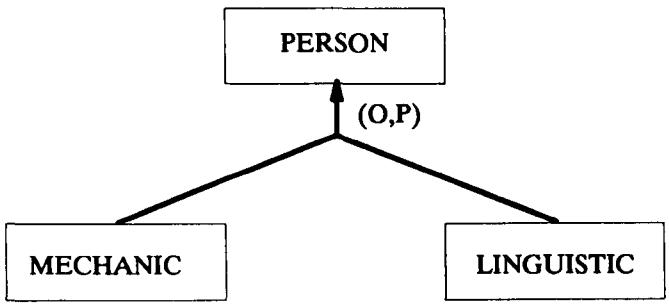


Figure 3.3

Mapping Generalization/Specialization

There are several options for mapping a number of subclasses that together form a specialization. We use Attrs(C) to denote the attributes of the object class C and *oid* to denote the object identifier.

We convert each specializtion with m subclasses $\{S_1, S_2,....S_m\}$ and generalized superclass C, where the attributes of C are $\{oid,a_1,...a_n\}$ into relation schemas using one of the following options:-

(1) Create a relation L for C with attributes

Attrs(L)= $\{oid,a_1,....a_n\}$.

Also create a relation L_i for each subclass S_i $1<i<m$, with the attributes

Attrs(L_i) = {*oid*}U {Attrs(S_i)}.

(2) Create a relation L_i for each subclass S_i, $1<i<m$, with the attributes

Attrs(L_i) = {Attrs(S_i)} U {*oid*,a_1,...a_n}

(3) Create a single relation L with attributes

Attrs(L)= {*oid*,a_1,...,a_n}U{Attrs(S_1)}U...U{Attrs(S_m)}U{t}.

This option is for specialization whose subclasses are disjoint, and t is a type attribute that indicates the subclass to which each object instance belongs.

(4) Create a single relation schema L with attributes

Attrs(L) = {*oid*,a_1,...,a_n} U { Attrs(S_1) }U...U{Attrs(S_m)}U {t_1,...,t_m}.

This option is for specialization whose subclasses are overlapping and each t_i, $1<i<m$, is a boolean attribute to indicate whether or not an object instance belongs to subclass S_i. This option has potential for creating a large number of null values.

Option(1):- creates a relation L_i for each subclass S_i. L_i includes the specific attributes of S_i plus the OID of the superclass C, which is propagated to L_i and becomes its OID. Another relation L is created for the superclass C and its attributes. An EQUIJOIN operation on the OID between any L_i and L produces all the specific and inherited attributes of the object instances in S_i. This option is illustrated in figure 3.4.a. Option(1) works for any constraints on the specialization (N,O,T,P).

Option(2):- In this option the EQUIJOIN operation is built into the schema and the relation L is done away with as illustrated in figure 3.4.b. This option works well only with both the disjoint and total constraints on the specialization. If the specialization is not total we lose any object instance that does not belong to any of the subclasses S_i . If the specialization is not disjoint, then an object instance belonging to more than one of the subclasses will have its inherited attributes from the superclass C

stored redundantly in more than one L_i, causing update anomalies. With option(2) we do not have any relation that holds all the objects in the superclass C; we need to apply an OUTER UNION operation to the L_i relations to retrieve all the objects in C. Also, whenever we search for an arbitrary object in C, we must search all the m relations L_i.

Option(3) and (4)(note figure 3.4.c, figure 3.4.d) create a single relation to represent the Superclass C and all its subclasses. Any object that does not belong to some subclasses will have null values for the specific attributes of these subclasses. If few specific subclass attributes exist, these mappings are preferable to options (1) and (2), because they do away with need to specify EQUIJOIN and OUTER UNION operations and hence can result in a more efficient implementation.

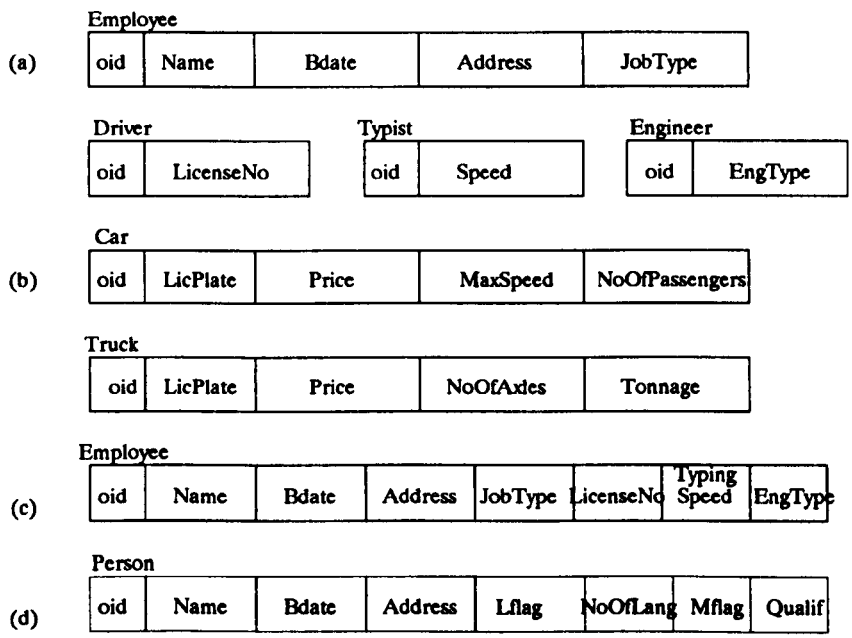


Figure 3.4 Illustrating Options for Mapping Specialization/Generalization

The following relation, 'inh_rel', is used to represent the relationship between a class and its superclasses. This relation is populated when a new type with an **inherit** clause is created. This relation can be manipulated directly by inserting and deleting such relationships to support schema evolution which will be discussed later.

inh_rel	
class_name	**superclass**
student	*person*
employee	*person*

3.4 Mapping of Association

When a relationship between two or more objects is of interest to an application and when the relationship itself takes properties, then such a relationship is modeled in OQUEL as an object relationship. The following mapping is used, based on the cardinality relationships:-

(1) for each binary 1:1 relationship R, we identify object classes S and T that correspond to the object classes participating in R. We choose one of the object classes, S say, and include in S the object identifier of T. It is better to choose an object class with total participation in R in the role of S. We include all the simple attributes of R as attributes of S, and the OID of composite ones.

(2) for each binary 1:N relationship R, we identify the object class S that represents the participating object class at the N-side of the relationship. We include in S the object identifier of the object class T, that represents the other object class participating in R. This is because each object instance on the N-side is related to at most one object instance on the 1-side of the relationship. We include any simple attributes of the 1:N relationship as attributes of S.

(3) For each M:N binary relationship R, we create a new relation S to represent R. We include in S the object identifiers of the participating object classes. We also

include any simple attributes of the M:N relationship as attributes of S.

(4) For each n-ary relationship R, n>2, we create a new relation S to represent R. We include in S the object identifiers of the object classes participating in the relationship. We also include any simple attributes of the n-ary relationship as attributes of S.

3.5 Aggregation

It may happen that we want to consider two or more classes, possibly connected by relations, as a higher order class with its own relations. This is represented as an aggregation. Aggregation is a tool to represent n-ary relationship. The aggregation concept corresponds to the notion of property in the sense of composition, since it expresses the idea that, for example, a computer consists of (cpu, main-memory, peripheral devices). More stringently, we can say that the aggregation concept describes the necessary properties, that an object must own, in order to exist consistently. Clearly, it is hard to imagine a computer without a cpu. The aggregation concept treats a collection of element objects as properties of a single higher level object. These properties, also called part-of properties, express relationships which usually do not change with time and which must be non-null in order to characterize objects completely. Applying aggregation recursively over the objects, an aggregation hierarchy is built, having the atomic objects as its leaves.

(1) For each aggregation we create a relation R. We include in R the simple attributes and object identifier of complex objects.

(2) For each multivalued attribute A, we create a new relation R that includes an attribute corresponding to A plus the object identifier of the relation that represents the object class.

3.6 Methods and the behaviour of objects

As mentioned in *chapter five* the operational knowledge of an object on how to manipulate and interpret the values of attributes is organized as a set of methods. Attributes can be read or written only by applying appropriate methods. The methods consist of a definition part and an implementation part. The definition of the method consists of the specification of the method selector (operation name), the arguments and their corresponding types and the result type, if any. These determine the method interface. The implementation of the method represents the functionality of an operation. In general methods will be defined with an object class. Their functionality is common for all instances of an object class. Therefore both the method interface and implementation need not be stored for each instance for economy.

We use the following relation, 'cl_methods', to store the definition and implementation of the methods and their respective classes.

cl_methods		
class_name	**method_name**	**method_code**
person	*get_dob*	**range of** r **is** *person* **retrieve**(r.dob)**where** r.name = $1
person	*get_age*	*@getage*

7.4 TYPE DEFINITION

OQUEL type definition is powerful and flexible. It captures data abstraction as a special case when there are only operations (methods) and no properties. It captures records as a special case when there are only properties and no operations (methods). The inheritance mechanism allows the new type to be defined incrementally by specifying only the operations and properties in which the new type differs from previously defined types. Since methods may be viewed as special kinds of properties and properties may be specified by a pair of get and put methods, we could specify object types in terms of just a set of methods or just a set of properties. But there are advantages in expressive power in allowing both methods and properties in specifying object types.

In this section we will illustrate through examples the syntax we use to capture the various structural concepts used to build an OQUEL schema, such as generalization, classification, instantiation, aggregation, and association. The behavioral aspects will be discussed in the method section (See section 7.6).

7.4.1 Generalization

In order to capture the generalization relationship between classes the **inherit** construct is introduced. The following examples illustrate its usage:-

Syntax:-

define type type_name (attr = dom, ...) **inherit** (type_name1, ..)

examples:-

Example 7.4.1

```
define type person
(
   name = c20 ,
   dob  = date
)
```

When the syntax used in example 7.4.1 is parsed, it results in the following actions:-

a) The definition of each class is extended by an object identifier (OID).

b) The catalog relation 'cl_cattrs' which represents the classes and their member attributes is populated as follows:-

cl_cattr		
cl_name	**cattr_name**	**cattr_type**
person	*oid*	*i4*
person	*name*	*c20*
person	*dob*	*date*

Example 7.4.2

```
define type student
(
   gpa = f4 ,
   dept = Department ,
   course = Course,
   level  = c20
) inherit ( Person )
```

Example 7.4.3

```
define type employee
(
   jobtitle = c20 ,
   dept     = Department ,
   manager  = Employee ,
   sal      = i4
)inherit ( Person )
```

When the syntax used in example 7.4.2 is parsed, it results in the following actions:-

a) The definition of each class is extended by an object identifier (OID).

b) The catalog relation 'cl_cattrs' which represents the classes and their member attributes is populated as follows:-

cl_cattr		
cl_name	**cattr_name**	**cattr_type**
person	*oid*	*i4*
person	*name*	*c20*
person	*dob*	*date*
student	*oid*	*i4*
student	*gpa*	*f4*
student	*dept*	*Department*
student	*course*	*Course*
student	*level*	*c20*

c) The inheritance relation 'inh_rel' which captures the generalization relationship between classes is populated as follows:-

inh_rel	
class_name	**superclass**
student	*person*

4.2 Aggregation

Aggregation is represented through the (attr = dom,...) sequence. Beside the value attributes which are used to represent atomic properties of classes, we use reference attributes to capture relationship between objects. These references can be shared or exclusive, dependent or independent, according to the intended semantics. The reference attributes will be used to provide navigational access which will be

illustrated in section 7.8 of this chapter when the support for complex objects is discussed.

7.5 OBJECT CREATION AND MANIPULATION

7.5.1 CREATING OBJECT INSTANCES

Support for Classification/Instantiation

Separating a type and its extension allows for type reusability. This facility is lacking in database systems although it is extensively used in programming languages. The aim can be achieved by the following construct in which 'relation_name' is the generic type, and 'type_name' is type of any instance. Note that the generic type may be a subset of a larger class.

Syntax:-

```
create relation_name { type_name }
```

Example 7.5.1

```
create people   { person  }

create students { student }

create postgrads { student }

create undergrads { student }
```

To create an instance of a class the following syntax is used:-

Syntax:-

```
add instance_of class_name ( attr = val, ...)
```

Example 7.5.2

```
add instance_of people (name = "Tom", dob = "10/10/59")
```

This command is translated as follows:-

a) An oid is generated by incrementing the relation which holds the oid counter as follows :-

```
replace oid_rel (oid=oid+1)
```

b) a new tuple is added to the appropriate class-relation.

```
range of tmp is oid_rel
append people (oid=tmp.oid,name="Tom",dob="10/10/59")
```

Example 7.5.3

```
add instance_of students (course="cs1",level="grad") where
  people.name = "Tom"
```

This command is translated as follows :-

a) Since a student instance is assumed to be a person and hence has an occurrence in the relation people, the oid of the student is retrieved as follows:-

```
range of person is people
retrieve tmp (oid=person.oid) where person.name = "Tom"
```

b) using the oid which has been retrieved, a new tuple is added to the sub-class relation.

```
range of tmp1 is tmp
append students (oid=tmp1.oid,course="cs1",level="grad")
```

7.5.2 UPDATING OBJECT INSTANCES

In order to make changes to objects the following syntax is used:-

```
modify instance_of <class_name> to contain <target_list> where <qual>;
```

Example 7.5.4

```
modify instance_of students to contain (course ="AI") where students.oid =
people.oid
and people.name = "Tom";
```

7.5.3 RETRIEVING OBJECT INSTANCES

Databases traditionally provide operations based on selection by content. This is especially true in relational systems, where all relationships between entities are represented by contents, and all operations are based on contents. These facts can be drawn from Codd's 12 golden rules for evaluating relational systems [Date86].

1) Rule1:- The Information Rule

All information in a relational database is represented explicitly at the logical level in exactly one-way, by values in tables.

2) Rule2:- Guaranteed Access Rule

Each and every atomic value in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name.

In object-oriented systems object contents are typically encapsulated i.e. hidden. We are not supposed to know the values of an object's attributes. By using methods which are discussed in section 7.6 an object can be manipulated through methods.

However, there is a need to access objects through more flexible syntax and to make the the object internally transparent at least to some categories of users (e.g.type implementer) and to allow for formulation of unpredictable queries. To

achieve this goal we provide the following retrieval syntax:-

retrieve <target_list> **where** <qual>;

<qual>::= <adt_qual> | <complex_obj_qual>

<adt_qual>::= <spatial_qual> | <geol_time_qual>

(See Sections 7.8 and 7.9 of this chapter).

7.5.4 REMOVING OBJECT INSTANCES

The following syntax is used to remove an object from the database.

Syntax:-

```
remove instance_of class_name where qual ;
```

Example 7.5.5

```
remove instance_of student where student.course = "AI" ;
```

Since the removal of a complex object will also necessitate the removal of its unshared component objects and the removal of an instance of a class will also imply the removal of the instance from each of the related subclasses, an overloaded syntax is used which achieves the removal of different objects through methods. For example, removing an object from the academic-people class will result in its removal from the student and staff classes. The following relation, 'cl_methods' shows the implementation of *remove* method for the class academic-people.

cl_methods		
class_name	**method_name**	**method_code**
academic-people	*remove*	**range of** s **is** students **range of** p **is** academic-people **range of** as **is** academic-staff **delete** students **where** s.oid = p.oid **and** p.name = $1 **delete** academic-staff **where** as.oid = p.oid **and** p.name = $1 **delete** academic-people **where** p.name = $1

Example 7.5.6 In order to remove Tom from the class academic-people we use the following syntax:-

```
execute (academic-people,remove,Tom)
```

When this command is executed the parameter $1 in the *remove* method of the class academic-people will be replaced by Tom and the query is executed as will be discussed in sections 7.6 and 7.10.

7.6 SPECIFICATION AND MANIPULATION OF METHODS

Associating methods with schema types provides a mechanism for defining derived attributes. Managing objects through methods enables the enforcement of semantic integrity. Accessing the schema only via its methods makes the schema type an abstract data type in its own right. In OQUEL, we have two types of methods, one type is written in Ingres/Quel with parameters, while the other is written in C++ with embedded Quel commands. The latter type is motivated by the fact that Quel like the other current query languages is not computationally complete.

7.6.1 Quel Methods

The implementation of methods involves an initial definition, storage of that definition, the ability to call (execute) the method, the display of methods for checking purposes and, lastly, deletion. These facilities are illustrated in the following examples:-

Methods Creation

Syntax:-

```
create class_name method method_name ( method_code )
```

Example 7.6.1

```
create people method get_dob ( range of person is people retrieve (person.dob)
where person.name = $1 )
```

Methods Storage

The OQUEL relation 'cl_methods' is used for the storage of methods. Thus the example above will be recorded in the following form:

cl_methods		
class_name	**method_name**	**method_code**
person	*get_dob*	**range of** r **is** people **retrieve** (r.dob) **where** r.name = $1

Methods execution

Syntax:-

Execute (type,method_name,object);

Example 7.6.2

Execute (people,get_dob,David)

Answer:-

10/10/67

Methods display

Syntax:-

display <class_name> **methods**

Example 7.6.3

display people **methods**

Answer:-

```
get_dob
```

Methods removal

Syntax:-

```
remove class_name method where method_name = $1
```

Example 7.6.4

```
remove people method where method_name ="get_dob"
```

7.6.2 C++ METHODS

Problems with DML

Most database query languages are not computationally complete. A computationally complete DML, will make it possible to do more computation in the database, unfortunately, this power may however make it difficult to do query optimization unless the properties of the operations like associativity, commutativity and other algebraic properties are specified to the query optimizer.

C++ Methods creation

These methods are written in C++ with embedded Quel code and stored in a file. The pathname of the file is stored in the method relation 'cl_methods' by the following command:-

Syntax:-

```
create class_name method method_name ( method_pathname );
```

Example 7.6.5

```
create people method get_age ( @getpersonage );
```

C++ Methods Storage

cl_methods		
class_name	**method_name**	**method_code**
people	*get_age*	@getpersonage

Methods execution

Syntax:-

```
Execute (type,method_name,object);
```

Example 7.6.6

A simple example of a user-defined datatype is the non-metric measurement system based on feet and inches. In this system, 3′11″ + 1″ = 4′ (3 feet 11 inches plus 1 inch equal 4 feet).

However, conventional DBMSs do not have a datatype for feet and inches and thus do not understand the concept of feet and inches. Therefore, this data must somehow be converted into a format understood by the database.

One option is to store the measurement either as decimal or floating point data. In either case, the values must be approximated. In the example above, the data would be stored as 3.91666 and 0.08333 feet in the database. However, adding

3.91666 and 0.08333 in floating point arithmetic gives you 3.99999 rather than 4. So, not only does this solution have a very unnatural user interface, it also sacrifices database integrity and precision.

In OQUEL, we have provided C++ methods to express more complex operations on data. For example, let us have the following relation (road) which holds information about road number (road_no) and road width (width). Suppose the road width is measured in feet and inches. Let us have the following requirements:-

User Requirements:-

Increase the width of road no. 10 by 3′ 10″ ?

OQUEL Query:-

execute (road,increase_width,10);

The system will use the class_name (i.e. road) and the method_name (i.e. increase_width) to extract the pathname to the method_code from the cl_methods relation (i.e. @increase_width).

cl_methods		
class_name	**method_name**	**method_code**
road	*increase_width*	*@increase_width*

The following steps are taken to include the methods into the system:-

1. The file road.h has been preprocessed by eqc preprocessor
2. The file @increase_width is renamed and compiled with C++ compiler
3. A new OQUEL system is formed by compiling and linking a version of OQUEL and the output of the C++ file @increase_width
4. The new OQUEL system is run . These steps are done through system calls.

When @increase_width is executed the system will interact with the user as follows:-

OQUEL/User Interaction :-

OQUEL: road_no increment ?

User: 10 3:10

The road relation will be as follows:-

i) **Before** execution of the **method:-**

road	
road_no	**width**
10	23:11
20	17:9

ii) **After** execution of the **method:-**

road	
road_no	**width**
10	27:9
20	17:9

```
//FILE @increase_width  //this file contains a call to the method increase_width
// this file is renamed by the system and passed to the C++ compiler through system calls
#include "road.h"
#include"/usr/include/CC/stream.h"
main()
{
road rd(0,"0");        // create a road object
int rd_no;             // declare a road no.
char incr[20];         // and the increment to the road width
printf("enter road_no , increment "); // enter the values
scanf("%i %s ", &rd_no,incr );
rd.increase_width(rd_no,incr); //call the method increase_width
}
```

```
//FILE ROAD.H  // this file contains the definition of the method increase_width
#include <stream.h>
#include <string.h>
static char *strsave(char* s) // allocate memory for the members
{
 char *p;
 extern char *malloc();
 p = new char[strlen(s)+1];
 strcpy(p,s);
 return(p);
}
class road
{
public:
  char* road_width;
  int road_no;
  road(int rdno,char* wdth); // a constructor for the road class
  void increase_width( int rd_no,char* width); // a method to increase the road width
};
road::road(int rdno, char* wdth)
{
  road_no=rdno;
  road_width=wdth;
}
void road::increase_width(int rd_no,char*  width) // define increase_width method
{
```

```
char *f1t,*f2t; // variables to hold extracted no. of feet from increment and the old road width
char *i1t,*i2t; // variables to hold extracted no. of inches from increment and the old road width
char *f3t, *i3t;// variables to hold the formatted total no. of feet and inches resp.
char str2[21];  // C++ variable to hold the new road width
char str3[22];  // C++ variable to hold the old road width
int f1,f2,f3; //no. of feet in the increment, no. of feet in the old width, and no. of feet in the new width.
int i1,i2,i3; //no. of inches in the increment, old road width, and the new road width resp.
road_no=rd_no;
road_width = strsave(width);
char cwidth[21]; // ingres variable to hold the road width
int rd1_no;     // ingres variable to hold the road no.
rd1_no=rd_no;
/* # line 38 "road.h" */          /* ingres */ // open ingres database
 {
  IIingopen(0,"oodb",(char *)0);
 }
/* # line 39 "road.h" */          /* range */
 {
  IIwritedb("range of r=road");
  IIsyncup((char *)0,0);
 }
/* # line 40 "road.h" */          /* retrieve */ // retrieve the road width to be modified
 {                                  // in the variable cwidth given the road no.
  IIwritedb("retrieve(cwidth=r.width)where r.road_no=");
  IIsetdom(1,30,4,&rd1_no);
  IIwritedb(" ");
  IIretinit((char *)0,0);
  if (IIerrtest() != 0) goto IIrtE1;
IIrtB1:
  while (IInextget() != 0) {
   IIretdom(1,32,0,cwidth);
   if (IIerrtest() != 0) goto IIrtB1;
  } /* IInextget */
  IIflush((char *)0,0);
IIrtE1:;
 }
/* # line 41 "road.h" */          /* host code */ // calculate the new road width
f1t = new char[10];      // create a buffer to hold the no. of feet from the increment variable
f1t = strtok(width,":"); // use strtok() from the standard library to pull out the no. of feet
f1 = atoi(f1t);          // from the incerement variable and convert to integers
i1t = new char[10];      // create a buffer to hold the no. of inches from the increment variable
```

```
i1t = strtok(0,0);        // use strtok() to strip off the inches and
i1 = atoi(i1t);           // convert to integers
strcpy(str3,cwidth);      // copy the old road width from the ingres variable
strcat(str3,'\0');        // into C++ variable and terminate with null
f2t = new char[10];       // create a buffer to hold the no. of feet from the old road width
f2t = strtok(str3,":");   // use strtok() to pull out the no. of feet
f2 = atoi(f2t);           // convert to to integers
i2t = new char[20];       // create a buffer to hold the no. of inches
i2t = strrchr(cwidth,':');// locate the colon and return a pointer to it
i2t++;                    // return a pointer to the inches
i2 = atoi(i2t);           // convert to integers
f3 = f1+f2+ (i1+i2) / 12; // calculate the total feet and
i3 = (i1+i2) % 12;        // total inches
*str2 = '\0';
f3t = new char[10];       // create a buffer and fill it with
sprintf(f3t,"%i",f3);     // the total no. of feet
int j = strlen(f3t);      // return the length of the buffer
f3t[j+1] = '\0';          // terminate the string
strcat(str2,f3t);         // concatenate the total no. of feet to str2
strcat(str2,":");         // add colon in the appropriate place
i3t = new char[10];       // create a buffer and fill it with
sprintf(i3t,"%i ",i3);    // the total no. of inches
int k = strlen(i3t);      // return the length of the length of the string in the buffer
i3t[k+1] ='\0';           // terminate the string
strcat(str2,i3t);         // concatenate the total no. of inches in the appropriate place
strcpy(cwidth,str2);      // copy from C++ variable to ingres variable
                          // replace the old road width with the new road width
/* # line 52 "road.h" */          /* range */
 {
  IIwritedb("range of r=road");
  IIsyncup((char *)0,0);
 }
/* # line 53 "road.h" */          /* replace */
 {
  IIwritedb("replace r(width=");
  IIsetdom(1,32,0,cwidth);
  IIwritedb(" )where r.road_no=");
  IIsetdom(1,30,4,&rd1_no);
  IIwritedb(" ");
  IIsyncup((char *)0,0);
 }
```

```
/* # line 54 "road.h" */        /* exit */
  {
    IIexit();
  }
/* # line 55 "road.h" */        /* host code */
}
```

It is noteworthy to mention here that we use the same syntax and storage structure for both Quel methods and C++ methods. However, we introduce the special character (@) as a prefix for C++ methods to differentiate them from Quel methods in the preprocessing.

7.6.3 Examples of applications of methods

Methods and Semantic Integrity

When we design a schema for a particular database application, one of the important activities is to identify the integrity constraints that must hold in the database. We usually implement a database to store information about some part of the real world about which we have many integrity constraints. We want to specify many of these constraints to the DBMS and if possible have the DBMS be responsible for enforcing them. In any data model, there are some types of integrity constraints that can be specified and represented directly in the database schema of that model. These are called the implicit constraints of the data model, and are specified using DDL. Each data model includes a different set of implicit constraints that can be directly represented in its schema. However, no data model is capable of representing all types of constraints that may occur in an application. Hence, it is usually necessary to specify additional explicit constraints on each particular database schema that represents a particular application.

Support and enforcement of semantic integrity is a weak point of many existing

RDBMSs. Examples of state constraints that occur frequently in database applications are (a) value set constraints, (b) attribute structural constraints, (c) relationship structural constraints , (d) superclass/subclass constraints, (e) general semantic constraints that do not fall in any of the above categories.

We are interested in category (e) of the constraints. An example is the constraint " the salary of an employee must not be greater that the salary of the manager of the department for whom that employee works ". Another example could be " the total number of hours that each employee works on all his projects should not exceed 45 hrs per week". These constraints and other explicit constraints can not be specified directly in the schema using the DDL.

In OQUEL, we propose a technique for handling explicit constraints by encoding them as part of the methods that are encapsulated with objects. Storing these constraints in the DBMS will provide applications with the opportunity to refer to these constraints and thus provide semantic integrity control. We are, in effect, allowing applications to share the semantics of the database. Since the values associated with constraints may change with time, we propose storing constraints in a relation so that the constraints can be updated without affecting the methods accessing them as parameters.

Example 7.6.7

Suppose we need to raise the salary of an employee and we have the following constraint:-

1. the total salaries of employees < dept.budget

This requirement can be achieved by implementing the **method** *raise_sal* as illustrated below:

employee		
emp_oid	**dep_oid**	**sal**
4001	2000	950.00
4002	2000	900.00

dep_cnstr	
dep_oid	**budget**
2000	10000.00
3000	8000.00

cl_methods		
class_name	**method_name**	**method_code**
employee	*raise_sal*	**range of** r1 **is** employee **range of** r2 **is** dep_cnstr **replace** r1 (sal = r1.sal + $2) **where** r1.emp_oid = $1 **and** sum(r1.sal) < r2.budget **and** r2.dep_oid = r1.dep_oid

This technique will provide flexibility since the department budget can be increased without affecting the code of the method and it provides semantic integrity since all applications will use the same method for raising the salary of an employee.

6.3.2 Management of Complex Objects

The use of methods allows the user to express more meaningful operations at a higher level. Even the traditional update operations (i.e. delete, modify, etc) can be overloaded to manipulate the complex objects using parameterized operations.

Let us have a complex object CELL3 which is composed from instances of CELL1, instances of CELL2 and instances of PATH, which are in turn composed of instances of SEGMENT. This information is represented in figure 6.1.

CELL			
PID	TID	cell_name	cdata
@000000000	@123000100	CELL1	xxxx
@000000000	@123000200	CELL2	yyyy
@000000000	@123000300	CELL3	zzzz

INSTANCE			
PID	TID	m_cell	idata
@123000300	@123000301	@123000100	aaaa
@123000300	@123000302	@123000100	bbbb
@123000300	@123000303	@123000100	cccc
@123000300	@123000304	@123000100	dddd
@123000300	@123000305	@123000200	eeee

PATH		
PID	TID	pdata
@123000300	@123000306	kkkk
@123000300	@123000307	llll
@123000300	@123000308	mmmm
@123000300	@123000309	nnnn
@123000300	@123000310	oooo
@123000300	@123000311	pppp

SEGMENT		
PID	TID	sdata
@123000308	@123000312	qqqq
@123000308	@123000313	rrrr
@123000308	@123000314	ssss

Figure 6.1 Complex Object Model description of the CELL

Let us have the following intended semantics:- deleting an instance of a CELL implies removing all its components from the INSTANCE and PATH relations, and removing a PATH implies removing all its components from a SEGMENT relation. This semantic can be expressed in OQUEL as a **delete** method. This method is

stored in the relation 'cl_methods' as a list of Quel commands with parameters. These parameters are supplied through the interface.

Example 7.6.8

User requirement:

Delete CELL3

OQUEL query:

execute (CELL,delete,CELL3);

When this method is executed the $1 parameter in figure 6.2 is replaced by CELL3 and the method is executed with deletion propagated to the component objects to achieve the desired semantics.

cl_methods		
class_name	**method_name**	**method_code**
CELL	*delete*	**range of** r1 **is** CELL **range of** r2 **is** INSTANCE **range of** r3 **is** PATH **range of** r4 **is** SEGMENT **delete** r4 **where** r4.PID=r3.TID **and** r3.PID = r1.TID **and** r1.cell_name=$1 **delete** r3 **where** r3.PID=r1.TID **and** r1.cell_name=$1 **delete** r2 **where** r3.PID=r1.TID **and** r1.cell_name=$1 **delete** r1 **where** r1.cell_name=$1

Figure 6.2

7.7 Schema Evolution

7.7.1 Introduction

7.7.2 OQUEL Schema

7.7.3 Schema operations

Addition of Attributes

Deletion of Attributes

Modification of Attributes Names

Addition of Methods

Deletion of Methods

Modification of Methods

Addition of Types

Deletion of Types

7.7.1 Introduction

One of the important requirements of advanced applications is schema evolution - the ability of the user to change the database schema. The types of changes required include creation and deletion of a class, addition and deletion of attributes and methods. In object-oriented databases there are two types of schema evolution: one involves the class definition and the other involves the generalization hierarchy.

Existing database systems allow only few types of schema changes. For example, they allow only creation and deletion of relations and no addition or deletion of attributes. This is mainly because the applications they support are conventional record-oriented business applications which do not require more than a few types of schema changes and also the data models they support are not as semantically rich as object-oriented data models.

7.7.2 OQUEL Schema

In OQUEL, a database schema is a set of class definitions connected by the superclass/subclass relationship. It is represented by a class lattice. Subclasses do not necessarily partition the instances of the superclasses. Therefore, the set of instances in subclasses are not necessarily disjoint. Orthogonal to the generalization hierarchy is the composition hierarchy. Composite object classes are defined using references to other classes. The referencing attributes have non_atomic values. The value domains are the sets of instances of the referenced classes.

7.7.3 Schema operations

Operations which change schemas fall into the following categories.

1) class definition changes

2) class relationship changes

In the first category we have the following changes:-

addition of attributes

deletion of attributes

modification of attributes name

addition of methods

deletion of methods

modification of methods

In the second category we have the following changes:-

addition of types

deletion of types

In this section we will discuss the implementation of some changes to the schema and their effect on the data in the database.

Addition of attributes

We use the following syntax to add attributes to the class definition

```
extend type <type_name> <attr_spec>
```

For example, if we have a class C1, with attributes a_1, and a_2, the following description will appear in the catalog relation 'cl_cattr':-

cl_cattr		
cl_name	**attr_name**	**attr_type**
C1	*a1*	*c20*
C1	*a2*	*c10*

Suppose we want to add a new attribute a3 to the class C1. We use the following command to make this change

extend type C1 (a3 = c15)

the effect of this command is as follows:-

1) change the description of C1 in the catalog table 'cl_cattr' to look as follows :-

cl_cattr		
cl_name	**attr_name**	**attr_type**
C1	*a1*	*c20*
C1	*a2*	*c10*
C1	*a3*	*c15*

2) a temporary relation tmp is created from this description by the following command

```
create tmp { C1 }
```

3) the old class C1 extension is copied to the tmp relation with null values for the newly added attribute (i.e a3).

4) the old class C1 is destroyed

5) the tmp class is renamed as C1

Deletion of attributes

The following syntax is used to remove an attribute from a class.

```
contract type <type_name> <attr_spec>
```

the effect of this command is as follows:-

1) The attributes in the specification list are removed from the catalog relation 'cl_cattr'.

2) A new tmp class is created from the updated description in the catalog using the following command:-

```
create tmp { type_name }
```

where type_name refers to the updated description in the catalog.

3) The tmp class is populated from the old class by the desired attributes

4) The old class is destroyed

5) The tmp class is renamed as the old class

Modification of attributes name

1) A new tmp class is created with the desired attribute names

2) The tmp class is populated from the old by the respective attribute names

3) The old class is destroyed

4) The tmp class is renamed as the old class

Addition of methods

Syntax for QUEL methods

```
create <class_name> method <method_name> (<method_code>);
```

Syntax for C++ methods

```
create <class_name> method <method_name> (<method_pathname>);
```

Deletion of methods

Syntax:-

```
remove <class_name> method where <method_name> = $1
```

Modification of methods

Syntax:-

```
modify <class_name> method ( <method_name> = $1, <method_code> =
$2 )
    where <method_name> = $3
```

Addition of types

Syntax:-

```
create <class_name> subtype of <class_name>
```

Deletion of types

Syntax:-

```
remove <class_name> from <class_name>
```

7.8 Support For Complex Objects

7.8.1 Introduction

7.8.2 Complex object definition

7.8.3 Complex object schema

7.8.4 Operations on complex objects

7.8.5 Mapping complex object schema to relational schema

7.8.6 Mapping object-oriented queries to relational queries

7.8.1 Introduction

Many application areas of computing such as artificial intelligence, office automation, computer-aided design, exhibit a unifying requirement of supporting complex objects. Support for complex objects will provide the user with the ability to define and manipulate a set of objects as a single logical entity for the purpose of semantic integrity and efficient storage and retrieval.

7.8.2 Complex object definition

In OQUEL we represent an arbitrary complex object as a recursively nested object. The class of an object may be defined as the domain of an attribute and the domain class, unless it is a primitive class, may in turn possess attributes and so on. The internal state of an object consists of the values of all its attributes. The value of an attribute is an instance of its domain, if the domain is a primitive class; and a reference to an instance of the domain otherwise. If the domain of an attribute is a class, the value of that attribute will be a set of object identifiers.

7.8.3 Complex object schema

In figure 8.1, we show the schema of a COMPUTER class in terms of the attributes, c_cpu, c_main_memory, c_peripheral_devices, c_manufacturer, c_software, c_users, and c_name. The domain of the c_cpu attribute is the class CPU, the attribute c_main_memory has the class MAIN_MEMORY as its domain, the domain of c_peripheral_devices is PERIPHERAL_DEVICE, the domain of c_manufacturer is COMPANY, the domain c_software is the class SOFTWARE, the domain for the attribute c_users is COMPUTER_USERS, and the domain of the c_name is the primitive class STRING. The non_primitive classes like CPU, MAIN_MEMORY, PERIPHERAL_DEVICE,

COMPANY, SOFTWARE, COMPUTER_USERS, each consist of their own sets of attributes, which in turn have associated domains.

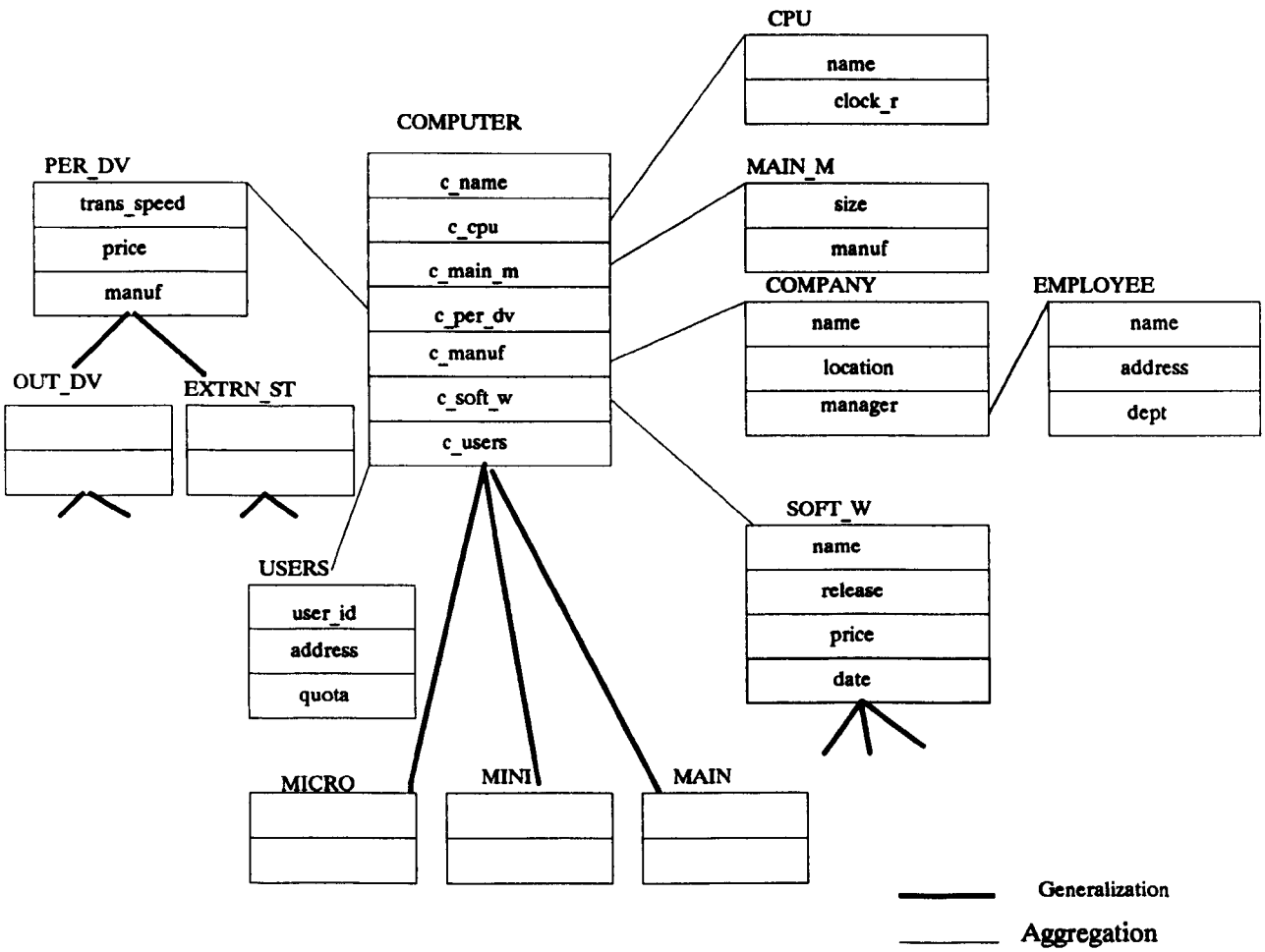


Figure 8.1

7.8.4 Operations on complex objects

The nesting of an object through the domains of its attributes immediately suggests that to fully fetch an instance, the instance and all objects it references through its attributes must recursively be fetched. In order to fetch one or more instances of a class, the class and all classes specified as non_primitive domains of the attributes of the class must be recursively traversed. For

example to fetch an instance of a COMPUTER from the example in figure 8.1 the classes which need to be traversed include CPU, MAIN_MEMORY, PERIPHERAL_DEVICE, COMPANY, SOFTWARE, COMPUTER_USERS, as well as the non_primitive domains of these classes.

However, a query may be formulated against an object_oriented schema, which will fetch only the instances of a class which satisfy certain search criteria. A query may restrict the instances of a class to be fetched by specifying predicates against any attributes of the class.

An example of a query against the schema of figure 8.1 is the following:-

Example 7.8.1

User Requirement:

Find all computers manufactured by a company located in London and which have cpu clock rate equal to 10 mhz.

OQUEL query:

```
computer get r ( r manufacturer location = "London" )
                and( r cpu clock_rate = 10 )
```

In OQUEL, in order to transform object-oriented queries into relational queries we require first the transformation of the object-oriented schema into an equivalent relational schema. Queries against the object-oriented schema can then be formulated as semantically equivalent queries against the relational schema. This provides us with techniques for building an object-oriented front end to relational systems.

7.8.5 Schema transformation

A class is modeled by a relation, and the class definition is mapped into a relation having the same name as the class's. The set of instances of a class is mapped into tuples of the relation. The attributes of the class are mapped into columns of the relation. Further the class is augmented with a system defined oid column. The value of this column in each tuple is the unique identifier of the tuple.

Transformation of the object-oriented schema of figure 8.1 yields the following relations, one for each class:-

COMPUTER (oid,c_cpu,c_main_memory,c_peripheral_devices,c_software,c_users,c_manufacturer,c_name)

CPU (oid,name,clock_rate)

MAIN_MEMORY (oid,size,manufacturer)

PERIPHERAL_DEVICE (oid,transmission_speed,price)

COMPANY (oid,name,location,manager)

SOFTWARE (oid,name,release,price,date)

COMPUTER_USERS (oid,userid,address,quota)

EMPLOYEE (oid,name,address,dept)

A set attribute requires the creation of an additional relation. If an attribute A of a class C is a set attribute whose domain is the class D, a relation C_A is created with two columns C_OID and D_OID. If an instance of C has an n_element set for the value of its attribute A, there will be n tuples in the relation C_A, and for each of these tuples the C_OID column will contain the OID of the instance of C. The n_tuples will have distinct values for the D_OID column, such that each value is the OID of an instance of the class D.

7.8.6 Query transformation

In OQUEL, we transform an object-oriented query into an equivalent relational query. For example, we use the following syntax to retrieve an object using a complex predicate.

object get o (o A1 A2 A3 ... An comp-op const)

where

object: refers to the target object class

get: is a message sent to the object

o : refers to a range variable

A1,A2,....An : refer to nested attributes whose domains are D1,D2,....Dn.

comp-op: is comparison operator

const: is a constant string or number.

The above syntax is translated to the following QUEL syntax:-

retrieve (r.oid) **where**

r.A1 = D1.oid **and** D1.A2 = D2.oid **and** D2.A3 = D3.oid

.... **and** Dn-1.An comp-op const

It is worth mentioning that these joins are hidden from the users and the system makes use of meta data to navigate the structure, resulting in a simplified user interface.

7.9 Abstract Data Types (ADTs) Domain Support

7.9.1 ADTs and domain concepts in the relational model

7.9.2 ADTs Domain Semantics

Semantics of ADTs in Spatial and Temporal Data

7.9.3. ADTs Domain Implementation

7.9.4. ADTs Domain Syntax

7.9.1 ADT and domain concepts in the relational model

In the relational model for databases, a relation is defined as a subset C of the cartesian product D_1x D_2x.. D_n, where the D_i are domains which are not necessarily distinct. No mention is made of what data types should or should not be allowed; the definition simply states that the values that appear in the relation in column i must be drawn from the set D_i. In order to distinguish between columns defined over the same domain, attribute names are introduced. Attribute names within a relation must be distinct. It is these attribute names that are used to express dependencies and are referenced in queries.

Current query languages for relational databases usually are fixed i.e. they provide only a fixed set of data types and operations. It is usually not possible to extend this set by user defined data types or functions. Yet, there is a continuing requirement as database languages evolve to extend these languages to incorporate new data types.

7.9.2 ADT Domain Semantics

Extending the domain of a relation to include instances of ADT or complex objects, requires us first to specify the required semantics of the domain. What kind of operations should be performed on instances of the domain ? These semantics can be specified in abstract terms independent of implementation details, stating, for example, what kind of comparison operations, conversion operations, computational operations, assignment operations, etc are allowed.

Semantics of ADT in Spatial and Temporal data

In this section we address two important types of data and show how they can be supported in our system. These are spatial and temporal relationships.

(a) Spatial data :-

In applications such as geophysical surveys, CAD/CAM , cartography, etc. it is important to model the space domain. The space semantics is captured by three common representations:-

1) solid representation- the space is divided up into pieces of various sizes. The spatial characteristics of an entity are then represented by the set of these pieces associated with that entity.

2) boundary representation- the spatial characteristics are represented by line segments or boundaries.

3) abstract representation- relationships with spatial semantics are used, such as above, near, behind, to associate entities.

While it is clear that many applications would benefit from such facilities, different application domains have varying requirements. For example, in VLSI application space is two dimensional, basic objects to be stored are points, line segments, rectangles and polygons. In solid modeling for most manufacturing applications and geological applications - space is three dimensional. The operations relevant to each application area are also different.

Example of support for spatial data

In geological applications, like other CAD/CAM applications, we need to define complex attributes and operations on them. It is useful to define the following data types as a first step:-

rectangle (X_b = int, Y_b = int, X_t = int, Y_t = int)

point (X = int, Y = int)

horizontal_line (Y = int)

vertical_line (X = int)

It makes no sense to define ADT without defining operations on them. We define the following operations on rectangles.

1) Exact rectangle search

To find a rectangle R1 that matches given rectangle R2, the equality operation is used. We define (R1 EQUALS R2) by the query

(R1.X_b = R2.X_b) and (R1.Y_b = R2.Y_b) and (R1.X_t = R2.X_t) and (R1.Y_t = R2.Y_t)

2) Point search

All rectangles R from a given set of rectangles that contain a given point P will satisfy the operation (R ENCLOSES P), which is defined by the query: (R.X_b <= P.X) & (R.Y_b <= P.Y) & (R.X_t >= P.X) & (R.Y_t >= P.Y)

3) Overlap detection

All rectangles R1 that overlap a given rectangular area R2 will satisfy the operation (R1 OVERLAPS R2), defined as (R1.X_b < R2.X_t) & (R1.Y_b < R2.Y_t) & (R1.X_t > R2.X_b) & (R1.Y_t > R2.Y_b)

4) Abutting rectangles

The operation (R1 ABUTS-LEFT R2) may be used to find any rectangle

R1 that touches the left edge of a given rectangular area R2. The operation is defined by the query (R1.X_t = R2.X_b) & (R1.Y_b < R2.Y_t) & (R1.Y_t > R2.Y_b)

The operation (R1 ABUTS-RIGHT R2) may be used to find any rectangle R1 that touches the right edge of a given rectangular area R2. The operation is defined by the query (R1.X_b = R2.X_t) & (R1.Y_b < R2.Y_t) & (R1.Y_t > R2.Y_b)

The operation (R1 ABUTS-BOTTOM R2) may be used to find any rectangle R1 that touches the bottom edge of a given rectangular area R2. The operation may be defined by the query (R1.Y_t = R2.Y_b) & (R1.X_b < R2.X_t) & (R1.X_t > R2.X_b)

The operation (R1 ABUTS-TOP R2) may be defined to find any rectangle R1 that touches the top edge of a given rectangular area R2. The operation is defined by the query (R1.Y_b = R2.Y_t) & (R1.X_b < R2.X_t) & (R1.X_t > R2.X_b)

5) Containment search

All rectangles R1 that are fully contained within a given rectangle R2 will satisfy the operation (R1 CONTAINED-IN R2), defined as (R1.X_b >= R2.X_b) & (R1.Y_b >= R2.Y_b) &
(R1.X_t <= R2.X_t) & (R1.Y_t <= R2.Y_t)

6) Upper/Lower & Left/Right rectangles

To find all rectangles R that lie above a horizontal-line H, the operation (R ABOVE H) is defined as follows (R.Y_b >= H.Y)

To find all rectangles that lie below a horizontal-line H, the operation (R BELOW H) is defined as follows (R.Y_t <= H.Y)

To find all rectangles R that lie to the left of a vertical-line V, the operation (R LEFT-OF V) is defined as follows (R.X_t <= V.X)

To find all the rectangles R that lie to the right of a vertical-line V, the

operation (R RIGHT-OF V) is defined as follows ($R.X_b$ >= V.X) The other operations can be expressed in a similar manner.

(b) Temporal data :-

Temporal information is a special one-dimensional case of spatial information. Temporal aspects built into databases must include three types of support for time : time points, time intervals, and abstract relationships involving time i.e before, after, during etc.

7.9.3 ADT Domain Implementation

Implementation of an ADT domain requires decisions about internal representation, operator implementation and host language interfacing. In OQUEL, we extend the relational system Ingres by an *interface layer*. The interface simulates the extended model by converting schemas and queries into their relational counterpart. The attraction of such an approach lies in its inexpensive implementation using reliable existing technology. In OQUEL, we used a field of *text* data type in the relational system Ingres to store instances of ADT domains and we translated the operations on ADT domains to operations on *text* data types supported by Ingres. Since Ingres query languages like other relational systems are not computationally complete, there is a need to resort to a host language to perform other computational operations. We chose C++ since it has the merits of OOPLs and was already incorporated in OQUEL through the provision of methods for certain complex objects. Some of the merits of the OOPL approach include :- (1) the existence of an IS-A hierarchy among domains to establish operations inheritance. (2) the possibility to use existing code to implement new operations.

7.9.4 ADT Domain Syntax

Inclusion of an ADT domain in the relational system implies extensions to the external language interface syntax to define, manipulate, and query instances of ADT domains. In OQUEL we have proposed extensions to define, manipulate and query object-extended schema.

Syntactic extensions

In the following section we illustrate through examples the handling of ADT domain attributes and their inclusion in queries.

Example 7.9.1 :-

A gravity survey class (abbreviated as gr_surv) is created as follows :-

define type gr_surv (reading = f4, time = date , elevation = f4 , location = cood)

This class contains four attributes where the first three are supported by the database system and the last one is a complex domain defined as an ADT. The location attribute takes its values from the cood domain which is defined as a list of two values i.e. <Latitude,Longitude> and stored as a *text* data type. A class gravity_surv1 will be created from the above definition using the command :-

create gravity_surv1 { gr_surv }

Instances may be added to gravity_surv1 as follows:-

add instance_of gravity_surv1 (reading = 12.4 , time = "10/10/89", elevation = 122.5, location = "23,60")

Let us suppose the following gravity_surv1 class resulted from the above population

gravity_surv1			
reading	**time**	**elevation**	**location**
12.4	*10/10/89*	*122.5*	*23,60*
13.4	*10/10/89*	*121.2*	*23,61*
14.1	*10/10/89*	*124.1*	*22,60*
15.3	*10/10/89*	*125.3*	*20,62*
16.2	*10/10/89*	*122.7*	*23,63*

Supporting ADT domains implies the needs for extensions to restriction predicates, joins , and projection clauses.

a) predicate extensions

Example 7.9.2

User requirement:

Which points have coods east of "23,61"?

OQUEL query:

range of r **is** gravity_surv1

retrieve (r.all) **where** r.location **east** "23,61"

This query with its extended predicate is translated into an equivalent relational query with operations on the ADT cood translated into operations on an equivalent *text* data type. Operations are provided in the Ingres system to manipulate *text* data types such as concatenation, extraction,length of text etc.

Example 7.9.3

Let us suppose we have an application in which rectangles form an

important abstraction for the representation of information. A rectangle is represented by its left lower corner (x_b,y_b) and top right corner (x_t,y_t). This can be defined in OQUEL as follows:-

define type <drawing> (colour=c20,rect=rectangle)

A design1 class can be created from the above definition as follows:-

create design1 {drawing};

Design1 can be populated as follows:-

add instance_of design1 (colour = "red",rect="10,20,30,30")

Let us have the following extension for the design1 class.

design1		
oid	**colour**	**rect**
1001	*red*	*10,20,30,30*
1002	*green*	*15,20,35,50*
1003	*yellow*	*10,30,40,50*
1004	*red*	*20,20,50,60*
1005	*black*	*15,20,40,40*
1006	*orange*	*5,30,40,60*

The following queries can be formulated against the stored data:-

Example 7.9.4

User requirement:

Which rectangles from the given set of rectangles overlap "10,20,30,40"

OQUEL query:

range of r **is** design1

retrieve(r.all) **where** r.rect **overlap** "10,20,30,40";

Example 7.9.5

User requirement:

Which rectangles from the given set of rectangles abut "20,20,30,40" from left ?

OQUEL query:

range of r **is** design1

retrieve (r.all) **where** r.rect **abuts_left** "20,20,30,40";

Example 7.9.6

User requirement:

Which rectangles from the given set of rectangles are contained in "10,10,60,60"

OQUEL query:

retrieve (r.all) **where** r.rect **contained_in** "10,10,60,60";

b) join extensions

Suppose we have the following class for a resistivity survey (abbreviated as resis_surv) with the following extension :-

resis_surv1	
reading	**loc**
45.3	*23,64*
22.8	*22,63*
34.6	*23,60*
45.7	*22,60*
44.5	*23,61*

Example 7.9.7

User requirement:

Which gravity readings and resistivity readings share the same coods ?

OQUEL query:

range of g **is** gravity_surv1

range of r **is** resis_surv1

retrieve (g.reading, r.reading, r.loc)

where r.loc **equals** g.location

7.10 OQUEL ARCHITECTURE

In this section we describe the architecture of OQUEL. The description consists of the hierarchies of menu interfaces used, the syntax of OQUEL, the algorithms used for the implementations, and the meta data used to support the implementation. Appendix 7A comprises a summary of OQUEL architecture.

7.10.1 Class Attributes : Type definition and evolution

7.10.1.1

define type <type_name> ($attr_i$ =dom_i,....)

This statement is used to define the type in terms of its attributes. This statement is parsed and the information about the class name, its attributes, and their domains are stored in the catalog relation <cl_cattr> as in figure 7A.24 of appendix 7A. This information in the catalog relation <cl_cattr> will be used to create a relation to hold instances of the type as illustrated in section 7.10.2.

7.10.1.2

define type <type_name> ($attr_i$ =dom_i,....) **inherit** (<type_name>,...)

This statement is similar to the above except for the inheritance part. The **inherit** construct is used to capture the generalization hierarchy. When this construct is parsed the information about the relationship between the type and its supertypes is stored in the <inh_rel> as in figure 7A.24 of appendix 7A.

7.10.1.3

extend type <type_name> (<attr_specs>)

This statement is used to extend the type definition held in the catalog relation <cl_cattr> as in figure 7A.24 of appendix 7A. The old type definition

is augmented with new attributes and their domains and any new classes defined from this type will use the new extended definition. This statement is parsed and an Ingres command is formulated to extend the catalog relation <cl_cattr> with the <class_name>,<cl_cattrs>, and their domains.

7.10.1.4

contract type <type_name> (<attr_comma_list>)

This statement is used to contract the definition of <type_name> and the attributes specified in <attr_comma_list> are dropped from the definition of the type and any new relation created from the type will use the contracted definition. This statement is parsed and an Ingres command to delete the specified attributes from <cl_cattr> of figure 7A.24 in appendix 7A is formulated.

7.10.2 Class Instances

7.10.2.1

create <class_name> { <type_name> }

This statement is used to create <class_name> to hold a set of instances of the <type_name>. When this statement is parsed the description of the <type_name> is retrieved from the <cl_cattr> of figure 7A.24 appendix 7A. An Ingres file <instance.f> with commands to create a relation with name <class_name> and description as retrieved from <cl_cattr> catalog relation is formulated. A C file <instance.c> is constructed with a system call to Ingres and to the C complier, and which takes input from <instance.f>. The program <instance.c> is compiled and its output <instance.out> is executed.

7.10.2.2

add instance of <class_name> (<attr_value_list>)

This command is parsed and an equivalent Ingres command is formulated.

If an object instance already exists, its OID is retrieved; if it is a new object an OID is created for it by updating <oid_rel> of figure 7A.24 appendix 7A. The modified command is written to an Ingres file <instance.f>. A C file <instance.c> is constructed with system calls to Ingres and to the C compiler and which takes input from <instance.f>. <instance.c> is compiled and its output <instance.out> is executed.

7.10.2.3

modify instance of <class_name> **to contain** (<target_list>) **where** <qual>;

This command is parsed and an equivalent Ingres command is formulated and written to Ingres file <instance.f>. A C file <instance.c> with a system call to Ingres and to the C compiler, and which takes input from <instance.f> is formulated. <instance.c> is compiled and its output is executed.

7.10.2.4

remove instance of <class_name> **where** <qual>;

A similar algorithm to the modify command (i.e 7.10.2.3) is used for this command.

7.10.3 Class Methods

7.10.3.1

create <class_name> **method** <method_name> (<method_code>);

This command is used to create Quel methods and to store them in textual fields in Ingres. The catalog relation <cl_methods> figure 7A.24 of appendix 7A is used to store the <class_name>, <method_name> and <method_code>. This command is parsed and an Ingres command is created to add the method to the class designated by <class_name>. The Ingres command is then stored in the Ingres file <instance.f>. A C file <instance.c> is

created with a system call to Ingres which takes input from <instance.f>. The program <instance.c> is compiled and executed.

7.10.3.2

```
modify <class_name> method (<method_name>=$1,<method_code>=$2)
where <method_name>=$3;
```

This command is parsed and an equivalent Ingres command is generated and stored in the file <instance.f>. A C file <instance.c> which takes input from <instance.f> is created. <instance.c> is compiled and executed.

7.10.3.3

```
remove <class_name> method where <method_name> =$1
```

This command is handled in a similar manner to the above command (i.e. 7.10.3.2).

7.10.3.4

```
display <class_name> methods
```

This command is parsed and an Ingres query is formulated and stored in <instance.f>. A C file <instance.c> is created which includes a system call to Ingres and takes input from <instance.f>. The program <instance.c> is compiled and executed.

7.10.3.5

```
execute (<class_name>,<method_name>,<params>);
```

When this method is parsed the <method_name> associated with the given <class_name> is retrieved from the catalog relation <cl_methods> in a long C variable. The formal parameters of the method are replaced by the actual parameters through a string substitution process. The modified method is stored in the file <instance.f>. A C file <instance.c> is created with system calls to Ingres and to the C compiler, and which takes input from <instance.f>.

The program <instance.c> is compiled and its output is executed.

7.10.4 Class Hierarchy

7.10.4.1

create <class_name> **subtype of** <class_name>

7.10.4.2

remove <class_name> **from** <supclass_name>

These commands are parsed and equivalent Ingres commands are formulated and sent to Ingres for execution.

7.10.5 Complex Objects

7.10.5.1

<object_class> **get** <range_var> <complex_qual>

This command is parsed and the information in the catalog relation <cl_cattr> of figure 7A.24 appendix 7A is used to establish joins between relations using OIDs. Equivalent Ingres commands are constructed with implicit joins converted to explicit joins. The constructed query is stored in the Ingres file <instance.f>. A C program <instance.c> is constructed with system calls to Ingres and to the C compiler, and taking input from <instance.f>. The program <instance.c> is compiled and executed.

7.10.6 Abstract data types (ADTs)

7.10.6.1

range of <var> **is** <class_name> **retrieve** <target_list> **where** <adt_qual>

<adt_qual> ::= <spatial_qual> | <geol_time_qual>

<spatial_qual>::= <operand1> <spatial_operator> <operand2>

<spatial_operator>::=

north

south

east

west

overlap

contained_in

abuts_top

abuts_bottom

abuts_right

abuts_left

<geol_time_qual> ::=

<operand1> <geol_time_operator> <operand2>

| <geol_time_operator> (operand1 , operand2)

<geol_time_operator>::=

before

after

between

ADTs like points, rectangles, geological time domain are stored in textual data types. When these commands are parsed the operations over ADTs are translated into primitive operations over a *text* data type. The modified query is stored in an Ingres file <instance.f>. A C file <instance.c> which takes input from <instance.f> is constructed with system calls to Ingres and to the C compiler. The program <instance.c> is compiled and its output is executed.

7.11 Comparsion of OQUEL with related work

In the following section we will compare RM/T[Codd79], OSQL[Beec88], and OQUEL approaches to provide object-orientation from the relational model. We will use the following "golden rules" introduced in [Atki89] for comparisons:-

7.11.1 Complex Objects

Unlike RM/T and OSQL, OQUEL has provided techniques for modeling complex objects and operations for dealing with these objects at different levels of abstraction. We have also modeled objects with unpredictable structure as parameterized QUEL methods which compose objects from different relations. In OQUEL we have simplified operations on complex objects through methods which capture the semantics of complex objects and preserve the integrity of complex objects (e.g. deletion propagation).

7.11.2. Object Identity

All these models support identity of objects which persists through time, independent of properties of objects which may change. This identity is represented by system-generated, unique, immutable object identifiers (*oids*) (e.g. OQUEL, OSQL) or *surrogates* (e.g. RM/T). Objects may be explicitly created and destroyed. In RM/T the database contains one E-relation for each entity type. The E-relation for a given entity type is a unary relation that lists the surrogates for all entities of that type currently existing in the database. The property types for a given type are represented by a set of P-relations. Unlike RM/T, in OQUEL we do not insist on one E-relation plus zero or more separate P-relation(s) for each entity type, but rather allow all of those relations to be collapsed into a single relation.

7.11.3. Encapsulation

The idea of encapsulation comes from (i) the need to cleanly distinguish between the specification and the implementation of an operation and (ii) the need for modularity. There are two views of encapsulation: the programming language view and the database adaptation of that view. The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an interface part and an implementation part. The interface part is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and a procedural part. The data part is the representation or state of the object and the procedure part describes the implementation of each operation. The database translation of the principle is that an object encapsulates both program and data. Encapsulation provides a form of "logical data independence": we can change the implementation of a type without changing any of the programs using that type.

RM/T lacks this concept. In OSQL we have three types of *functions* (*stored, derived, or foreign).* The extension of the *stored functions* corresponds to our relational tables which store the representation of instances of the type. *Derived functions* correspond to our QUEL methods while *foreign functions* correspond to our C++ methods.

7.11.4. Types/Classes

All these models have types or classes. RM/T provides an *entity classification scheme*. Entities are divided into three categories: (1) *Kernel entities:* (i.e. *independent existence* entities), (2) *Characteristic entities:* (i.e. *existence dependent* entities), (3) *Associative entities:* (representing a *many-to-many relationship*). *Kernels* corresponds to our *object classes*, while *characteristics* and

associative entities correspond to our *association* in OQUEL. The primary purpose of entity classification is to impose some structure on the real world, which might otherwise appear to be just an amorphous jumble of facts. A secondary purpose is to introduce some discipline into the integrity enforcement. Both aspects are of major significance in database design.

In OSQL types are used to categorize objects into sets that are capable of participating in a specific set of functions.

7.11.5. Inheritance

Inheritance has two advantages: it is a powerful modeling tool, because it gives a consise and precise description of the world and it helps in factoring out shared specifications and implementations in applications. All these models have type hierarchies. However, in RM/T inheritance is restricted to attributes since methods are not applicable.

7.11.6. Overloading/ Late Binding

In OQUEL, we have access to C++ which supports overloading of functions and operators and late binding. The basic idea behind operator overloading is to redefine a commonly used symbol so that it applies to a new set of values. The capacity to overload operators enhances the extensibility of the language. The idea behind function overloading is that several functions with the same name can represent different pieces of code. Using overloaded functions we can dispense with the sometimes awkward code that is needed to choose one implementation of a function over another and we no longer need to give functions unnatural names.

7.11.7. Extensibility

The ways in which programming languages provide extensibility include allowing the programmer to define new types from scratch (such as enumerated data types), to combine existing types to create new ones, and to define procedures by using procedure and control structures. Databases usually limit extensibility to domain and record definition. Operations are limited to query, insert, delete, and replace. The three update operations deal with only relation or view. In OQUEL, in addition we have provided a mechanism for the definition, storage, and execution of *complex operations* on objects through QUEL and C++ methods. In OSQL foreign functions are provided to perform complex operations while RM/T provided only a predefined set of generic operations.

7.11.8. Computational Completeness

Since QUEL is not computationally complete, there are certain computations that cannot be expressed as QUEL methods. However, C++ methods provide a mechanism for incorporating such computations into the system. While RM/T operations are confined to creation, retrieval and updates, OSQL has included *foreign functions,* similar to our C++ methods, which are written and compiled outside the system.

Chapter 8

Application of OQUEL to a Geological Domain

Chapter 8

Application of OQUEL to a Geological Domain

8.1 Introduction

The integrated interpretation of data used for petroleum exploration requires access to a variety of data. These include: 1) seismic data used to map subsurface formation and depths to these formations exploiting the different acoustic properties of these formations. 2) gravity data used to map subsurface structures using the density contrast property. 3) magnetic data used to delineate subsurface structures based upon magnetic characteristics. 4) stratigraphic data collected from geophysical logs used for correlation, basin evaluation, selection of prospective stratigraphic intervals, as well as areas for detailed study. These data are, usually, presented as maps and cross-sections. Maps and cross-sections help in the following respect: i) visualizing the physical picture of the earth's strata ii) determining local and regional structures iii) determining the origin of rock strata iv) working out rock sequence v) constructing geologic history.

Since the geological data are characterized by a wide range of data types and complex relationships in time and space, the need for a data model with rich semantics and support for abstract data type domains is obvious. In this chapter we will show how OQUEL handles some of the geological data requirements.

8.2 Information hiding

The basic idea in object-oriented models is that the user should not have to wrestle with computer-oriented constructs such as records and fields, but rather should be able to deal with objects and operations that closely resemble their counterparts in the real world. For example, instead of thinking in terms of a DEPT tuple plus a collection of EMP tuples that include foreign key values that reference that DEPT tuple, the user should be able to think directly of a department "object" that actually contains a set of employee "objects". Similarly instead of (e.g.) having to add a tuple into the employee relation with appropriate DEPT# (foreign key) value, the user should be able to create a new employee object and include it in the relevant department object directly. In other words, the fundamental idea is to raise the level of abstraction. The idea of dealing with a database that is made up of "encapsulated objects" (a department object that "knows what it means" to add an employee or to cut the budget), instead of having to understand relations, tuple updates, foreign keys, etc., is naturally much more attractive from the user's point of view.

In OQUEL, we have provided a hybrid object/relational model to provide a flexible environment. The designer of a database will choose one solution or the other according to his needs. On one side he will have inheritance and encapsulation and on the other side easy manipulation of sets using declarative language. Because of encapsulation, we will separate the designer view of the database, that is the classes implementation, from the user view which contains the class interface.

8.2.1 Designer view

The designer creates a class by giving it a name, a type, and a set of methods, by describing its interface and eventually by establishing a hierarchical relationship with other classes.

8.2.2 The user's view

As mentioned before, users are not aware of the object implementation. They will receive information or modify objects via methods made available to them by the designer.

In the geological domain for example, a manager of a petroleum exploration company will view a geological database concerning the exploration activities in terms of meaningful operations as illustrated in figure 8.1.

Geophysical surveys are usually conducted along lines which are composed of segments and those segments are composed of points at which readings are taken. For instance to get the cost of a gravity survey between two points in a survey line (note figure 8.2) , the manager will execute a *get_cost* **method** associated with the line and will be prompted for the relevant parameters and the cost will be calculated.

We use the following relations to illustrate this situation.

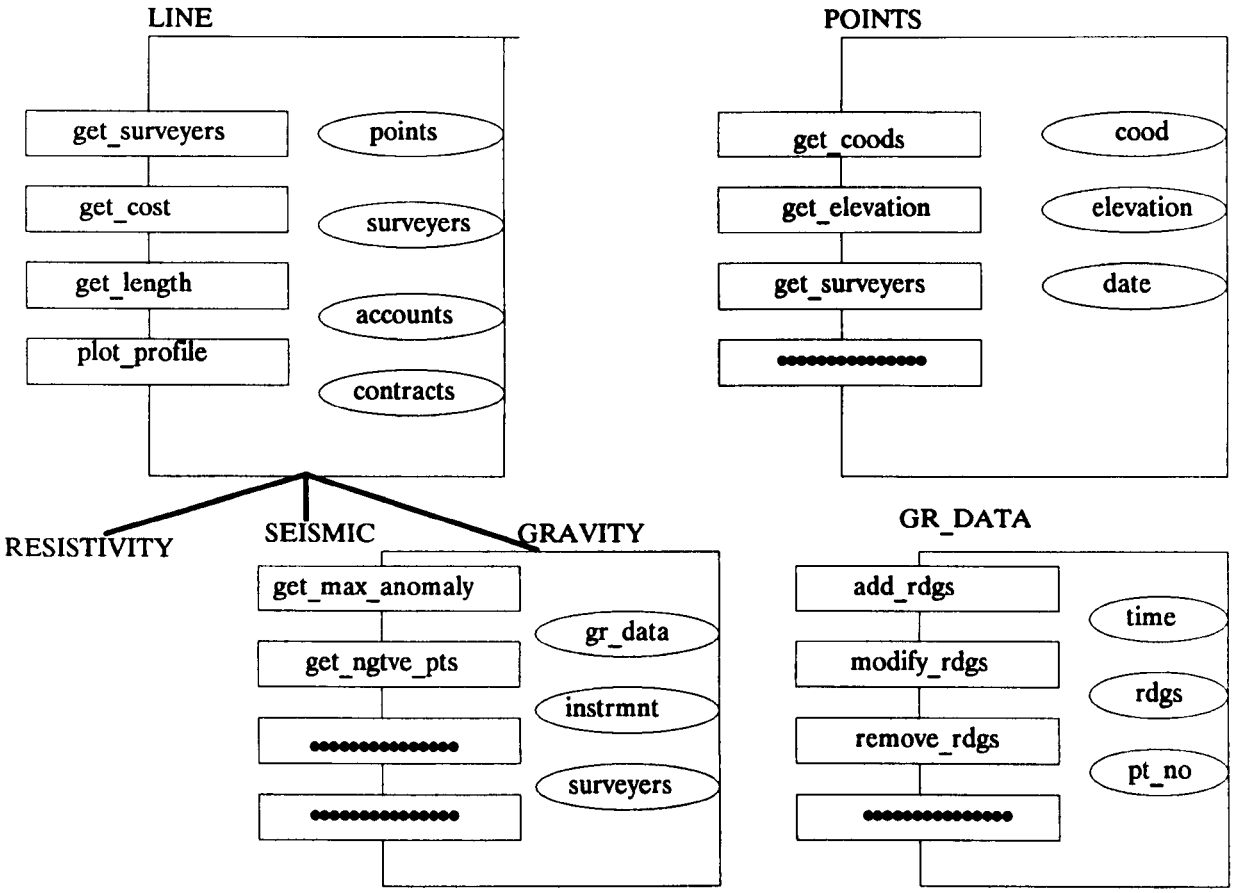


Figure 8.1 An Example of Object Modeling in Geological Domain

distance				
line_no	**from_pt**	**to_pt**	**distance**	**order**
L1	*pt100*	*pt101*	*30.000*	*1*
L1	*pt101*	*pt102*	*45.000*	*2*
L1	*pt102*	*pt103*	*90.000*	*3*
L1	*pt103*	*pt104*	*55.000*	*4*
L1	*pt104*	*pt105*	*35.000*	*5*

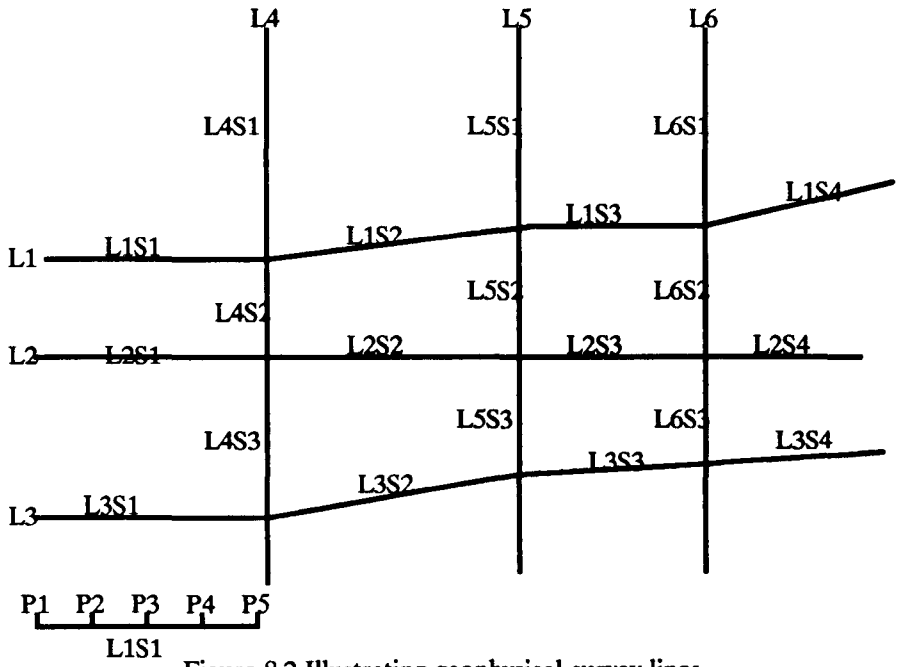


Figure 8.2 Illustrating geophysical survey lines

survey_costs	
type	**price_km**
gravity	*10000.000*
seismic	*20000.000*
resistivity	*15000.000*

Example 8.1

User requirement:

What is the cost of gravity survey between points *pt101* and *pt104* ?

OQUEL query:

execute (line,get_cost,L1);

OQUEL/User Interaction:

OQUEL: (from: to: type-of-survey:) ?

User: *pt101 pt104 gravity*

OQUEL: The total cost is :- *totalcost*

Example 8.2

User requirement:

What is the cost of seismic survey between points *pt102* and *pt104* ?

OQUEL query:

execute (line,get_cost,L1);

OQUEL/User Interaction:

OQUEL: (from: to: type-of-survey:) ?

User: *pt101 pt104 seismic*

OQUEL: The total cost is :- *totalcost*

Since the cost changes with time and survey lines are extended with time, sophisticated methods can be written to compute the actual costs of surveying. The main point here is that all this information is hidden from naive users.

8.3 Temporal Modeling in a Geological Domain

Geological objects are highly complex and embedded in time and space. The time dimension in geological data has two aspects. One is the calendar time which is studied in many temporal models and the other is the geological time. As an illustration of the application of O-O systems, and OQUEL in particular, we give an account in this section of the extension of the work in temporal models to geological time and provide some operators for geological time qualification.

Geological Time Scale			
era	**period**	**beginning of period (m Yrs)**	**ordinal_no**
Quaternary	Holocene	.012	16
	Pleistocene	2	15
Tertiary	Pliocene	7	14
	Miocene	26	13
	Oligocene	38	12
	Eocene	54	11
	Palaeocene	65	10
Mesozoic	Cretaceous	135	09
	Jurassic	195	08
	Triassic	225	07
Upper Palaeozoic	Permian	280	06
	Carboniferous	345	05
	Devonian	395	04
Lower Palaeozoic	Silurian	440	03
	Ordovician	500	02
	Cambrian	570	01
Archaeozoic	Precambrian	4,600	00

geo_history	
sample	**age**
s1	cambrian01
s2	permian06
s3	jurassic08
s4	cretaceous09
s5	carboniferous05
s6	silurian03
s7	pliocene14
s8	pliocene14

Example 8.3

User requirement:

Which samples were formed after *cretaceous* ?

OQUEL query:

range of r **is** geo_history

retrieve (r.sample) **where** r.age **after** *cretaceous*

Answer:-

sample
s7
s8

OQUEL to QUEL Translation:-

range of r **is** geo_history

retrieve (r.sample) **where** right(r.age,2) > 09

Direct QUEL Implementation:-

range of r **is** geo_history

retrieve (r.sample) **where** r.age_code > 09

It is noteworthy that in the direct implementation case the user has to remember the implementation codes compared with more natural OQUEL query.

Example 8.4

User requirement:

Which samples was formed during Mesozoic era ?

OQUEL query:

range of r **is** geo_history

retrieve (r.sample) **where** r.age **between** *devonian* **and** *permian*

Answer:-

sample
s2
s5

OQUEL to QUEL Translation:-

range of r **is** geo_history

retrieve (r.sample) **where** right(r.age,2) > = 04

and right(r.age,2) < = 06

Direct QUEL Implementation:-

range of r **is** geo_history

retrieve (r.sample) **where** r.age_code > = 04

Example 8.5

User requirement:

Which samples were formed before *silurian* ?

OQUEL query:

range of r **is** geo_history

retrieve (r.sample) **where** r.age **before** *silurian*

Answer:-

sample
s1

OQUEL to QUEL Translation:-

range of r **is** geo_history

retrieve (r.sample) **where** right(r.age,2) < 03

Direct QUEL Implementation:-

range of r **is** geo_history

retrieve (r.sample) **where** r.age_code < 03

8.4 Spatial Data in a Geological Domain

Geological samples are collected from the field and their locations are plotted in two dimensional maps. These samples are studied for various properties and characteristics (geophysical, geochemical, etc.) and from macro level to crystal level. Discovering particular properties of a sample a geologist might want to further study samples in the vicinity of that sample or might want to see if there is any alignment with other samples having similar characteristics. Hence he/she needs to pose queries against the data using coordinates qualification. For example a geologist might use the data in the following table to retrieve samples within a given rectangle (note figure 8.3) or located north, south, etc. of a given sample.

geol_samples	
sample_no	**location**
s4	*55,12*
s5	*50,20*
s9	*30,15*
s22	*20,15*

These can accomplished by the following queries:-

Example 8.6

User requirement:

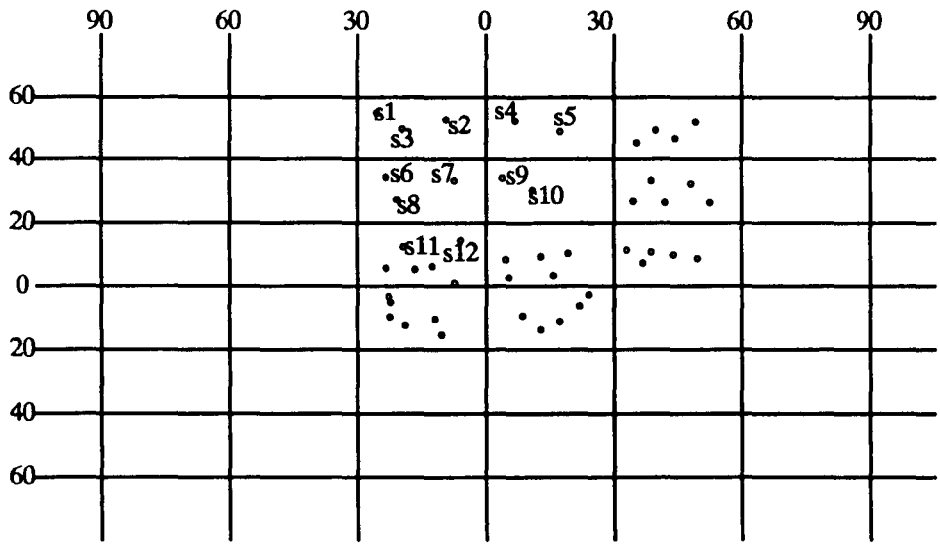


Figure 8.3 Latitude/Longitude Map with Geological Samples

Which geological samples are contained in the grid "10,15,30,30"?

OQUEL query:

range of r **is** geol_samples

retrieve(r.all) **where** r.location **contained_in** "10,15,30,30"

OQUEL to QUEL Translation:-

range of r **is** geol_samples

retrieve (r.all) **where**

left(r.location,2) > = 10 **and**

left(right(r.location,size(r.location)-3),2) > = 15 **and**

left(r.location,2) < = 30 **and**

left(right(r.location,size(r.location)-3),2) < = 30

Direct QUEL Implementation:-

range of r **is** geol_samples

retrieve (r.all) **where**

r.x > = 10 **and** r.x < = 30 **and** r.y > = 15 **and** r.y < = 30

It is worth mentioning that in the direct implementation we have two columns for x and y coordinates while in OQUEL we have one column for x and y coordinates. In addition the OQUEL query is more briefer and user friendly.

Example 8.7

User requirement:

Which samples are located south of latitude 20 ?

OQUEL query:

range of r **is** geol_samples

retrieve(r.all) **where** r.location **south** "30,20"

OQUEL to QUEL Translation:-

range of r **is** geol_samples

retrieve (r.all) **where**

left(r.location,2) < left("30,20",2) **and**

left(right(r.location(r.location,size(r.location)-3,2) = right("30,20",2)

Direct QUEL Implementation:-

range of r **is** geol_samples

retrieve (r.sample) **where**

r.latitude < 30 and r.longitude = 20

Same comments as for example 8.6 are applicable.

8.5 Complex Objects in a Geological Domain

There seem to be two main reasons for using a hierarchical tree structure in a complex geological description. One is the need to specify clearly what is being described and to set it in its geological context. The other is the economy of thought that can be achieved by using the same terms for analogous situations in a variety of contexts. "Medium grained" for instance, has different meanings when applied to a siltstone and to a congolomerate. Properties such as size, shape, orientation, proportion and arrangement can be observed, and can often be described by the same terminology, in entities of a wide range of size and character, such as continents, depositional basins, geological formations, facies, beds, sedimentary structures, fossils, grains, crystals, or even molecules. As can be seen, one of the above entities could be a component of another. Thus grains and fossils might be components of a bed, which in turn was part of a formation and so on.

Clearly, in any description, the object under consideration must be clearly identified and its relation to its components and to the entity of which it is itself a component must be indicated. The statement that an object is rather large, reddish-purple, ellipsoid in a plan view, elongated in a north-south direction, is of little value unless it is known whether the object is a fossil or a basin of deposition. If the latter, it would be important to know which basin it is, and whether rocktypes mentioned elsewhere are part of it.

The following description of a hypothetical rock unit provides an example of a written description showing hierarchical relationships that can be readily structured as a tree.

"The basal six feet of strata comprise sandstone and shale. The sandstone

is crossbedded, medium-grained, somewhat coarser towards the base. It contains occasional, scattered, sub-angular grains of purplish quartz. The shale forms thin partings which contain flattened thin-shelled lamellibranchs". Having specified in the first sentence the set of objects under consideration (the basal six feet of strata), the geologist subsequently considers a class of objects drawn from the set (sandstone), then certain constituents of the sandstone (quartz grains). In the next sentence, another class of objects is considered (shale) drawn from the six feet of strata. Then one of its constituents is mentioned (lamellibranchs), and their attributes described (flattened, thin-shelled).

If a complete measured section is described in such a way that each bed is represented by a tree, the complete description comprises a set of trees (note figure 8.4).

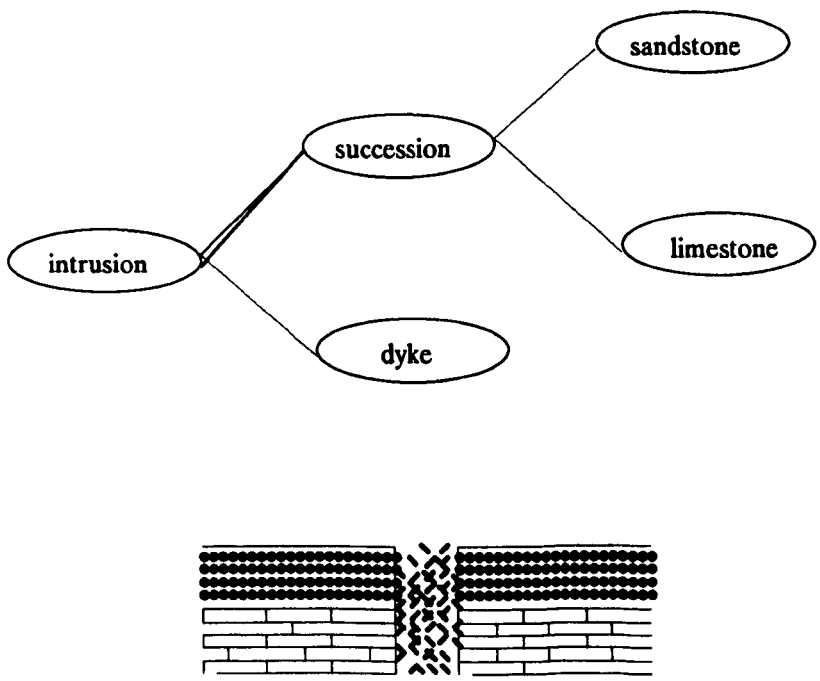


Figure 8.4 Object Modeling of An Intrusion

Written English, which is a string of words, is itself a somewhat artificial medium for representing a tree structure. Various devices are needed to allow

the reader to mentally organize the data, choosing an appropriate structure as the data are presented. Summaries, headings, subheadings, sentences and paragraph construction, and connective words, phrases and sentences may all be required to help the reader in this task.

In OQUEL we have simplified queries against complex objects by allowing joins of objects without explicit join-predicates. For example we can pose the following query against the complex object database in figure 8.5.

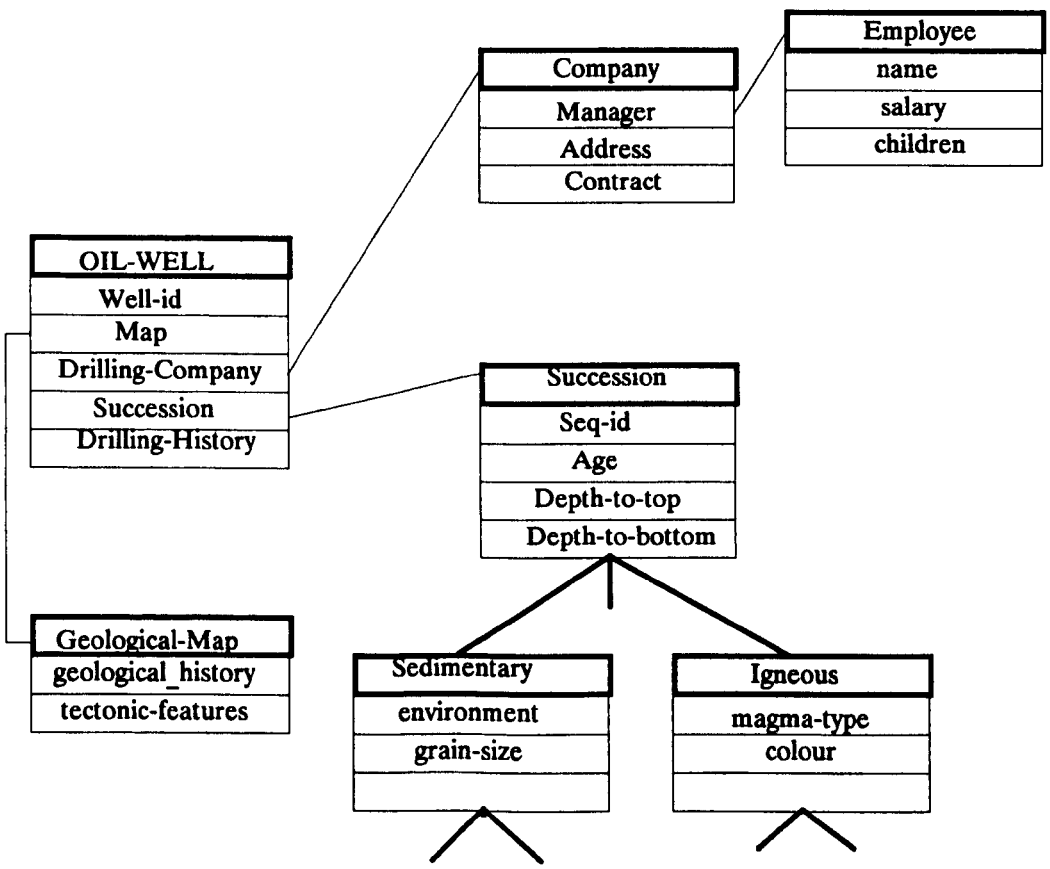


Figure 8.5 Object Modeling of An Oil Well

Example 8.8

User requirement:

Which oil-wells are drilled by a company whose manager salary is more than 11000 ?

OQUEL query:

oil_wells **get** r (r drilling_company manager salary > 11000)

OQUEL to QUEL Translation:-

range of r **is** oil_wells

retrieve (r.oid) **where**

r.drilling_company = company.oid **and** company.manager = employee.oid **and** employee.salary > 11000

Direct QUEL Implementation:-

range of r **is** oil_wells

range of c **is** company

range of e **is** employee

retrieve (r.oid) **where**

r.drilling_company = c.oid **and** c.manager = e.oid **and**

e.salary > 11000

In OQUEL we have shifted the join operations to the system and hence it is possible to express queries against the object-oriented schema more simply.

Figure 8.6 shows an example of a taxonomic hierarchy of tectonic features and figure 8.7 shows its relational mapping.

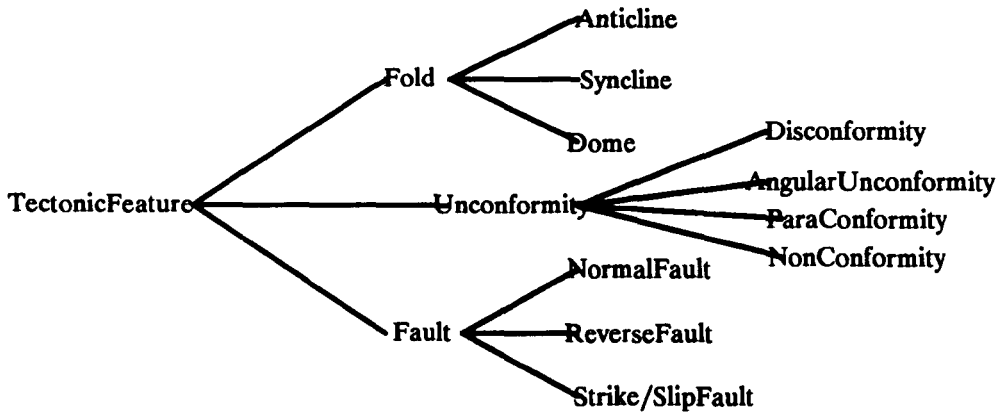


Figure 8.6 Taxonomic Hierarchy of Tectonic Features

Folding

Tect feat id	age	Fold-type	trend	dip
20014	Cretaceous	anticline	N10W	1/4E10N
20015	Carboniferrous	anticline	E12N	1/3N12W

Faulting

Tect feat id	age	Fault-type	direction	displacment
20011	Cretaceous	normal	N22E	200D.SE
20012	Cretaceous	reverse	E42N	140
20017	Triassic	normal	E44N	150

Tectonic-Features

Tect_feat_id	age
20010	Cambrian
20016	Cretaceous
20018	Carboniferrous
20019	Carboniferrous

Figure 8.7 Horizontal Partioning of Tectonic Features Data

From this example it is obvious that there is a need for viewing data at different levels of abstraction and there is a need for a union construct to build such view. For example, to get all tectonic features we need a union of the three horizontally partioned relations in figure 8.7. However, this can also be

achieved by associating a method such as get_all_tectonic_feature at a higher level in the hierarchy to collect tuples from the respective relations.

To conclude this section we believe that the object-oriented approach provides a considerable flexibility for modeling a complex domain such as the geological domain, and the examples given in this chapter, although they are simple, illustrate how OQUEL provides capabilities to model and query data in such a complex domain.

Chapter 9

Conclusions and Directions for Further Research

Chapter 9

Conclusions and Directions for Further Research

9.1 OQUEL

Neither the relational model nor the object-oriented model is suitable for all database applications. We have proposed a hybrid object-relational model (OQUEL) that employs the object-oriented paradigm concepts such as objects, classes, methods, inheritance, and object-identity. Some previous extensions of the relational model such as complex objects, derived attributes, abstract data types, and generalization are also supported in our model. OQUEL is engineered from an existing relational system (Ingres), the C language and its object-oriented extension (C++). OQUEL shows that it is possible to offer the benefits of an object model without sacrificing the facilities of the current relational model. OQUEL has the following features:-

* It extends the data structures and operations of the relational data model and provides the desirable features of the OOPL paradigm and semantic data models such as improved semantics, data abstraction, reusability of data structure and code, extensibility, complex object support, schema evolution and ADT domains.

* OQUEL introduces the concept of an object identifier (OID) to improve the semantics and reduce the space required to store the database. OID captures the uniqueness of entities in the real world and allows for modeling complex objects. An OID may be used to refer to an object instead of copying it.

* The built-in support of object-identity and domains in the conceptual object model simplifies queries because most join operations are replaced by simpler path traversals. However, these joins are still performed by the

underlying relational system because OQUEL queries are transformed into pure relational implementation. Using indexing techniques on OIDs will enhance these join operations.

* OQUEL provides two access modes for the database. One is through the relational interface which helps to formulate unpredictable queries and provide flexibility in the selection of a target list through different combinations of attributes - this is due to the no-information-goal-dependency of the relational formalism. The other is a method-based interface which improves reliability by providing semantic integrity, managing complex objects and propagating updates through different semantic references, providing information hiding and operations for naive users.

* OQUEL provides structuring for complex objects through specialized attributes and provides operations through methods to enforce the integrity of the complex objects. Abstracting complex objects through tuples and relations provides for organisation of data by object or relation.

* OQUEL extends the domain of the relational model by making provision for abstract data types (ADTs) which simplify queries and provide a more natural interface for spatial and temporal data. This and the next points are considered in more detail in the next section.

* OQUEL provides operations to manipulate the evolution of structure as well as content.

* OQUEL extends the relational model by inheritance to allow for sharing of data structures and operations and this improves the productivity of the programmer.

* We have also provided an interface between C++ and Ingres to provide freedom for the implementation of any required methods.

9.1.1 Abstract Data Types and Methods

Traditional relational database systems are not able to support objects such as coods and boxes, and hybrid scientific units such as the "feet inches" system as atomic objects. However, ADTs provide the capability to extend the basic types of the database system with user-defined data types and related operations. In OQUEL, we considered the introduction of ADTs in a relational context from two orthogonal points of view : (a) granularity of types:- ADTs granularity (relation / domain), (b) the description of operations:- language to implement ADT operations (database language / OOPL with data access).

In OQUEL, an object-oriented database can be viewed basically as a relational database in which tuples contain not only primitive data values, but also methods and pointers to other objects. However, for efficiency reasons, methods applicable to the relation instances are physically grouped together into separate catalog relation. Thus viewing a relation as an ADT provides a high level of abstraction, data protection, and integrity.

The operations to manipulate objects are expressed into two languages: (1) QUEL, and (2) C++ with embedded QUEL commands. These approaches have the following limitations:-

QUEL methods:- QUEL methods are stored in a long field of type text. These methods have to be retrieved from the database and the formal parameters have to be replaced by the actual parameters and the resultant query is sent back to Ingres for execution. Although we have accessed Ingres more than once QUEL methods can be optomized by the dbms since they are expressed in pure relational language.

C++ methods:- Relational query languages restrict operations that can be invoked on stored data to selection, updates, joins and projections. Complex data specific operations need to be decomposed into these simple operations,

but this is not always possible since a query language is not computationally complete. In OQUEL, objects can be associated with class-specific methods that are written in C++ with Ingres access and which can be invoked by users to perform complex operations on objects. The cost of this approach is that Ingres is accessed more than once. Firstly to retrieve the path name of the C++ code. Secodly to retrieve the data associated with the C++ method. In the case of an update Ingres is accessed once more. Since C++ is operating at tuple-at-a-time level we are still having the impedance-mismatch problem.

In OQUEL, we have tried a feasible way of implementing ADTs at the domain level on top of existing DBMS. To achieve this we have extended the QUEL dialect to accommodate new operators. These operators are transformed through query modification to equivalent Ingres expressions. Performance improvement from the use of ADTs at the domain level can be realized from from the following:- (1) manipulation of a smaller number of columns. For example, a rectangle can be retrieved as a single column rather than four constituent parts. (2) Simplification of queries due to the introduction of new operators. This is especially noticeable in spatial and temporal windowing.

In the case in which the operations on an ADT domain cannot be formulated in terms of query language expressions we manipulate such domains through C++ methods.

9.2 Relationship of Object-Oriented Models to other Data Models

Object-oriented models have relationships to other data models and the application of the object-oriented paradigm to databases has a significant impact on the architecture of database systems. These aspects are reviewed in the following sections.

9.2.1 Hierarchical and Network Models

There are at least two types of similarity between object-oriented and hierarchical or network databases. One important similarity is the nested structure of objects in object-oriented databases, and the nested structure of records in hierarchical databases. However, although both databases admit objects (records) which refer to other objects (records) for the values of their attributes, there is an important difference. The nested object schema in object-oriented databases contains cycles; although hierarchical databases can admit cycles, they require artificial record types to be introduced in the schema. Another similarity is between the object identifiers in object-oriented databases and the record pointers in hierarchical databases. However, an object-identifier is a logical pointer and is never reused, and as such may be used for enforcing referential integrity. A record pointer is a physical pointer and is reused. However, there are major differences that distinguish object-oriented databases from hierarchical and network databases. Object-oriented databases support such concepts as a class hierarchy, inheritance, and methods; hierarchical and network databases obviously do not include these concepts.

9.2.2 Functional Models

There have been attempts to construct database systems based on functions instead of relations, and a driving force behind such attempts was to address some of the shortcomings of the original relational model. The functional model shares certain ideas with the OO approach, including its navigational style of addressing objects that are functionally related to other objects. However, this compromises the encapsulation objectives.

The function in the functional model is typically not a mathematical function at all; for example it may be allowed to return multiple values. In fact

considerable violence has to be done to the function in order to make it capable of doing all things needed in the functional data model context.

9.2.3 Semantics Models

The basic idea behind semantic modeling is to try to identify a set of constructs that seem to be generically useful, in the sense that they recur in some form or shape in a wide variety of applications. If we can identify and formalize such constructs, and operations for them, we can build them into the database systems and make them more intelligent.

The object-oriented approach to intelligence differs. The intelligence resides not in generically understood constructs, but rather in encapsulated objects that some technical specialist has had to define. When we say that a department object "knows" what it means to add an employee, we mean that some specialist has written code to perform the operation.

9.2.4 Nested Relations

The basic idea here is to increase the utility and functionality of the relational model by dropping the requirement that the relations be normalized. These models address the shortcomings of the relational models and hence share the same objectives of the object-oriented approach. However, they differ in that they have a rigorous mathematical theoretical foundation which is lacked by the object-oriented approach at the present time. Unfortunately, they also lack the concept of inheritance which is fundamental in the object-oriented approach.

9.2.5 Extensible Systems

It will never be possible to provide built-in features that will directly support everything that users will ever want to do; hence, there is a need to allow users to tailor the system in various ways by defining their own data types, functions, access methods, storage structure and so forth. And, of course, the objective of the object-oriented approach is to allow the users to build their own tailored systems. If a database system is implemented in an object-oriented style or in an object-oriented programming language, then it tends to make it easier to add new database functionality than if it is implemented in a conventional programming style. The notion of inheritance is what makes systems implemented in object-oriented style potentially extensible.

9.3 The Impact of Object-Oriented Concepts on the Architecture of a Database System

9.3.1 Generalization Hierarchy and Aggregation Hierarchy

The generalization and aggregation hierarchy relationships which are inherent in object-oriented data models but which do not exist in relational database management systems have required a reexamination of a number of architectural concepts including schema evolution, query languages, indexing, storage structures, and authorization.

9.3.1.1 Schema Evolution

The database schema for an object-oriented database has two dimensions. One dimension is the class hierarchy which captures the generalization relationships between a class and its subclasses. Another dimension is the class composition hierarchy which represents the aggregation relationships between a class

and its attributes and the domains of the attributes. In other words, every class in an object-oriented database simultaneously belongs somewhere in the class hierarchy and somewhere in the composition hierarchy. The semantics of the class hierarchy is what complicates schema changes. For example, when a class is dropped, all its subclasses will lose the attributes and methods they had inherited from the class, and therefore instances of the subclasses will lose the values of the attributes. Furthermore, when a new class is added the class will inherit attributes and methods from existing classes specified as its superclasses, and will also provide attributes and methods for its subclasses to inherit from it. The class hierarchy also gives rise to a number of meaningful schema changes beyond those possible under the relational model. For example, making an existing class a new superclass of another existing class is a meaningful operation.

9.3.1.2 Queries

Because of the nested structure of the definition of a class, one may use the nested relational model as the starting point for defining a query model for object-oriented databases. However, the theory of nested relations is inadequate as a model of queries for object-oriented databases. There are a few important reasons for this. First, the definition of a class may form a directed acyclic graph, while the nested relational model deals with a strict hierarchy of relations. Second, the current theory of nested relations has not taken into account some of the object-oriented concepts.

Object-oriented systems model every real-world entity as an object with a unique identifier; the object belongs to a class, and a class has a position somewhere in the class hierarchy. These simple principles impose some difficult constraints on the query model for object-oriented databases. In relational

databases, the result of a query is itself a relation. This situation is not so simple for object-oriented databases, because of the nested definition of classes.

Despite the differences in the data models, object-oriented queries may be evaluated on top of relational systems as illustrated by OQUEL.

9.3.1.3 Storage Structures

Relational database systems represent the database schema in the form of a set of relations, including a relation for all other relations in the database, a relation for all columns of each relation, and so on. It is more difficult to represent and maintain the schema of an object-oriented database since the schema is no longer a simple collection of largely independent relations, but a collection of classes which are interrelated to one another through the generalization and aggregation relationships.

9.3.1.4 Authorization

The concepts of inheritance and encapsulation introduce new dimensions to the authorization model. For example, who is allowed to inherit from which classes ? Who is allowed to access which methods ?

9.4 Integration of Programming Languages and Database Systems

Object-oriented programming techniques are rapidly gaining support for applications where conventional procedural programming paradigms limit development. The benefits are straightforward: Object-oriented programming reduces the cost and time required to produce complex applications by supporting the development of reusable code, which is easier to maintain and enhance.

Implementing systems using the object-oriented conceptual model results in: (1) type specific representation , (2) insulation of the application from

changes (encapsulation), (3) guaranteed consistency between the operations executed on objects of a given type by different applications, (4) dramatic reduction in the cost of developing new applications because of code inheritance from predefined object types.

The combined notions of a class, attributes, and a class hierarchy mean that the semantic data modeling concepts instance-of, aggregation, and generalization are inherent in the object-oriented paradigm. This means that the gap between applications implemented in an object-oriented programming language and an object-oriented database is much narrower than that between an object-oriented application and a non-object-oriented database. In particular, one of the problems with implementing object-oriented applications on top of a relational database system is that a relational system does not directly support a class hierarchy and the nested definition of a class, and as such the application programmer must map these constructs to relations.

However, there is another type of impedance mismatch caused by the record-at-a-time orientation of OO languages which is a throwback to the days of prerelational systems. The solution to this problem is to introduce set-level facilities into programming languages.

9.5 Information Hiding

Information hiding is unquestionably a good idea in many cases: The twin concepts of (a) concealing irrelevant details from the user (thus allowing those details to be changed, when necessary, in a controlled and comparatively painless manner), and (b) providing disciplined access to objects through a public interface only, are clearly appropriate for many users and many applications. Thus, it would seem desirable to extend existing relational DBMSs, to support such facilities. However, there will always be a need to access data in

unforeseen ways for the purposes of ad hoc query, and thus the notion of only being able to operate via predefined methods is not acceptable in some situations. OO systems tend to be too rigid in this regard. OQUEL proposes two modes of access. (1) A Method-based access in which an object responds according to its procedural and declarative knowledge, and (2) A relational-based access supported by Ingres.

OO systems and languages, and therefore object-oriented concepts, have been developed largely independently of any consideration of very large databases; that is, they have assumed that all objects reside in a large virtual memory. This means that object identifiers have been used as the sole means of specifying desired objects; the notion of a query for selecting an arbitrary set of objects that satisfy an arbitrary combination of search predicates has been an alien concept to the designers of OO languages. In OQUEL, we have proposed relaxation of encapsulation to provide visible attributes and hence content-based selection.

9.6 Complex Objects

It might be true that the object-oriented approach eliminates the "need to normalize" in the sense that it directly supports unnormalized objects. However, it does not follow that it automatically eliminates the problems that unnormalized objects cause. Normalization is still needed unless we are prepared to tolerate bad database design.

The object-oriented approach represents "complex objects" as hierarchies, but not all such objects actually have a hierarchic structure in the real world. The object-oriented approach has difficulties over many-to-many relationships. In fact, all the old arguments made against the hierarchic approach in the 1970s can now be made again. In particular, hierarchic structures tend to simplify

some applications at the expense of others.

OQUEL proposes two approaches to complex objects. One approach is to store abstract data types such as rectangles in a relation attribute of type "text", and to provide operations for their manipulations and to extend the query predicates for qualifications. The other approach is to build complex objects on top of relations, not including them within relations as attribute values. This is achieved by using parameterized methods written in QUEL and stored in Ingres.

9.7 Directions for Further Research

Object-Oriented databases are still a fertile ground for research. We now offer some thoughts on future research directions.

9.7.1 Nested Relation Model + Inheritance

There is a clear need to formalize or at least standardize object-oriented concepts if a true foundation for object-oriented databases is to be laid. The notions of inheritance and queries are topics for future theoretical research. In view of the fact that the importance of an object-oriented approach is founded on the reusability and extensibility it offers, it is obvious that more research should be directed to the notions of inheritance. The proposals on query languages for the nested relational model should be extended to account for class hierarchy or inheritance.

9.7.2 Database Design Tools

The richness of an object-oriented data model is a mixed blessing. On one hand, it makes it easier for the users to model their applications. On the other hand, the complexity of an object-oriented schema significantly complicates the problem of logical and physical database design. Thus the need for friendly and efficient design aids for the logical and physical design of object-oriented databases is significantly stronger than that for relational databases. There is the problem of clustering object-oriented databases, and the need for class-hierarchy indexes to expedite the evaluation of object-oriented queries. It might be useful to extend the storage structures of non-first-normal-form (NF2) relational databases for the physical design of object-oriented databases.

9.7.3 Semantics Modeling

The work in temporal data models has been mainly on extending the relational data model to support the time dimension of information. Hence, a number of temporal query languages has been proposed (e.g. TQUEL, TSQL). It is interesting to investigate the temporal extensions to the relational model in object-oriented context and to develop object query languages with temporal support.

Incomplete information has been studied in the relational model and different types of nulls has been identified. It is worth to reconsider these in the object-oriented models which support aggregation and generalization and to see how inheritance eliminates some types of nulls (i.e. inapplicable).

9.8 Closing Remark

Object-orientation from the relational model is viable and gives an evolutionary approach to relational database systems. However, many issues should be reconsidered by introducing object-oriented extensions to the relational databases. Both the relational view and the object-oriented view have their natural applications. A system which supports both views gives its users a wider selection which leads to a better match between database problems and an appropriate implementation.

References

[AbHu87]

Abiteboul, S. and Hull, R. 1987. IFO: A formal semantic database model. *ACM Trans. Database Systems,* 12(4): 525-565.

[Abri74]

Abrial, J.R. Data Semantics. In *Data Management Systems.* J.W. Klimbie and K.L. Koffeman, eds., North Holland, 1974.

[Adib88]

Adiba, M., C. Collet. Managing Complex Objects as Dynamic Forms *VLDB 1988, Los Angeles.*

[Ahn 86]

Ahn, I. Towards an implementation of database management systems with temporal support. In *Proc. of the Intl. Conf. on Data Engineering* (Los Angeles, Calif. Feb.). IEEE Press, New York, 1986, pp. 374-381.

[Ande82]

Anderson, T. L., 1982. Modeling Time at the Conceptual Level. in *Proc. Second Int'l Conf. Databases,* June 1982.

[AnHa87]

Andrews, T., and Harris, C. 1987. Combining Language and Database Advances in an Object-Oriented Development. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages and Applications.* (Oct.), pp. 430-440.

[Aria86]

Ariav, G. A Temporally Oriented Data Model. *ACM Trans. Database Systems.* 11,4 (Dec.86) pp 499-527.

[Astr76]

Astrahan, M. et al. System R: A relational Approach to Data Base Management, *TODS,* 1:2, June 1976.

[AtBu87]

Atkinson, M. P. and Buneman, O. P. 1987. Database Programming Languages, *ACM Computing Survey 19(2)* :105-190

[Atki89]

Atkinson, M. P. et al. 1989. The Object-Oriented Database System Manifesto. In *Proc. of the First Deductive and Object-Oriented Database Conference.* Kyoto Japan, (Dec.) 1989.

[Bach77]

Bachman, C.W. and Dayal, M. The Role Concepts in Data Models. In *VLDB 77.*

[Banc88]

Bancilhon et al. 1988. The Design and Implementation of O2, an Object-Oriented Database System, In *Proc. of the OODBS II Workshop, Bad Munster, Sept. 1988.*

[Bane87a]

Banerjee, J., Chou, H. T., Gazra, J., Kim, W., Woelk, D., Ballou, N. and Kim, H. J. 1987 Data Model Issues for Object-Oriented Applications., *ACM TOIS, (Jan.) 1987.*

[Bane87b]

Banerjee, J., W. Kim, H Kim., and Korth H. 1987: Semantics and Implementation of Schema Evolution in Object-Oriented databases, in *SIGMOD[1987].*

[Bato84]

Batory, D. and Buchmann, A. 1984: Molecular Objects, Abstract Data Types, and Data Models: A Framework, in *VLDB[1984]*

[Bato85]

Batory, D. S., and Kim, W. 1985. Modeling Concepts for VLSI CAD Objects. *ACM Trans. Database Systems,* 10(3): 322-346.

[Bato88]

Batory, D. S., Barnett, J. R., Garza, F. F., Smith, K. P., Tsukauda, K., Twichell, B. C. and Wise, T. E. 1988 GENESIS: A Reconfigurable Database Management System. *IEEE Trans. Soft. Eng.*

[Beec88]

Beech, D., A Foundation for Evolution from Relational to Object Databases, in *Advances in Database Technology - EDBT 1988,* Lecture Notes in Computer Science 303, J.W. Schmidt, S. Ceri and M. Missikoff, eds. Springer-Verlag, 1988.

[Belk86]

Belkhouche, B., and Urban, J. May 1986. Direct Implementation of Abstract Data Types From Abstract Specifications. *IEEE Transactions on Software Engineering* vol. SE-12 (5).

[Berk88]

Berkel, T., P. Klahold, G. Schlageter, W. Wilkes. Modelling CAD-Objects by Abstraction *Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, June 1988.*

[Bloo87]

Bloom, T., S.B. Zdonik Issues in the Design of Object-Oriented Database Programming Languages. *OOPSLA'87*

[Booc86]

Booch, G. February 1986. Object-Oriented Development. *IEEE Transaction on Software Engineering vol. 12(2).*

[Borg85]

Borgida, A. 1985 Feature of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software* 2(1): 63-72.

[Brod81a]

Brodie, M. L., 1981. On Modeling Behavioral Semantics of Database. in *Proc. Seventh Int'l Conf. Very Large Databases,* 1981.

[Brod81b]

Brodie, M. L., and Zilles, S. (eds.), *Proc. of the Pingree Park Workshop on Data Abstraction, Databases and Conceptual Modeling,* Joint SIGART, SIGMOD, SIGPLAN Newsletter, ACM New York, Jan. 1981.

[Brod84]

Brodie, M. L., Mylopoulos, J. and Schmidt, J. W. Eds. 1984. *On Conceptual Modeling* Springer-Verlag, New York.

[Brod86]

Brodie, M. L. and J. Mylopoulos, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies,* Spriger-Verlag, New York, 1986.

[Bube77]

Bubenko, J.A., The Temporal Dimension in Information Modeling, In *Architecture and Models in Database Management Systems,* G.M. Nijseen(ed), North-Holland Amsterdam, 1977, 93-118.

[Card85]

Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys, vol. 17(4), p. 475.*

[Care88]

Carey, M. et al. 1988. A Data Model and Query Language for EXODUS, in *Proc. 1988 ACM SIGMOD Conf. on Management of Data,* Chicago, IL. June 1988.

[Chan82]

Chan, A., et al., Storage and Access Structures to Support a Semantic Data Model, *8th VLDB, 1982.*

[Chan83]

Chan, A., Dayal, U., Fox, S., and Ries, D. 1983. Supporting a semantic data model in a distributed database system. In *Proc. of the 9th Int'l Conf. on VLDB.* Very Large Database Endowment, Saratoga, Calif., pp. 354-363.

[Chen76]

Chen, P. P., 1976. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems,* 1: 9-36.

[Clif83]

Clifford, J. and Warren, D.S., Formal Semantics for Time in Databases, *ACM TODS* 6:2 June 1983.

[Codd70]

Codd, E. F., 1970. A Relational Model for Large Shared Data Banks, *Communications of the ACM,* Vol. 13, No. 6,(Jun. 1970), pp. 377-387.

[Codd79]

Codd, E. F., 1979. Extending the Database Relational Model To Capture More Meaning. *ACM Transactions on Database Systems,*
4:397-434.

[Cope84]

Copeland, G. and Maier, D., Making Smalltalk a Database System, In *Proc. 1984 ACM SIGMOD Conf. on Management of Data,* Boston, MA, June 1984.

[Cox 86]

Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach.* Reading, MA:

Addison-Wesley.

[Dada86]

Dadam, P. et al., A DBMS Prototype to Support NF2 relations, In *Proc. 1986 ACM SIGMOD Conf. on Management of Data,* Washington, DC. 1986.

[Date81]

Date, C.J. Referential Integrity. In *Proc of the 7th Intl. Conf. on VLDB 1981.*

[Date86]

Date, C. 1986: *An Introduction to Database Systems,* Vol.1 Fourth Edition, Addison-Wesley, 1986.

[Daya88]

Dayal, U., A.B. Buchmann, D.R. McCarthy. Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System. in *[Ditt88]*

[Derr85]

Derrett, N., Kent, W., and Lynbaek, P. 1985. Some Aspects of Operations in an Object-Oriented Database. *IEEE Database Eng. Bull.* 8,4(Dec).

[Ditt86]

Dittrich, K. R. and Dayal, U. (eds): *Proc. of the 1986 Intl. Workshop on Object-Oriented Database Systems,* IEEE Computer Science Press.

[Ditt87]

Dittrich, K. R., W. Gotthard and P. C. Lochemann, DAMOKLES--A Database System for Software Engineering Environments, Proc. IFIP Workshop on Advanced Programming Environments, R. Conradi, T.M. Didriksen and D.H. Wanvik(eds.), Trondheim, Norway, June 1986, *Lecture Notes in Computer Science 244,* Springer-Verlag.

[Ditt88]

Dittrich, K. R. (ed): *Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science,* Vol. 334, Springer-Verlag., 1988.

[Douq75]

Douque, B.C.M. and Nijssen, G.M. (eds.), *Database Description,* North Holland, 1975.

[ECOOP'88]

Proceedings of ECOOP'88: European Conference on Object-Oriented Programming. August 1988. New York, NY: Springer-Verlag.

[EDS 84]

EDS [1984] *Expert Database Systems,* Kerschberg, L. (editor), (Proceedings of the First International Workshop on Expert Database Systems, Kiawah Island, South Carolina, Oct.84), Benjamin/Cummings, 1986.

[EDS 86]

EDS [1986] *Expert Database Systems,* Kerschberg, L. (editor), (Proceeding of the First International Conference on Expert Database Systems, Charleston, South Carolina, 1986), Benjamin/Cummings, 1987.

[EDS 88]

EDS [1988] *Expert Database Systems,* Kerschberg, L. (editor), (Proceeding of the Second International Conference on Expert Database Systems, Tysons Corner, Virginia April 1988), Bejamin/Cummings, 1989.

[Elma85]

Elmasri, R. et al. The Category Concept: An Extension to the Entity-Relationship Model. *Intl. Jour. on Data and Knowledge Engineering,* 1:1, May 1985.

[Fish86]

Fishman, D. et al., IRIS: An Object-Oriented Database Management System, *ACM TOIS* 5:1, pp 48-69, (Jan.) 86.

[Fram85]

Framer, D. B., King, R. and Myers, D. A. 1985. The Semantic Database Constructor. *IEEE Trans. Softw. Eng.* SE-11, 7, 583-591.

[Gadi88]

Gadia, S. 1988: A Homogeneous Relational Model and Query Language for Temporal Databases, *TODS,*13:4,Dec.1988.

[Gibb83]

Gibbs, S. and Tsichritzis, D. A Data Modeling Approach for Office Information Systems. *ACM TOOIS,* vol. No.4, Oct. 1983.

[Gold83]

Goldberg, A.: *Smalltalk-80, the language and its implementation.* Addison-Wesley, 1983.

[Gray88]

Gray, P.M.D. Expert Systems and Object-Oriented Databases: Evolving A New Software Architecture. in *[EDS 88]*

[Gutt80]

Guttag, J. 1980. Abstract Data Types and the Development of Data Structures, in *Programming Language Design* New York, NY: Computer Society Press.

[Hamm81]

Hammer, M. and McLeod, D., 1981. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems,* 6(3): 351-386.

[Hask82]

Haskin, R. and Lorie, R.[1982]: On Extending the Functions of a Relational Database System, in *SIGMOD[1982].*

[Hayn81]

Haynie, M. The Relational/Network Hybrid Data Model for Design Automation Databases. *Proc. 18th Design Automation Conf.* ACM IEEE New York 1981.

[Hitc87]

Hitchcock, P. A Database View of the PCTE and ASPECT, pp. 37-49 in *Software Engineering Environments,* (ed.) P. Brereton, Ellis Horwood 1987.

[Huds86]

Hudson, S. E., and King, R. 1986. CACTIS: A Database System for Specifying Functionally-Defined Data. In *Proc. of the Workshop on Object-Oriented Databases* (Asilomar, Pacific Grove, Calif., Sept.). IEEE, New York.

[Huds87]

Hudson, S. E., and King, R. 1987. Object-Oriented Database Support for Software Environments. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data* (San Francisco, Calif., May). ACM, New York, pp. 491-503.

[Hull87]

Hull, R. and King, R. 1987. Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys,* 19:3, Sep. 1987.

[Imie81]

Imielinski, T. and Lipski, W. 1981: On Representing Incomplete Information in a Relational Database, in *VLDB 1981.*

[Jaga89]

Jagadish, H.V. and L. O'Gorman An Object Model for Image Recognition. *Computer 1989*

[Karl88]

Karl, S. and P.C. Lockemann Design of Engineering Databases: A Case For More Varied Semantic Modelling Concepts. *Information Systems Vol. 13, No. 4, pp. 335-357, 1988.*

[Katz85]

Katz, R.H. Information Management for Engineering Design, *Spriger-Verlag,* 1985.

[Kemp87]

Kemper, A., Lockemann, P., and Wallrath, M. 1987. An Object-Oriented Databse System for Engineering Applications, in *SIGMOD 1987.*

[Khos86]

Khoshafian, S. N., and Copeland, G. P. 1986. Object Identity. In *proc. of the Conf. on Object-Oriented Programming Systems Languages and Applications* (Portland, Oreg. Sep.) ACM New York, pp. 404-416.

[Kim85]

Kim, W., D. Reiner, and D. Batory (eds.), *Query Processing in Database Systems,* Springer-Verlag, 1985.

[Kim88]

Kim, W. et al. 1988. Integrating an Object-Oriented Programming System with a Database System. in *Proc. OOPSLA'88 Conf.*, Los Angeles, CA, Sep. 1988.

[Kim 89]

Kim, W., and Lochovsky, K. 1989. *Object-Oriented Concepts, Databases, and Applications.* Reading, MA: Addison-Wesley.

[King82]

King, R., and McLeod, D. 1982. The Event Database Specification Model. In *Proc. of the 2nd. Intl. Conf. on Databases: Improving Usability and Responsiveness.* pp. 299-321.

[King84]

King, R. 1984. Sembase: A Semantic DBMS. In *Proc. of the First Intl. Workshop on Expert Database Systems* (Oct.) Univ. of Southern Carolina, Columbia, S.C., pp. 151-171.

[Kort84]

Korth, H. et. al., System/U : A Database System Based on the Universal Relation Assumption. *ACM-TODS,* Sep. 1984.

[Kulk86]

Kulkarni, K. G. and Atkinson, M.P. 1986. EFDM: Extended Functional Data Model, *Computer Journal,* vol.29, No.1.

[Lars83]

Larson, J. 1983. Bridging the Gap Between Network and Relational Database Management Systems, *IEEE Computer,* 16:9, Sep.1983.

[Lecl88]

Lecluse, C., Richard, P., and Velez, F., O2 An Object-Oriented Data Model, In *Proc. ACM SIGMOD Conf.,* Chicago, IL. 1988.

[Lecl89]

Lecluse, C. and Richard, P., The OO Database Programming Languages, In *Proc. 15th VLDB Conf.,* Amsterdam, Aug. 1989.

[Lohm83]

Lohman, G. et al., Remotely Senses Geophysical Databases: Experience and Implications for Generalized DBMS, *Proc. 1983 ACM-SIGMOD.*

[Lori83]

Lorie, R. and W. Plouffe. Complex Objects and Their Use in design Transactions, *Database Week,* San Jose, Calif. May 1983.

[Lori85]

Lorie, R. and W. Kim, D. McNabb, W. Plouffe, A. Meier. Supporting Complex Objects in a Relational System for Engineering Databases. In Kim W. and D. Reiner D. Batory (eds): *Query Processing in Database Systems* Springer-Verlag 1985.

[Lum85]

Lum, V. et. al., Design of an Integrated DBMS to Support Advanced Applications, *Proc. Intl.*

Conf. on Foundations of Data Organization. Kyoto Univ. Japan, May 1985.

[Lyng86]

Lyngbaek, P., and Kent, W. 1986. A Data Modeling Methodology for the Design and Implementation of Information In *Intl. Workshop on OODBS* (Asilomar, Pacific Grove, Calif. Sept.). IEEE, New York.

[Lyng87]

Lyngbaek, P. and Vianu, V. 1987. Mapping a Semantic Database Model to the Relational Model. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (San Francisco, Calif., June). ACM, New York.

[MacG85]

MacGregor, R. M. 1985. ARIEL- A Semantic Front-End to Relational DBMSs in *Proc. of the 11th Intl. Conf. on Very Large Data Bases.* Saratoga, Calif., pp.305-315.

[Maie83]

Maier, D. *The Theory of Relational Databases.* Pitman, London 1983.

[Maie86]

Maier, D., Stien, J., Otis, A., and Purdy, A. 1986. Development of an Object-Oriented DBMS. In *Proc. of the Conf. on OOPSLA* (Sep. 29-Oct. 2). ACM, New York, pp.472-482.

[Mano86]

Manola, F., and Dayal, U. 1986. PDM: An Object-Oriented Data Model. In *Proc. of the Workshop on Object-Oriented Databases* (Pacific Grove, Calif., Sept. 23-26). IEEE, New York, pp. 18-25.

[Metaxides]

Metaxides, A., discussion on pp. 181 of [Douq75].

[Meye88]

Meyer, B. 1988. *Object-Oriented Software Construction* New York, NY: Prentice Hall.

[Mins75]

Minsky, M. A Framework for Representing Knowledge. In P.H. Winston (eds). *The Psychology of Computer Vision* New York, McGraw-Hill.

[Moha88]

Mohan L. and R. L. Kashyap An Object-Oriented Knowledge Representation for Spatial Information.

IEEE Software Engineering 1988

[Mylo84]

Mylopoulous, J., Levesque, H. J.: An Overview of Knowledge Representation. In *[Brod81b].*

[Mylo85]

Mylopoulos, J., and Brodie, M.L. AI and Databases: Semantic vs. Computational Theories of Information, in Ariav G., Clifford J. (eds.), *New Directions for Database Systems,* Norwood, NJ: Ablex Publ. 1985.

[Nava88a]

Navathe, S. and Ahmed, R. 1988. A Temporal Relational Model and Query Language, *Information Sciences.*

[Nava88b]

Navathe, S. and Pillalamarri, M. 1988. Toward Making the ER Approach Object-Oriented, in *ER Conference 1988.*

[Nixo87]

Nixon, B., L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos and M. Stanley, Implementation of a Compiler for a Semantic Data Model: Experiences with TAXIS, *Proc. ACM SIGMOD 87.*

[O'Br86]

O'Brien, P., P. Bullis and C. Schaffert, Persistent and Shared Objects in Trellis/Owl. in *[Ditt86]*

[OOPSLA'86]

Proceedings of OOPSLA'86: Object-Oriented Programming Systems, Languages, and Applications. November 1986.

[OOPSLA'87]

Proceedings of OOPSLA'87: Object-Oriented Programming Systems, Languages, and Applications. October 1987.

[OOPSLA'88]

Proceedings of OOPSLA'88: Object-Oriented Programming Systems, Languages, and Applications. September 1988.

[OOPSLA'89]

Proceedings of OOPSLA'89: Object-Oriented Programming Systems, Languages, and Applications. October 1989.

[Osbo86]

Osborne, S. and Heaven, T., The Design of a Relational System with Abstract Data Types As Domains, *ACM TODS,* Sep. 1986.

[Pate90]

Patel-Schneider, P.F. Practical, Object-Based Knowledge Representation For Knowledge-Based Systems

[Pato88]

Paton, N.W. and P.M.D. Gray An Object-Oriented Database for the Storage and Analysis of Protein Structure Data. In *Prolog and Databases: Implementations and New Directions.* (eds.) P.M.D. Gray and R.J. Lucas Ellis Horwood, 1988.

[Pete87]

Peterson, G. ed. 1987. *Object-Oriented Computing Concepts.* New York, NY: Computer Society Press of the IEEE.

[Pist86]

Pistor, P. and F. Anderson, Designing a Generalized NF2 Data Model with SQL-Type Language Interface. *Proc. VLDB,* pp. 278-285, Kyoto, Japan (Aug. 1986).

[Rowe79]

Rowe, L., and Schoens, K. A. 1979. Data Abstractions, Views, and Updates in RIGEL. In *Ingres Papers* by Stonebraker et al.

[Rowe87]

Rowe, L. and Stonebraker, M., The POSTGRES Data Model, In *Proc. 1987 VLDB Conf.,* Brighton, England, Sep. 1987.

[Rumb87]

Rumbaugh, J. October 1987. Relations as Semantic Constructs in an Object-Oriented Language. *SIGPLAN Notices vol. 22(12).*

[Rumb88]

Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology.

Communications of the ACM, vol. 31(4).

[Sche86]

Scheck H.J., and M. Scholl: The Relational Model with Relation-Valued attributes, *Information Systems 1986,* vol.11, No.2, 1986, pp.137-147

[Sche82]

Scheuermann, P. (editor) 1982. *Improving Database Usability and Responsiveness,* Academic Press, 1982.

[Schl88]

Schlageter, G., R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl, and R. Meyer. OOPS-An Object-Oriented Programming System with Integrated Data Management facility. *Proc. IEEE 4th Int'l Conf. Data Engineering,* Los Angeles, California, Feb. 1988, 118-125.

[Schr87]

Schriver, B., and P. Wegner. *Research Directions in Object-Oreinted Programming.* Cambridge, MA: The MIT Press.

[Shav81]

Shave, M.J.R. Entities, Functions and Binary Relations : Steps to a Conceptual Schema, *Computer Journal, 24(1) 1981.*

[Sheu88]

Sheu, P.C. VLSI Design with Object-Oriented Knowledge Bases, *Computer-Aided Design 1988.*

[Ship81]

Shipman, D. W., 1981. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems,* 6(1): 140-173.

[SmSm77]

Smith, J. M. and Smith, D. C. P., 1977. Database abstractions : Aggregation and Generalization. *ACM Transactions on Database Systems,* 2(2): 105-133.

[SmZd87]

Smith, K. and Zdonik, S. Intermedia: A Case Study of the Differences between Relational and Object-Oriented Database Systems. In *Proc. OOPSLA 87.*

[Snod85]

Snodgrass, R. and Ahn, I., A Taxonomy of Time in Databases, *Proc 1985 ACM-SIGMOD.*

[Stal86]

Staley, S.M. and Anderson, D.C. Functional Specification for CAD Databases. *Computer-Aided Design* 18, pp. 132-138 (1986).

[Stef86]

Stefik, M., and Bobrow, D., Object-Oriented Programming: Themes and Variations. *AI Magazine* 6:4(1986),40-62.

[Ston83a]

Stonebraker, M., Ong, J., and Fogg, D. Implementation of Data Abstraction in the Relational Database System Ingres. *SIGMOD Record,* ACM, New York 1983, pp. 107-113.

[Ston83b]

Stonebraker, M. Document Processing in a Relational Database System. *ACM TOOIS,* April 1983.

[Ston84]

Stonebraker, M. et al. Quel as a Data Type, *Proc. ACM SIGMOD Conf.,* Boston, Mass., June 1984, pp. 208-214

[Ston86a]

Stonebraker, M.(editor0 1986. *The Ingres Papers,* Addison-Wesley, 1986.

[Ston86b]

Stonebraker, M. et al. Application of Abstract Data Type and Abstract Indices to CAD Database. in *Ingres Papers.*

[Ston86c]

Stonebraker, M. and Rowe, L. A. 1986. The Design of Postgres. In *Proc. of Intl. Conf. on the Management of Data* (May). ACM, New York, pp. 340-355.

[Ston86d]

Stonebraker, M., Inclusion of New Types in Relational Database Systems, In *Proc. 2nd Intl. Conf. on Data Engineering,* Los Angeles, CA, Feb. 1986.

[Ston87]

Stonebraker, M., J. Anton, E. Hanson: Extending a Database System with Procedures, *ACM*

Trans. on Database Systems, vol.12, No.3, Sep. 1987, pp.350-376

[Ston88]

Stonebraker, M. et al. The POSTGRES Rules System, *IEEE Trans. Software Eng.* July 1988.

[Stro86]

Stroustrup, B. 1986. *The C++ Programming Language.* Reading, MA: Addison-Wesley.

[Stro88]

Stroustrup, B. May 1988. What is Object-Oriented Programming? *IEEE Software, vol. 5(3),p.10.*

[Su 83]

Su, S. A Semantic Association Model for Corporate and Scientific-Statistical Databases. *Information Science,* 29, 1983.

[Su 86]

Su, S. Y. W., 1986. Modeling Integrated Manufacturing Data with SAM*. *IEEE Computer,* 19(1): 34-49.

[Tans86]

Tansel, A. U. Adding time dimension to relational model and extending relational algebra. *Information systems.*

[Teor86]

Teorey, T. J., Yang, D., and Fry, J. P. 1986. A Logical Design Methodology for Relational databses Using the Extended Entity Relationship Model. *ACM Computing Survey* 18(2): 197-222.

[Thom89]

Thomas, D. March 1989. What's an Object?. *Byte, vol.14(3).*

[Tsic76]

Tsichritzis, D.C., and Lochvsky, F.H., 1976. Hierarchical Database Management Systems, *ACM Computing Surveys,* vol. 8. No.1 March.

[Tsur84]

Tsur, S. and Zaniolo, C. 1984. An Implementation of GEM-Supporting a Semantic Data Model on a Relational Back-End. In *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data.* ACM, New York, pp. 286-295.

[Ullm85]

Ullman, J. Implementation of Logical Query Languages for Databases. *ACM-TODS* Sep. 1985

[Ullm88]

Ullman, J.D., *Principles of Database and Knowledge-Base Systems,* Volume I, Computer Science Press, Potomac, MD, 1988.

[Urba86]

Urban, S. D., and Delcambre, L. M. L. 1986. An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In *Proc. of the 2nd IEEE Intl. Conf. on Data Engineering* (Feb.) IEEE, New York, pp. 382-389.

[Vald86]

Valduriez, P., Khoshafian, S. and Copeland, G. 1986. Implementation Techniques of Complex Objects. *VLDB'12* Kyoto Japan.

[Wegn87]

Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22(12).

[Wied84]

Wiederhold, G. 1984. *Knowledge and Database Management,* IEEE Software, Jan. 1984.

[Wied86]

Wiederhold, G. Views, Objects, and Databases, *IEEE Computer,* 19, no. 12, (Dec. 86)

[Woel86]

Woelk, D., W. Kim and W. Luther, An Object-Oriented Approach to Multimedia Databases, *ACM SIGMOD, 1986.*

[Zani83]

Zaniolo, C. 1983. The Database Language GEM. In *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data.* ACM, New York, pp. 207-217.

[Zani84]

Zaniolo, C. Database Relations with Null values. In *Journal of Computer and System Sciences.* vol. 28 No.1 1984 pp. 142-166.

Appendix 7A

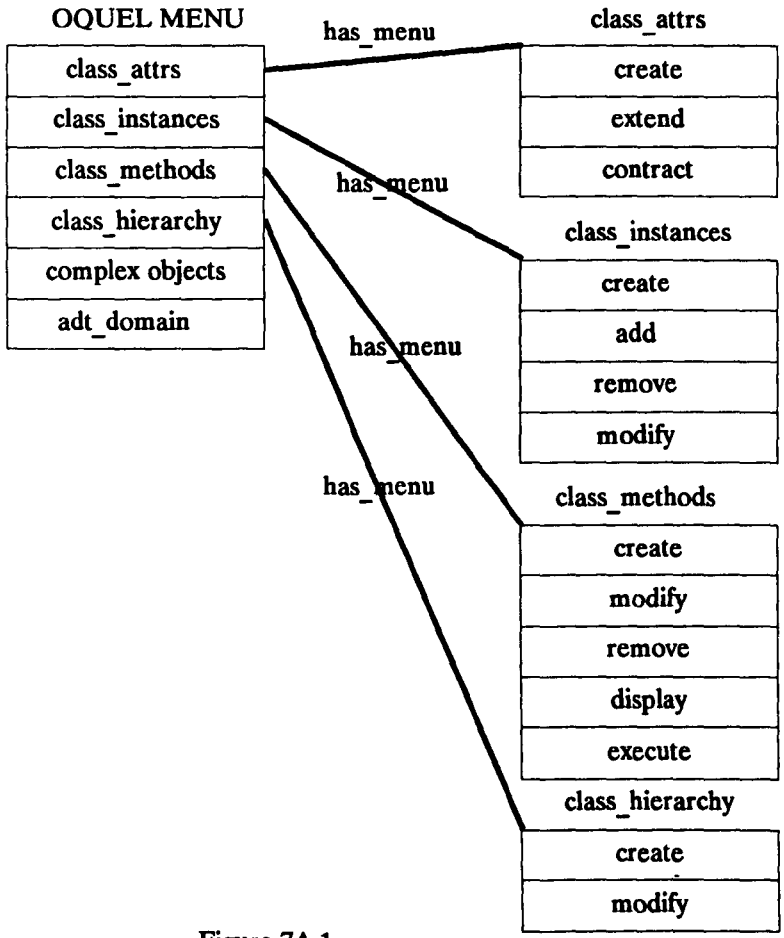


Figure 7A.1

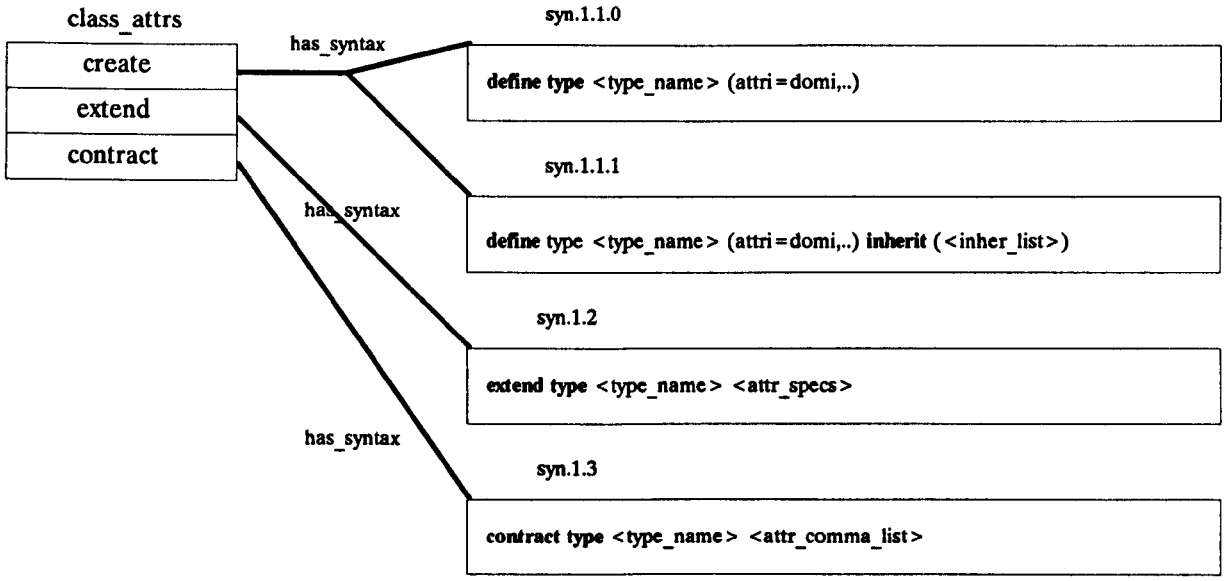

class_attrs
create
extend
contract
has_syntax
has_syntax
has_syntax
syn.1.1.0
define type <type_name> (attri=domi,..)
syn.1.1.1
define type <type_name> (attri=domi,..) inherit (<inher_list>)
syn.1.2
extend type <type_name> <attr_specs>
syn.1.3
contract type <type_name> <attr_comma_list>

Figure 7A.2

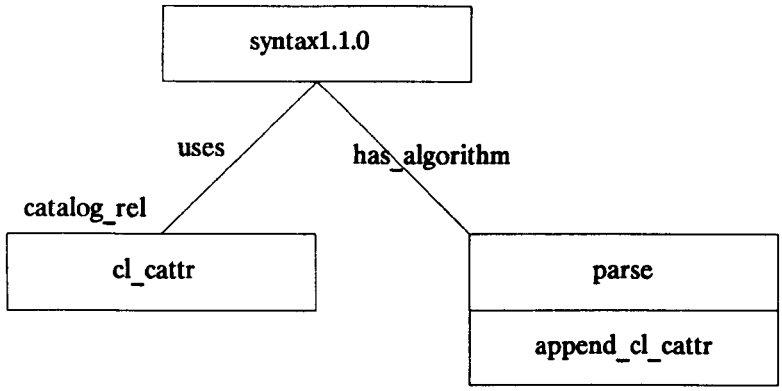


Figure 7A.3

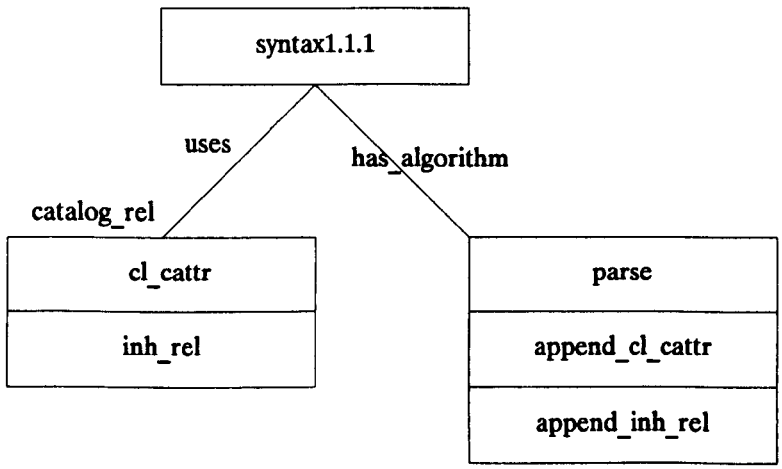


Figure 7A.4

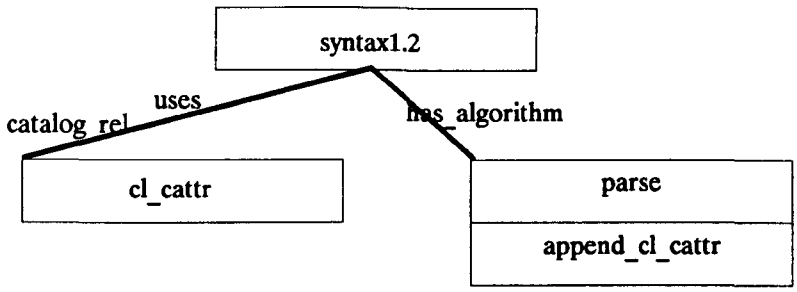


Figure 7A.5

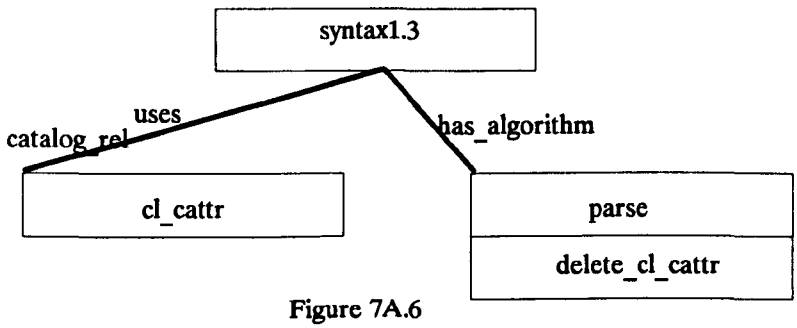


Figure 7A.6

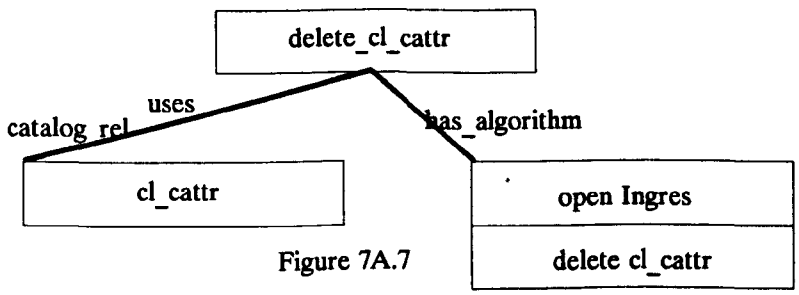


Figure 7A.7

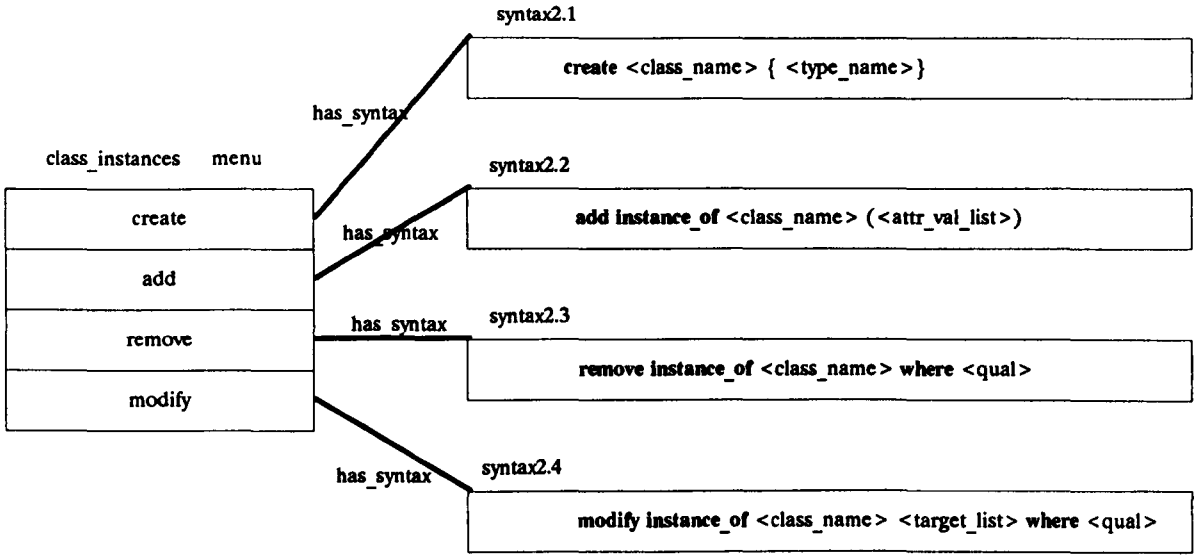


Figure 7A.8

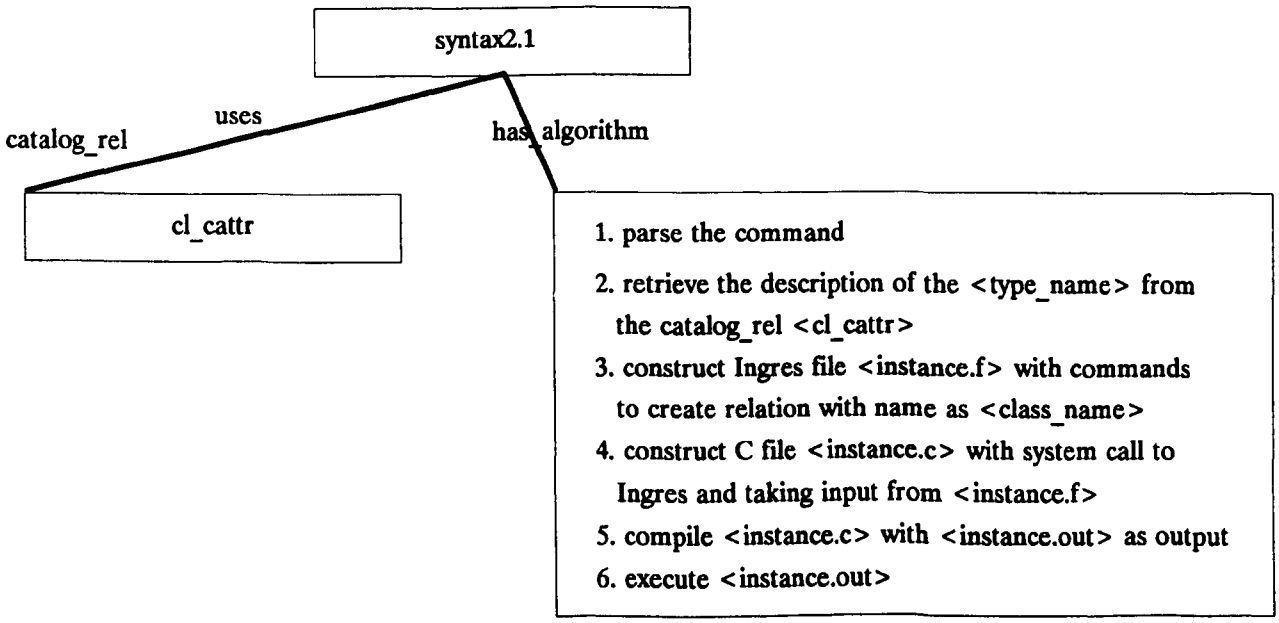


Figure 7A.9

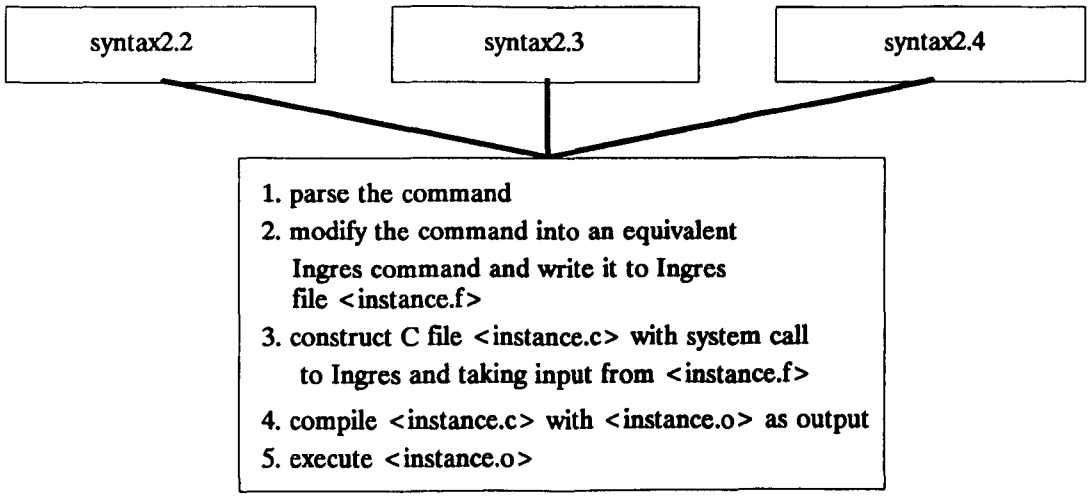

syntax2.2
syntax2.3
syntax2.4
1. parse the command
2. modify the command into an equivalent
Ingres command and write it to Ingres
file <instance.f>
3. construct C file <instance.c> with system call
to Ingres and taking input from <instance.f>
4. compile <instance.c> with <instance.o> as output
5. execute <instance.o>

Figure 7A.10

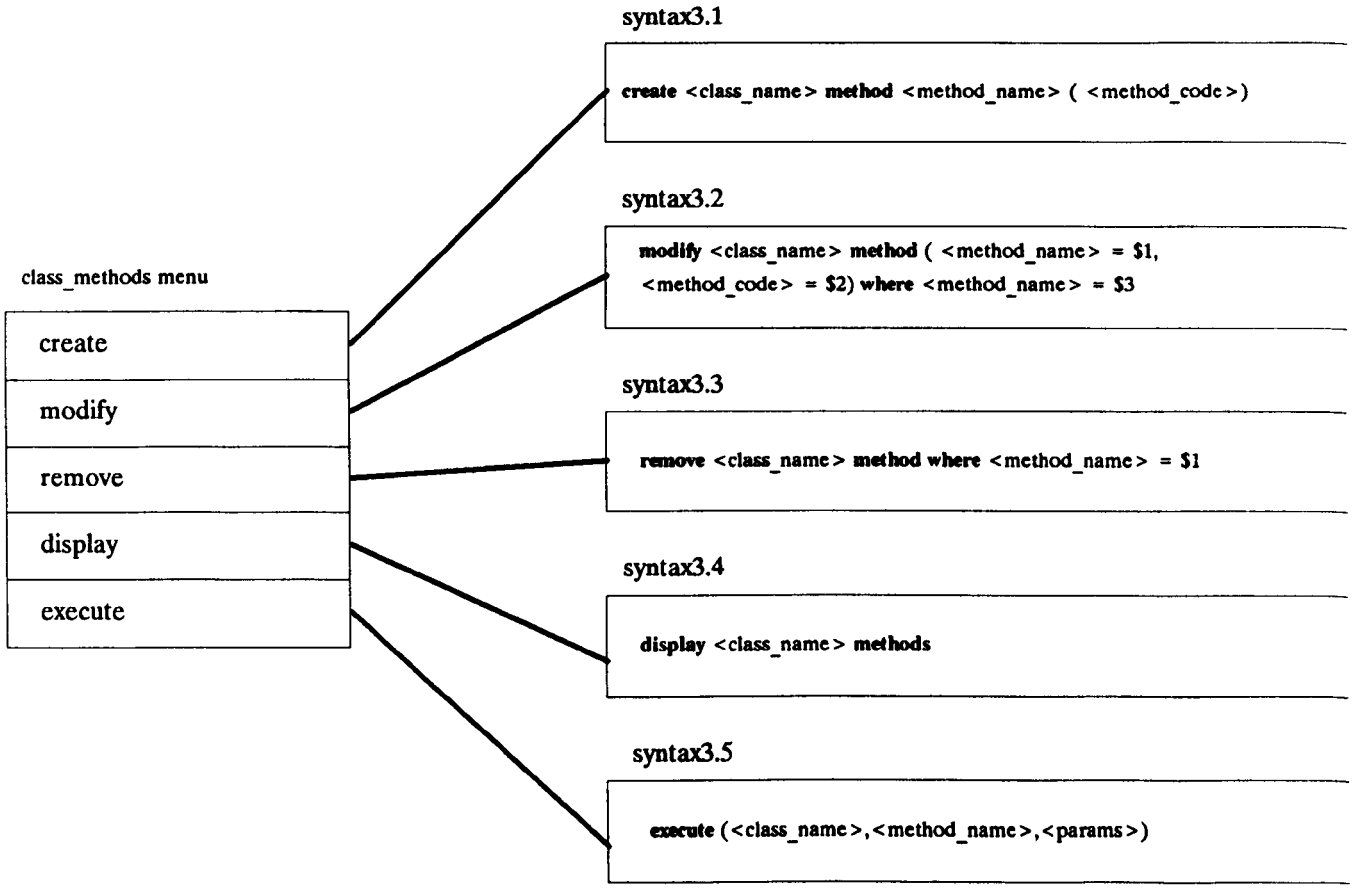

class_methods menu
create
modify
remove
display
execute
syntax3.1
create <class_name> method <method_name> (<method_code>)
syntax3.2
modify <class_name> method (<method_name> = $1,
<method_code> = $2) where <method_name> = $3
syntax3.3
remove <class_name> method where <method_name> = $1
syntax3.4
display <class_name> methods
syntax3.5
execute (<class_name>,<method_name>,<params>)

Figure 7A.11

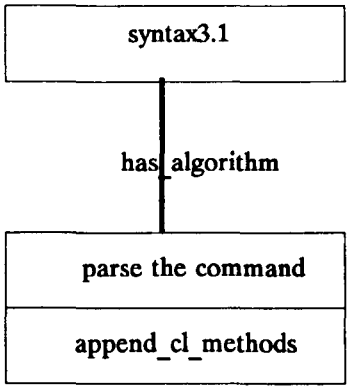


Figure 7A.12

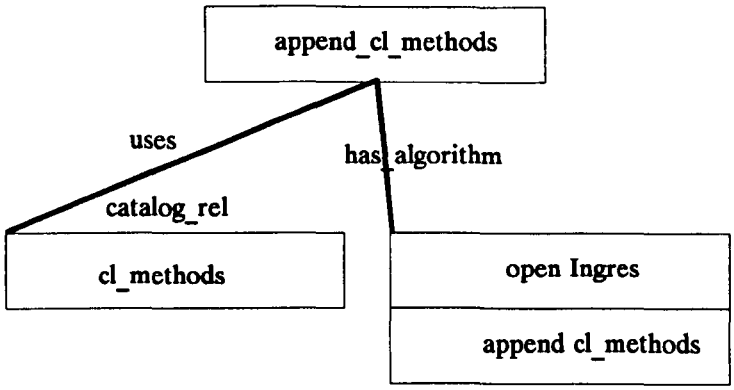


Figure 7A.13

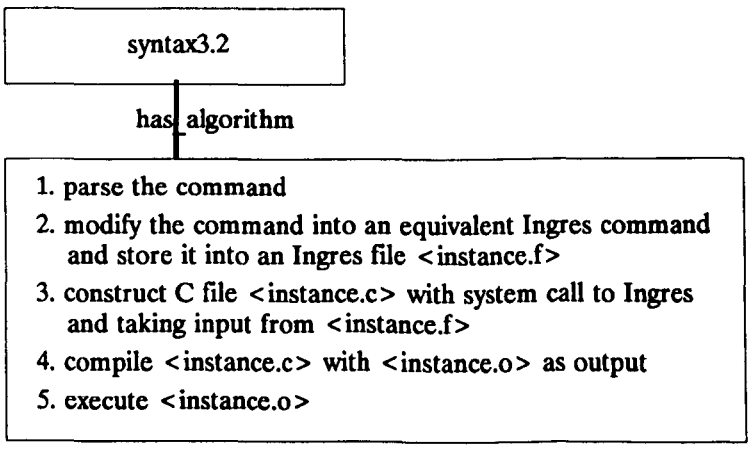


Figure 7A.14

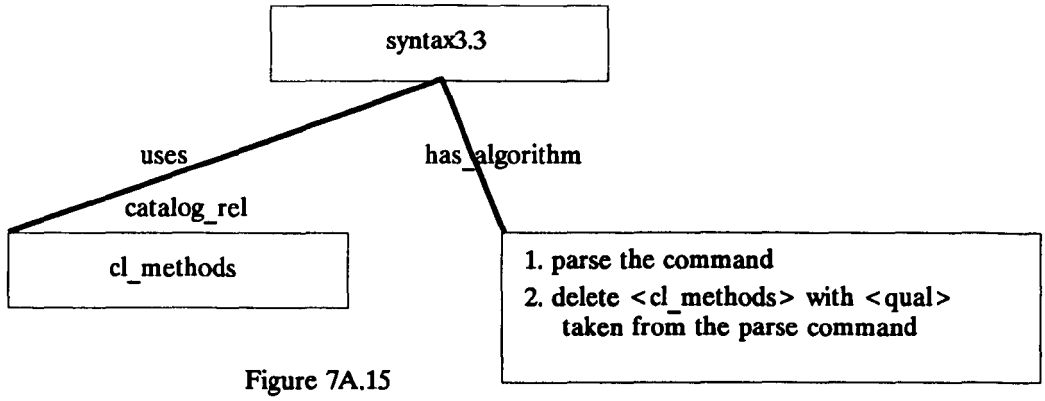


Figure 7A.15

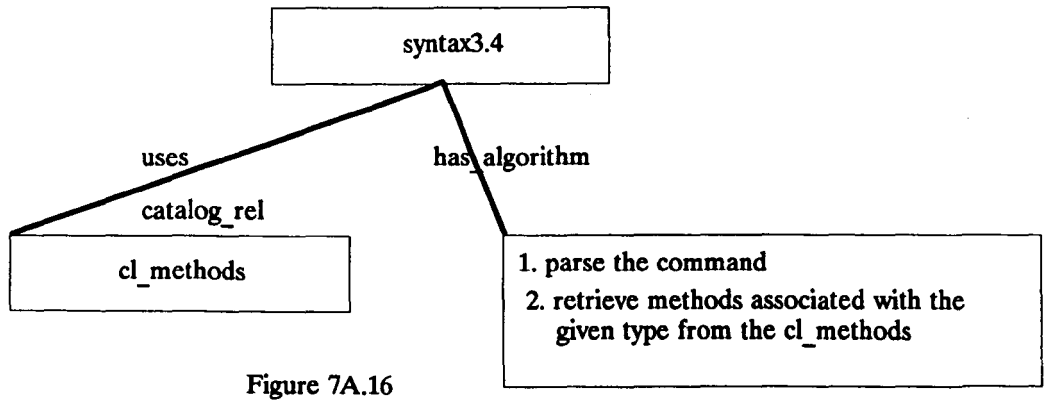


Figure 7A.16

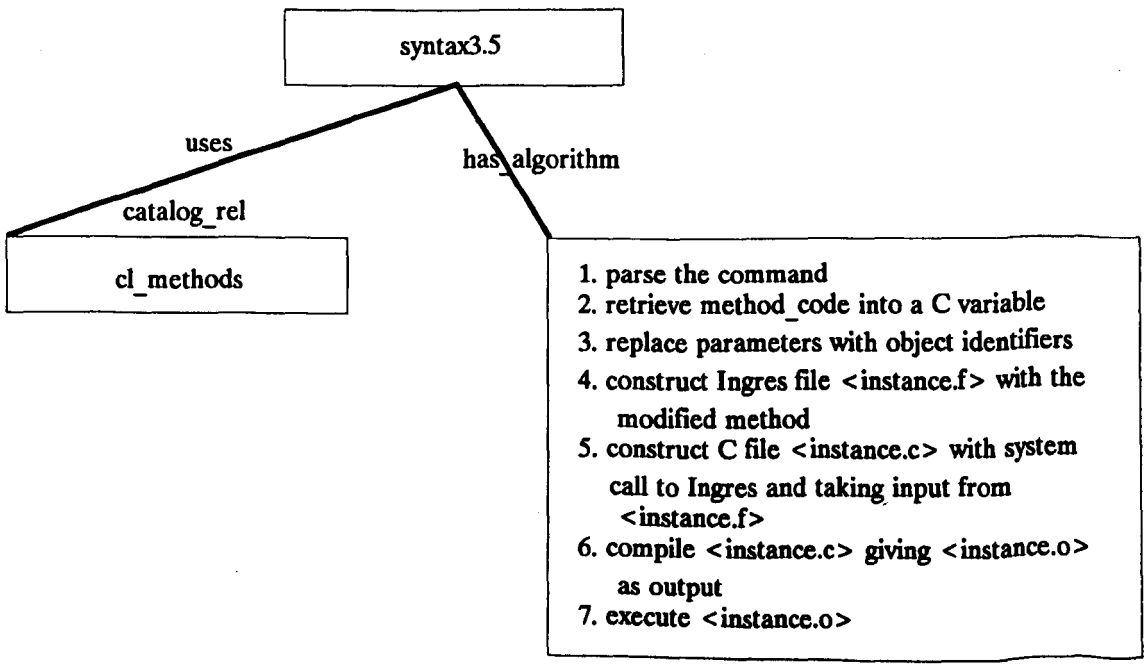


Figure 7A.17

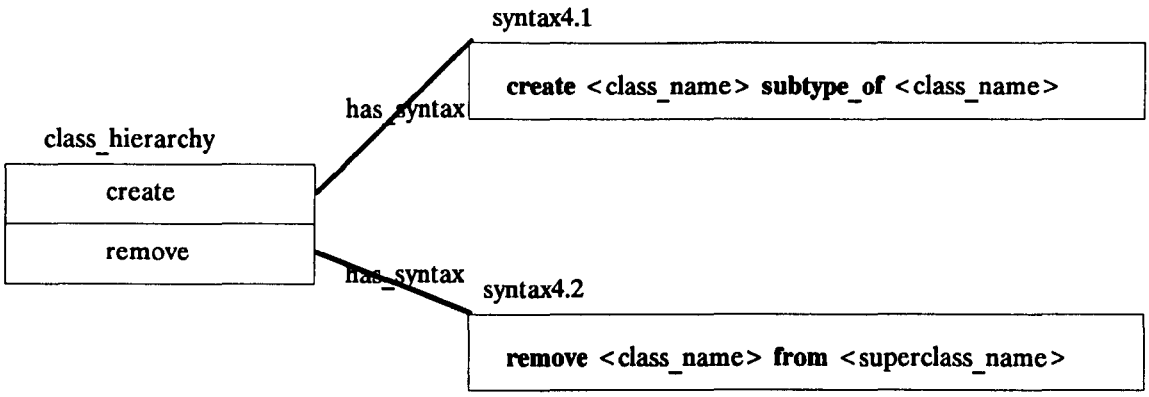


Figure 7A.18

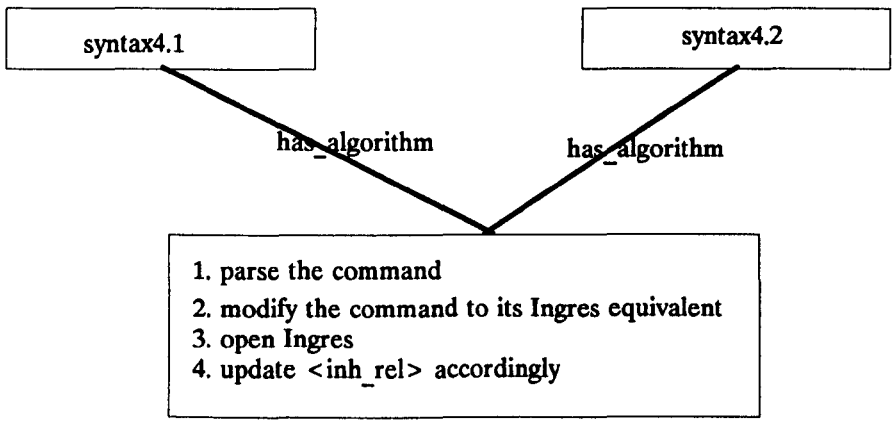


Figure 7A.19

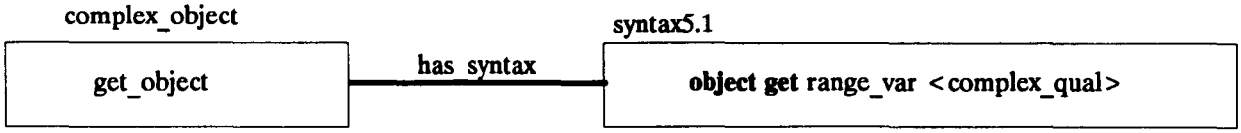


Figure 7A.20

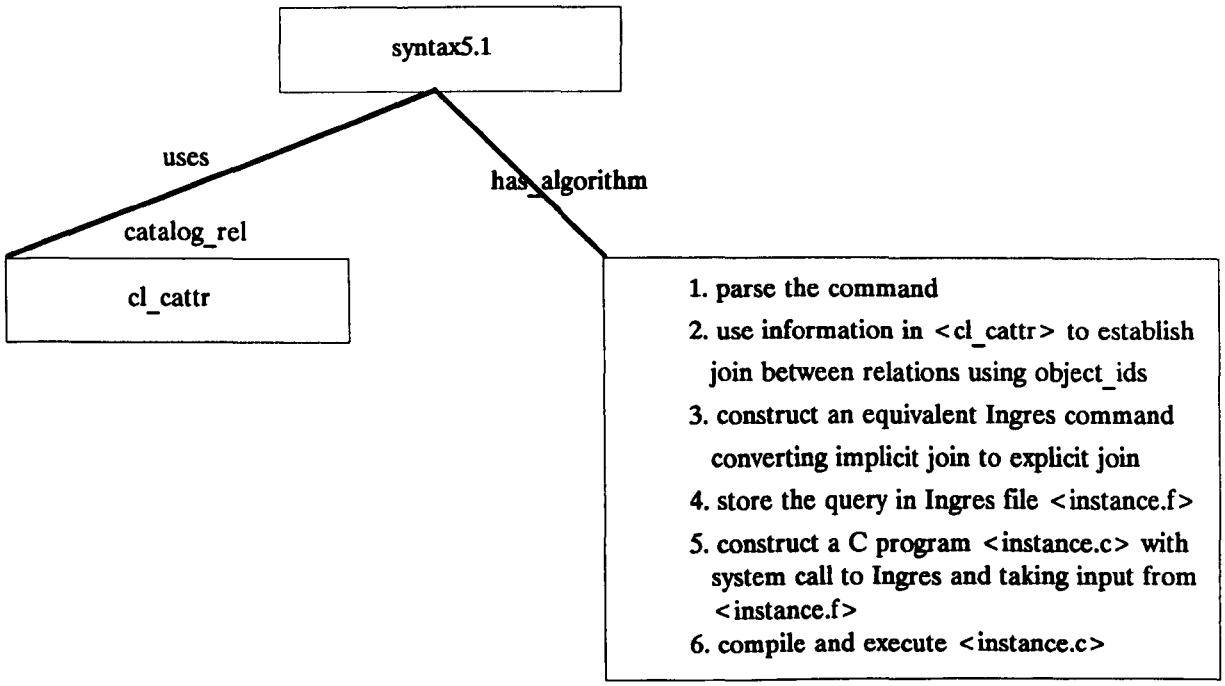


Figure 7A.21

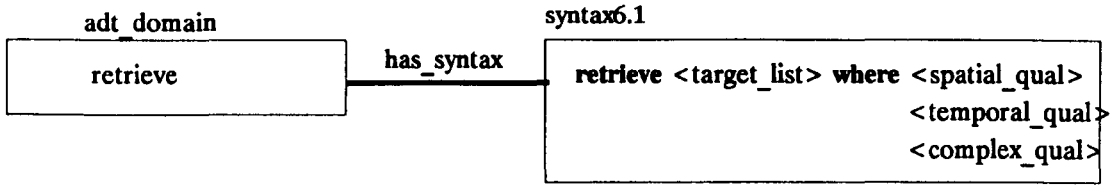


Figure 7A.22

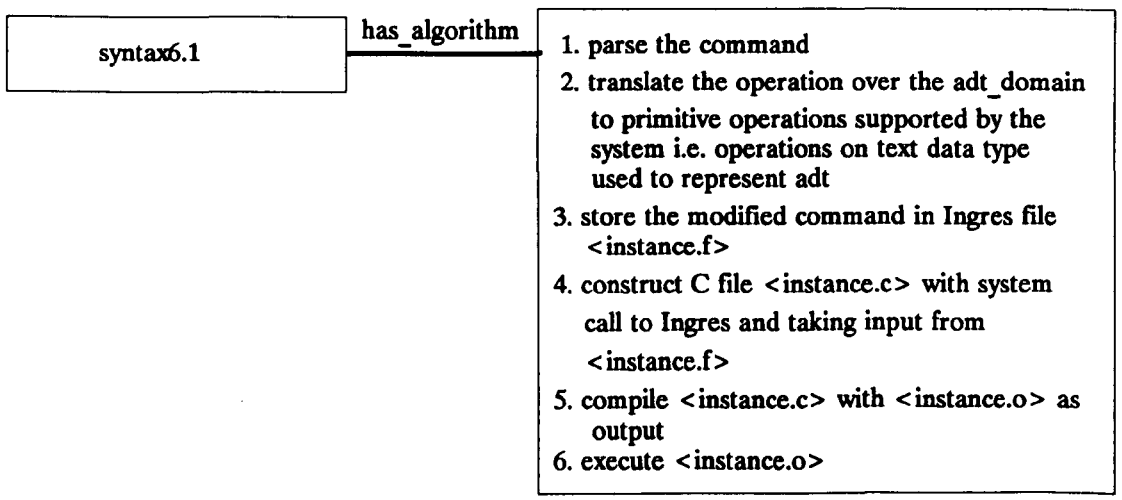


Figure 7A.23

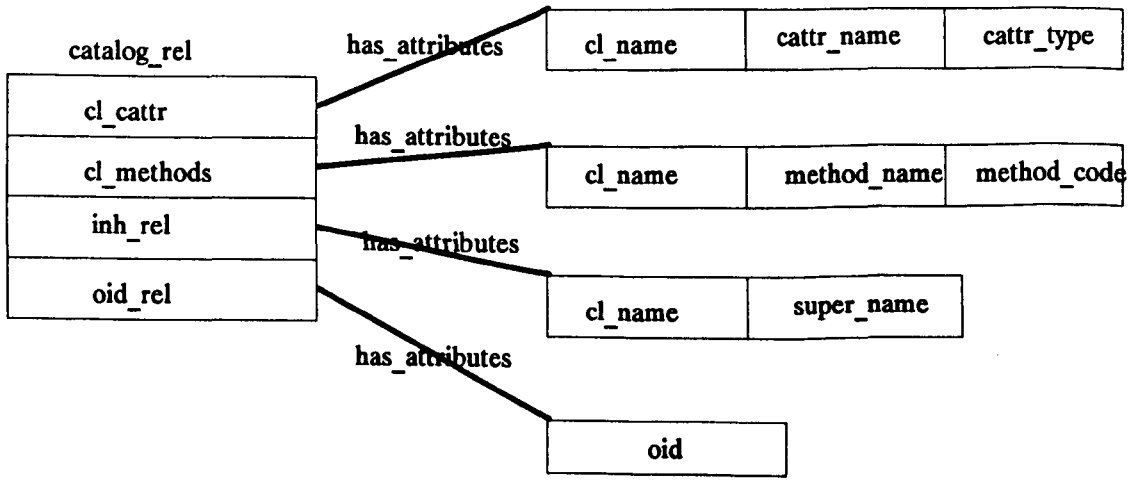


Figure 7A.24

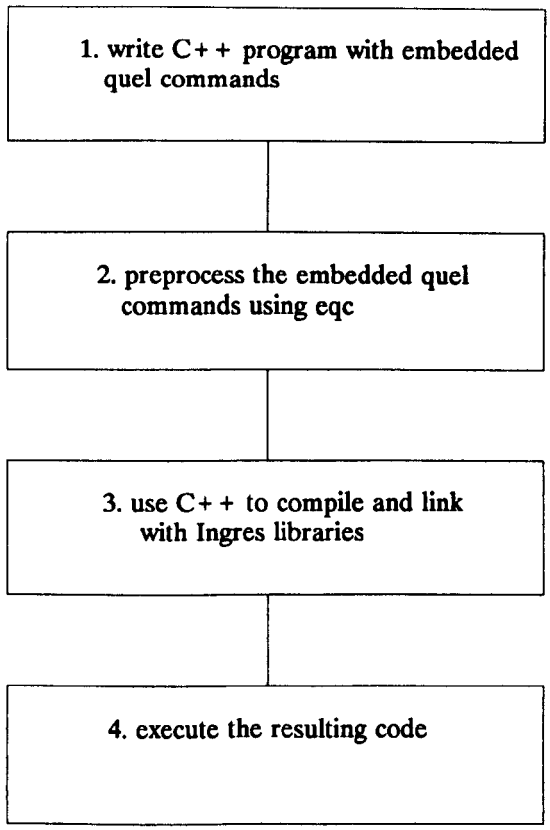

1. write C++ program with embedded quel commands
2. preprocess the embedded quel commands using eqc
3. use C++ to compile and link with Ingres libraries
4. execute the resulting code

Figure 7A.25

Appendix 7B

OQUEL Listings

```
/* oquel.c */
#include<stdio.h>
#include<ctype.h>

/*-----------------------------------------
|                definitions                !
 -----------------------------------------*/
#define BOOL    int
#define EQUAL    0
#define FALSE    0
#define TRUE     1
#define NULL     0
#define EOS     '\0'
/*-----------------------------------------
|                geol_time definitions          |
 -----------------------------------------*/
#define CAMBRIAN      0
#define ORDIVICIAN    1
#define CARBONIFEROUS 2
#define CRETACEOUS    3
#define TERTIARY      4
/*-----------------------------------------
|                macro definitions              |
 -----------------------------------------*/

#define iswhite(c) ((c)==' ' || (c)=='\t')
#define islparen(c) ((c)=='(')
#define isrparen(c) ((c)==')')
#define iscomma(c) ((c)==',')
#define isequal(c) ((c)=='=')
#define islcurlb(c) ((c)=='}')
#define isrcurlb(c) ((c)=='}')
#define isquote(c)  ((c)=='"')
/*-----------------------------------------
|                program limits                 |
 -----------------------------------------*/

#define NAMELEN  21
#define BUFMAX   800
```

```
/*-------------------------------------------------
|                  token definitions                  |
 -------------------------------------------------*/
#define ADD          -1
#define ADT          -2
#define CREATE       -3
#define DEFINE       -4
#define DELIMITER    -5
#define DISPLAY      -6
#define END          -7
#define EXECUTE      -8
#define FINISHED     -9
#define FROM        -10
#define INHERIT     -11
#define INSTANCE_OF-12
#define METHOD      -13
#define MODIFY      -14
#define NUMBER      -15
#define REF         -16
#define REMOVE      -17
#define RESERVED    -18
#define SELECT      -19
#define SOME        -20
#define STRING      -21
#define SUBTYPE_OF -22
#define TYPE        -23
#define WHERE       -24
#define VARIABLE    -25
/*-----------------------------------------------
|             error code definitions              |
 -----------------------------------------------*/

#define SYNTAX       1
#define WROPTION     2


char token[NAMELEN];      /* holds string rep. of token */
char tok;                 /* holds internal rep. of token */
char token_type;          /* contains type of token */
char obj_typ;
char word[NAMELEN];
```

```
char loperand[NAMELEN];
char roperand[NAMELEN];
char operand2[NAMELEN];
char operator[NAMELEN];
char logicalopr[NAMELEN];
static char *cmd_lptr;
static char *method_ptr;
static char mcode[8001];
static char methodpath[30];
char *txt;
char *get_word();
char *str_replace();
char type_name[NAMELEN];
char supclass_name[NAMELEN];
char method_name[NAMELEN];
char method_code[BUFMAX];
char attr_val_list[BUFMAX];
char where_clause[BUFMAX];
char rest_of_cmd[BUFMAX];
char inherit_var1[NAMELEN];
char inherit_var2[NAMELEN];
static struct commands {
       char reserved[20];
       char tok;
} table[] = {
       "add",          ADD,
       "adt",          ADT,
       "create",       CREATE,
       "define",       DEFINE,
       "delimiter",    DELIMITER,
       "display",      DISPLAY,
       "execute",      EXECUTE,
       "finished",     FINISHED,
       "from",         FROM ,
       "inherit",      INHERIT,
       "instance_of",INSTANCE_OF,
       "method",       METHOD,
       "modify",       MODIFY,
       "number",       NUMBER,
       "ref",          REF,
       "remove",       REMOVE,
```

```
        "reserved",   RESERVED,
        "select",     SELECT,
        "some",       SOME,
        "string",     STRING,
        "subtype_of", SUBTYPE_OF,
        "type",       TYPE,
        "variable",   VARIABLE,
        "where",      WHERE
};
static struct dict {
        char attr_name[20];
        char attr_type[20];
} dc [20];
static struct dict dc1[20];
static struct geo_table {
        char geo_time_name[20];
        char geo_time_code[3];
} geo_time[] = {
         "holocene",     "16",
         "pleistocene",  "15",
         "pliocene",     "14",
         "miocene",      "13",
         "oligocene",    "12",
         "eocene",       "11",
         "palaeocene",   "10",
         "cretaceous",   "09",
         "jurassic",     "08",
         "triassic",     "07",
         "permian",      "06",
         "carboniferous","05",
         "devonian",     "04",
         "silurian",     "03",
        "ordivician",   "02",
        "cambrian"  ,   "01"
         "precambrian",  "00"
};
char ln[120];
char str[BUFMAX+1];
struct inh_dict{
        char super_name[20];
} inh_dc[3];
```

```
int lnctr=0;
int attr_ctr=0;
int k=0;
int inh_ctr=0;
int err_flag;
int ch;
char *s;
char dbname[NAMELEN];
char o1[21];
char o2[21];
void get_line() ,skip_lparen(),skip_rparen(),skip_comma(),skip_equal(),skip_white(),
clearscreen(), skip_lcurlb(),skip_rcurlb(),get_buff();
main()
{
        FILE *fp1,*fopen();
        int ch,option;
        clearscreen();

        printf("*******************************\n");
        printf("*                             *\n");
        printf("* Enter database name ,please *\n");
        printf("*                             *\n");
        printf("*******************************\n");


        scanf("%s",dbname);
        fp1=fopen("out.c","w");
        printf(fp1,"%s",dbname);
        clearscreen();
        do {

                printf("************************************************\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("* please enter one of the following options    *\n");
                printf("* (1) class_attributes                         *\n");
                printf("* (2) class_instances                          *\n");
                printf("* (3) class_methods                            *\n");
```

```
            printf("* (4) class_heirarchy                          *\n");
            printf("* (5) adt_domain                               *\n");
            printf("* (6) complex_objects                          *\n");
            printf("* (7) quit                                     *\n");
            printf("*                                              *\n");
            printf("*                                              *\n");
            printf("*                                              *\n");
            printf("************************************************\n");
            scanf("%d",&option);
            switch(option)

            {
            case 1 :
                  class_attributes();
                  break;
            case 2 :
                  class_instances();
                  break;
            case 3 :
                  class_methods();
                  continue;
            case 4 :
                  class_heirarchy();
                  break;
            case 5 :
                  adt_domain();
                  break;
            case 6 :
                  complex_objects();
                  break;
            case 7 :
                  quit();
                  break;
            default :
                  err_flag=1;
                  break;
            };
      } while(option<=0);
      fclose(fp1);
}
```

```
/*-------------------class_attributes----------------*/


class_attributes()
{
        int option;
        clearscreen();

        do {
                printf("***********************************************\n");
                printf("*                                             *\n");
                printf("*                                             *\n");
                printf("*                                             *\n");
                printf("* Please Enter One Of The Following Options   *\n");
                printf("* (1) Create class attributes                 *\n");
                printf("* (2) Add class attributes                    *\n");
                printf("* (3) Delete class attributes                 *\n");
                printf("* (4) Quit                                    *\n");
                printf("*                                             *\n");
                printf("*                                             *\n");
                printf("***********************************************\n");
                scanf("%d",&option);
                switch(option)
                {
                case 1 :
                        create_class_attr();
                        break;
                case 2 :
                        add_class_attr();
                        break;
                case 3 :
                        delete_class_attr();
                        break;
                case 4 :
                        quit();
                        break;
                default:
                        err_flag=1;
                        break;
                };
        } while(option <=0);
```

```
}

/*-------------------create_class_attr---------------*/

/* define type type_name ( attri=domi, ...);          */
/* define type type_name ( attri=domi, ...) inherit (t1,..); */

create_class_attr()
{
        char *cp;
        int i;
        clearscreen();
        printf("*************************************************\n");
        printf("* define type type_name (attr1=val1,...) ;     *\n");
        printf("*                  or                           *\n");
        printf("* define type type_name (attr1=val1,..) inherit*\n");
        printf("* (type_name1,..)                               *\n");
        printf("*************************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();    /* define word */
        if (token_type != RESERVED)
        {
                error(11);
                return;
        }
        get_token();   /* type word */
        if (token_type != RESERVED)
        {
                error(12);
                return;
        }
        get_token();   /* class name */
        if (token_type != VARIABLE)
        {
                error(13);
                return;
        }
        strcpy(type_name,token);
        while (*cmd_lptr != ')') {
```

```
        skip_white(&cmd_lptr);
        skip_lparen(&cmd_lptr);
        get_token();
        strcpy(dc[attr_ctr].attr_name,token);
        skip_white(&cmd_lptr);
        get_token();
        skip_equal(&cmd_lptr);
        skip_white(&cmd_lptr);
        get_token();
        strcpy(dc[attr_ctr].attr_type,token);
        skip_white(&cmd_lptr);
        skip_comma(&cmd_lptr);
        attr_ctr++;
};
skip_rparen(&cmd_lptr);
skip_white(&cmd_lptr);
if (*cmd_lptr == ';')
{
        append_cl_cattr() ;
        exit (0);
}
else
{
        get_token();
        if (strncmp(token,"inherit",7)==0)
        {
                skip_white(&cmd_lptr);
                skip_lparen(&cmd_lptr);
                skip_white(&cmd_lptr);
        };
        while (*cmd_lptr != ')') {
                get_token();
                strcpy(inh_dc[inh_ctr].super_name,token);
                skip_white(&cmd_lptr);
                skip_comma(&cmd_lptr);
                skip_white(&cmd_lptr);
                inh_ctr++;
        };
        append_cl_cattr();
        append_inh_rel();
};
```

```
}

/* ------------------- append class common attributes ------*/
append_cl_cattr()
## {
int i;
## char cln[20],can[20],cat[20];
## ingres oodb
strcpy(cln,type_name);
for (i=0;i<attr_ctr;i++)
{
        strcpy(can,dc[i].attr_name);
        strcpy(cat,dc[i].attr_type);
## append cl_cattr(cl_name=cln,cattr_name=can,cattr_type=cat)
};
## exit
## }

/*-------------------append inheritance relation-----------*/
append_inh_rel()
## {
int i;
## char cln[20],super_n[20];
## ingres oodb
strcpy(cln,type_name);
for (i=0;i<inh_ctr;i++)
{
        strcpy(super_n,inh_dc[i].super_name);
## append inh_rel(class_name=cln,super_name=super_n)
};
## exit
## }

add_class_attr()
{
        clearscreen();
        printf("*****************************************************\n");
        printf("*                                                   *\n");
        printf("* add type <type_name> <attr_specs>                 *\n");
        printf("*                                                   *\n");
        printf("*                                                   *\n");
```

```
        printf("*************************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();     /* get add keyword */

        skip_white(&cmd_lptr);
        get_token();     /* get type keyword */
        skip_white(&cmd_lptr);
        get_token();     /* get type_name   */
        strcpy(type_name,token);
        while(*cmd_lptr != ')') {
               skip_white(&cmd_lptr);
               skip_lparen(&cmd_lptr);
               get_token();     /* attribute name  */
               strcpy(dc[attr_ctr].attr_name,token);
               skip_white(&cmd_lptr);
               get_token();
               skip_equal(&cmd_lptr);
               skip_white(&cmd_lptr);
               get_token();
               strcpy(dc[attr_ctr].attr_type,token);
               skip_white(&cmd_lptr);
               skip_comma(&cmd_lptr);
               attr_ctr++;
        };
        append_cl_cattr();
}

delete_class_attr()
{
        clearscreen();
        printf("*************************************************************\n");
        printf("*                                                           *\n");
        printf("* delete type <type_name> <attr_comma_list>                 *\n");
        printf("*                                                           *\n");
        printf("*                                                           *\n");
        printf("*************************************************************\n");

        get_line();
        cmd_lptr = ln;
```

```
        get_token();        /* get delete keyword */
        get_token();        /* get type keyword   */
        get_token();        /* get type name      */
        strcpy(type_name,token);
        while(*cmd_lptr != ')') {
                skip_white(&cmd_lptr);
                skip_lparen(&cmd_lptr);
                get_token();
                strcpy(dc[attr_ctr].attr_name,token);
                skip_white(&cmd_lptr);
                skip_comma(&cmd_lptr);
                attr_ctr++;
        };
        delete_cl_cattr();
}

delete_cl_cattr()
## {
int i;
## char cln[20],can[20];
## ingres oodb
strcpy(cln,type_name);
for(i=0;i<attr_ctr;i++)
{
        printf("%s",dc[i].attr_name);
        strcpy(can,dc[i].attr_name);
## range of r is cl_cattr
## delete r where r.cl_name = cln and  r.cattr_name=can
};
## exit
## }
modify_class_attr()
{
}


/*-------------------class_instances--------------------*/

class_instances()
{
        int option;
```

```
        clearscreen();
        do {
                printf("************************************************\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("* Please Enter One Of The Following Options    *\n");
                printf("* (1) Create class instances                   *\n");
                printf("* (2) Add class instances                      *\n");
                printf("* (3) Remove class instances                   *\n");
                printf("* (4) Modify class instances                   *\n");
                printf("* (5) Quit                                     *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("************************************************\n");
                scanf("%d",&option);
                switch(option)
                {
                case 1 :
                        create_class_instances();
                        break;
                case 2 :
                        add_class_instances();
                        break;
                case 3 :
                        remove_class_instances();
                        break;
                case 4 :
                        modify_class_instances();
                        break;
                case 5 :
                        quit();
                        break;
                default:
                        err_flag=1;
                        break;
                };
        } while(option <=0);
}
```

```
/*-------------------------create_class_instances----*/

/*  create relation_name { type_name }                */
create_class_instances()
## {
FILE *fp1,*fopen();
char *cp;
int flag;
## char ta_name[21],ta_type[21];
## char  a_name[21],a_type[21];
## char cla_name[21],typ_name[21];
## char *ptyp_name;
## int co;
*str='\0';
clearscreen();
printf("*********************************************\n");
printf("* create class_name { type_name }           *\n");
printf("*                                           *\n");
printf("*********************************************\n");
get_line();
cmd_lptr = ln;
get_token();        /* create word */
skip_white(&cmd_lptr);
get_token();       /* class name */
strcpy(cla_name,token);
printf("%s \n",cla_name);
skip_white(&cmd_lptr);
get_token();
skip_lcurlb(&cmd_lptr);
skip_white(&cmd_lptr);
get_token();    /* type name */
strcpy(typ_name,token);
ptyp_name=typ_name;
skip_white(&cmd_lptr);
skip_rcurlb(&cmd_lptr);
skip_white(&cmd_lptr);
strcpy(str,"create ");
strcat(str,cla_name );
strcat(str,"(");
flag=TRUE;
flag=!flag;
```

```
## ingres oodb
## range of r is cl_cattr
## retrieve (co=count(r.all))
## where r.cl_name = ptyp_name
## retrieve (a_name=r.cattr_name ,a_type=r.cattr_type)
##  where r.cl_name=  ptyp_name
## {
if (flag)

{
        strcat(str,",");
        strcpy(ta_name,a_name);
        strcpy(ta_type,a_type);
        strcat(str,ta_name);
        strcat(str," = ");
        strcat(str,ta_type);
} else
{
        strcpy(ta_name,a_name);
        strcpy(ta_type,a_type);
        strcat(str,ta_name);
        strcat(str," = ");
        strcat(str,ta_type);
        flag= !flag;
}

## }
strcat(str,")");
fp1=fopen("instance.f","w");
fprintf(fp1,"%s",str);
fprintf(fp1,"\n\\go\n");
fclose(fp1);
call_ingres();
## exit
## }
call_ingres()
{
        FILE *fp1,*fopen();
        fp1=fopen("instance.c","w");
        fprintf(fp1,"main()\n");
        fprintf(fp1,"{\n");
```

```
        fprintf(fp1,"system(\"/ingres/bin/ingres/ ");
        fprintf(fp1,"%s ",dbname);
        fprintf(fp1," -s < instance.f\");");
        fprintf(fp1,"}\n");
        fclose(fp1);
        system("cc -o instance.out instance.c ");
        system("instance.out");
};

call_cc()
{
int return_code;
char *oldname,*newname;
oldname=methodpath;
newname="ccin.q";
return_code=rename(oldname,newname);
         system("eqc ccin.q");
        system("CC -o ccin.o ccin.c /ingres/lib/libqlib /ingres/lib/compatlib -lm -lC -lc")
        system("ccin.o");
oldname="ccin.q";
newname=methodpath;
return_code=rename(oldname,newname);
};

/*------------------add_class_instances---------*/

/* add instance_of class_name  (attr1=val1, ..) */
add_class_instances()
{
        char *cp;
        int i=0;
        FILE *fp, *fopen();
        *str='\0';
        clearscreen();
        printf("*************************************************\n");
        printf("*  add instance_of class_name (attr1=val1,..) *\n");
        printf("*                                                *\n");
        printf("*************************************************\n");

        get_line();
        cmd_lptr = ln;
```

```
        get_token();      /* get add  keyword  */
        skip_white(&cmd_lptr);
        get_token();     /* get instance_of keyword */
        skip_white(&cmd_lptr);
        get_token();      /* get type name */
        strcpy(type_name,token);
        while(*cmd_lptr != ')' ){
              attr_val_list[i++] =*cmd_lptr++;
        };
        attr_val_list[i++]=')';
        strcat(str," append  ");
        strcat(str,type_name );
        strcat(str, attr_val_list);
        fp=fopen("instance.f","w");
        fprintf(fp,"%s",str);
        fprintf(fp,"\n\\go\n");
        fclose(fp);
        call_ingres();
}

/*---------------remove_class_instances--------*/

/* remove instances_of class_name where qual ;       */

remove_class_instances()
{
        char *cp;
        int i=0;
        FILE *fp, *fopen();
        *str='\0';
        clearscreen();
        printf("*****************************************\n");
        printf("* remove instance_of class where qual ;    *\n");
        printf("*                                          *\n");
        printf("*****************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();      /* get remove keyword */
        skip_white(&cmd_lptr);
        get_token();      /* get instance_of keyword */
```

```
        skip_white(&cmd_lptr);
        get_token();      /* get type_name */
        strcpy(type_name,token);
        while(*cmd_lptr != ';') {
              where_clause[i++] =*cmd_lptr++;
        };
        strcat(str,"range of ");
        strcat(str,type_name);
        strcat(str, " is ");
        strcat(str,type_name);
        strcat(str," ");
        strcat(str, " delete ");
        strcat(str, type_name);
        strcat(str,where_clause);
        fp=fopen("instance.f","w");
        fprintf(fp,"%s",str);
        fprintf(fp,"\n\\go\n");
        fclose(fp);
        call_ingres();
}

modify_class_instances()
{
        int i=0;
        FILE *fp, *fopen();
        *str = '\0';
        clearscreen();
        printf("*****************************************************\n");
        printf("*                                                   *\n");
        printf("* modify inst_of <class_name> <target_list>         *\n");
        printf("* where <qual>;                                     *\n");
        printf("*                                                   *\n");
        printf("*****************************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();      /* get modify keyword */
        skip_white(&cmd_lptr);
        get_token();      /* get inst_of keyword */
        skip_white(&cmd_lptr);
        get_token();      /* get class_name      */
```

```
        strcpy(type_name,token);
        while(*cmd_lptr != ';') {
                rest_of_cmd[i++] = *cmd_lptr++;
        };
        strcat(str," range of ");
        strcat(str,type_name);
        strcat(str," is ");
        strcat(str,type_name);
        strcat(str," replace ");
        strcat(str,type_name);
        strcat(str,rest_of_cmd);
        fp=fopen("instance.f","w");
        fprintf(fp,"%s",str);
        fprintf(fp,"\n\\go\n");
        fclose(fp),
            call_ingres();
}

/*--------------------class_methods---------------------*/


class_methods()
{
        int option;
        clearscreen();
        do {
                printf("************************************************\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("*                                              *\n");
                printf("* Please Enter One Of The Following Options     *\n");
                printf("* (1) Create class methods                     *\n");
                printf("* (2) Modify class methods                     *\n");
                printf("* (3) Remove class methods                     *\n");
                printf("* (4) Display methods                          *\n");
                printf("* (5) Execute methods                          *\n");
                printf("* (6) Quit                                     *\n");
                printf("*                                              *\n");
                printf("************************************************\n");
                scanf("%d",&option);
                switch(option)
```

```
                {
                case 1 :
                        create_class_methods();
                        break;
                case 2 :
                        modify_class_methods();
                        break;
                case 3 :
                        remove_class_methods();
                        break;
                case 4 :
                        display_class_methods();
                        break;
                case 5 :
                        execute_class_methods();
                        continue;
                        /* break; */
                case 6 :
                        quit();
                        break;
                default:
                        err_flag=1;
                        break;
                };
        } while(option <=0);
}
/*------------------create_class_methods------*/

/*  create class_name method method_name  (     ) */

create_class_methods()
{
        char *cp;
        int i=0;
        clearscreen();
        printf("*************************************************\n");
        printf("*                                               *\n");
        printf("* create class_name method method_name (    )   *\n");
        printf("*                                               *\n");
        printf("*************************************************\n");
```

```
        get_line();
        cmd_lptr = ln;
        get_token();        /* get create keyword */
        get_token();        /* get class name */
        strcpy(type_name,token);
        get_token();        /* get method keyword */
        get_token();        /* get_method_name  */
        strcpy(method_name,token);
        skip_white(&cmd_lptr);
        skip_lparen(&cmd_lptr);
        while (*cmd_lptr != ')') {
               method_code[i++] =*cmd_lptr++;
        };
        method_code[i]='\0';
        append_cl_method();
}

/*-----------------append_cl_method--------------*/

append_cl_method()
##{
## char cln[20],cmn[20],cmc[BUFMAX];
## ingres oodb
strcpy (cln,type_name);
strcpy (cmn,method_name);
strcpy (cmc,method_code);
## append cl_methods(class_name=cln,method_name=cmn,method_code=cmc)
## exit
##}



/*-----------------modify_class_methods-------*/
modify_class_methods()
{
        char cl_n[20],cmn[20];
        int i=0;
        char str2[900];
        FILE *fp, *fopen();
        *str2 = '\0';
        clearscreen();
```

```
        printf("*****************************************************\n");
        printf("*                                                   *\n");
        printf("* modify cl_name method ( method_name = $1,         *\n");
        printf("* method_code = $2 ) where method_name = $3         *\n");
        printf("*                                                   *\n");
        printf("*****************************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();    /* get modify keyword  */
        get_token();    /* get class name      */
        strcpy(cl_n,token);
        get_token();    /* get method keyword  */
        skip_white(&cmd_lptr);
        while (*cmd_lptr != ')') {
              attr_val_list[i++] = *cmd_lptr++;
        };
        attr_val_list[i]=')';
        skip_rparen(&cmd_lptr);
        get_token();    /* get where keyword */
        get_token();    /* get method_name keyword */
        get_token();    /* get equal sign */
        get_token();
        strcpy(cmn,token);
        strcat(str2, "range of r is cl_methods ");
        strcat(str2, " replace r ");
        strcat(str2, attr_val_list);
        strcat(str2, " where r.method_name = ");
        strcat(str2, "\"");
        strcat(str2, cmn);
        strcat(str2,"\"");
        strcat(str2, " and  r.class_name = ");
        strcat(str2,"\"");
        strcat(str2, cl_n);
        strcat(str2,"\"");
        fp=fopen("instance.f","a");
        fprintf(fp,"%s",str2);
        fprintf(fp,"\n\\go\n");
        fclose(fp);
        call_ingres();
}
```

```
/*------------------remove_class_methods--------*/

/* remove class_name method where method_name = $1 */

remove_class_methods()
{
## char cl_n[20],cmn[20];
        char *cp;
        clearscreen();
        printf("*****************************************************\n");
        printf("* remove class_name method where method_name = $1*\n");
        printf("*                                                   *\n");
        printf("*****************************************************\n");

        get_line();
        cmd_lptr = ln;
        get_token();        /* get remove keyword */
        get_token();        /* get type name */
        strcpy(cl_n,token);
        get_token();        /* get method keyword */
        get_token();        /* get where keyword  */
        get_token();        /* get method_name    */
        skip_white(&cmd_lptr);
        skip_equal(&cmd_lptr);
        skip_white(&cmd_lptr);
        get_token();
        strcpy(cmn,token);
## ingres oodb
## range of r is cl_methods
## delete r where r.class_name = cl_n and r.method_name = cmn
## exit
}

/*------------------display_class_methods------*/

/* display class_name methods                  */

display_class_methods()
{
        char str2[900];
        FILE *fp, *fopen();
```

```
        clearscreen();
        printf("*********************************************\n");
        printf("* display class_name methods                *\n");
        printf("*                                           *\n");
        printf("*********************************************\n");
        *str2='\0';
        get_line();
        cmd_lptr = ln;
        get_token();      /* get disply keyword */
        get_token();      /* get class_name     */
        strcpy(type_name,token);
        get_token();      /* methods      */
        strcat(str2, " range of r is cl_methods ");
        strcat(str2, " retrieve  (methodname = ");
        strcat(str2, " r.method_name ) ");
        strcat(str2, " where r.class_name = ");
        strcat(str2, "\"");
        strcat(str2, type_name);
        strcat(str2, "\"");
        fp=fopen("instance.f","w");
        fprintf(fp,"%s",str2);
        fprintf(fp, "\n\\go\n");
        fclose(fp);
        call_ingres();
}

/*-----------------execute_class_methods------*/
/* execute (type,method_name,object);          */


execute_class_methods()
{
        char str0[8001];
        char str2[8001];
        char *cp;
        char *cp1;
        char obj_typ1;
        char obj_typ2;
## char str1[8001];
## char clname[21];
## char methodn[21];
```

```
        FILE *fp,fp1,*fopen();
        *str2='\0';
        clearscreen();
        printf("*****************************************\n");
        printf("* execute(type,method_name,obj);      *\n");
        printf("*                                       *\n");
        printf("*****************************************\n");
        fp=fopen("instance.f","w");
        get_line();
        cmd_lptr = ln;
        get_token();      /* get execute keyword */
        skip_white(&cmd_lptr);
        skip_lparen(&cmd_lptr);
        get_token();      /* get class */
        strcpy(type_name,token);
        strcpy(clname,type_name);
        skip_comma(&cmd_lptr);
        get_token();      /* get method_name */
        strcpy(method_name,token);
        strcpy(methodn,method_name);
        skip_comma(&cmd_lptr);
        get_token();      /* get object   */
        strcpy(o1,token);
        obj_typ1= token_type;
## ingres oodb
## range of r is cl_methods
## retrieve (str1= r.method_code) where r.class_name = clname
## and r.method_name=methodn
## exit
        strcpy(str2,str1);
if (strncmp(str2,"@",1)==0)
{
strncpy(methodpath,str2,30);
call_cc();
} else {
        /* cp= str_replace(str2,"o1",o1);*/
        strcpy(str0,cp);
        strcat(str0,";");
        method_ptr=cp;
        skip_comma(&cmd_lptr);
        get_token();
```

```
        obj_typ2= token_type;
        obj_typ=obj_typ2;
        strcpy(o2,token);
        cp1=str_replace(str2,"o2",o2);
        obj_typ=obj_typ1;
        strcpy(str0,cp1);
        strcat(str0,";");
        cp = str_replace(str0,"o1",o1);
        fp=fopen("instance.f","a");
        fprintf(fp,"%s",cp);
        fprintf(fp,"\n\\go\n");
        fclose(fp);
        call_ingres();
}
}


/*--------------------class_heirarchy--------------------*/

class_heirarchy()
{
        int option;
        clearscreen();
        do {
                printf("*************************************************\n");
                printf("*                                               *\n");
                printf("*                                               *\n");
                printf("*                                               *\n");
                printf("* Please Enter One Of The Following Options      *\n");
                printf("* (1) Create class heirarchy                    *\n");
                printf("* (2) Remove class heirarchy                    *\n");
                printf("* (3) Quit                                      *\n");
                printf("*                                               *\n");
                printf("*                                               *\n");
                printf("*                                               *\n");
                printf("*************************************************\n");
                scanf("%d",&option);
                switch(option)
                {
                case 1 :
```

```
                create_class_heirarchy();
                break;
            case 2 :
                remove_class_heirarchy();
                break;
            case 3 :
                quit();
                break;
            default:
                err_flag=1;
                break;
            };
        } while(option <=0);
}


/*---------------create_class_heirarchy---------*/

/*   create class_name subtype_of class_name     */

create_class_heirarchy()
{
## char cln[20],supcn[20];
        char *cp;
        clearscreen();
        printf("************************************************\n");
        printf("* create class_name subtype_of class_name      *\n");
        printf("*                                              *\n");
        printf("************************************************\n");
        get_line();
        cmd_lptr = ln;
        get_token();      /* get create keyword */
        get_token();      /* get class name     */
        strcpy(cln,token);
        get_token();      /* get subtype_of keyword */
        get_token();      /* get superclass name    */
        strcpy(supcn,token);

## ingres oodb
## range of r is inh_rel
## append inh_rel(class_name=cln,super_name=supcn)
```

```
## exit
}


/*--------------remove_class_heirarchy----------*/

/* remove class_name from supclass_name          */

remove_class_heirarchy()
{
## char type_name[20],supclass_name[20];
       char *cp;
       clearscreen();
       printf("*************************************************\n");
       printf("* remove class_name from supclass_name          *\n");
       printf("*                                               *\n");
       printf("*************************************************\n");

       get_line();
       cmd_lptr = ln;
       get_token();       /* get remove keyword */
       get_token();       /* get class name     */
       strcpy(type_name,token);
       get_token();       /* get from keyword    */
       get_token();       /* get superclass name */
       strcpy(supclass_name,token);

## ingres oodb
## range of r is inh_rel
## delete r where r.class_name = type_name and r.super_name=supclass_name
## exit
}


adt_domain()
{
       int *loc;
       char xb[21], yb[21];
       char xt[21], yt[21];
       char axis[21];
       int i;
```

```
char *bufptr;
FILE *fp1,*fopen();
*str='\0';
clearscreen();
printf("*****************************************\n");
printf("*  range of r is <class_list>           *\n");
printf("*  retrieve <target_list>               *\n");
printf("*  where <spatial_qual>                 *\n");
printf("*        <geo_time_qual>                *\n");
printf("*  east,west,north,south                *\n");
printf("*  before, after, between               *\n");
printf("*****************************************\n");

get_line();
cmd_lptr = ln;
/* *loc = right_index(cmd_lptr,"where");*/
get_token();
strcat(str,token);
while(strncmp(token,"where",5)!=0){
       get_token();
       strcat(str," ");
       strcat(str,token);
};
while (*cmd_lptr != ";") {       /* 13 dec */

       if (tok == FINISHED) {
              get_line();
              cmd_lptr = ln;
       };
       get_token();
       strcpy(loperand,token);
       get_token();
       strcpy(operator,token);
       get_token();
       if (strncmp(operator,"overlap",7)==0 ||
             strncmp(operator,"contained_in",12)==0 ||
           strncmp(operator,"abuts_top",9)==0 ||
             strncmp(operator,"abuts_bottom",12)==0 ||
           strncmp(operator,"abuts_left",10)==0 ||
             strncmp(operator,"abuts_right",11)==0)
       {
```

```
        strcpy(xb,token);
        skip_comma(&cmd_lptr);
        get_token();
        strcpy(yb,token);
        skip_comma(&cmd_lptr);
        get_token();
        strcpy(xt,token);
        skip_comma(&cmd_lptr);
        get_token();
        strcpy(yt,token);
}
strcpy(roperand,token);
if (strncmp(operator,"east",4)==0 || strncmp(operator,"west",4)==0 ) {
        strcat(str," ");
        strcat(str," right(");
        strcat(str,loperand);
        if (strncmp(operator,"east",4)==0)
                strcat(str,",2) > right (");
        else
                strcat(str,",2) < right (");
        strcat(str,"\n");
        strcat(str,roperand);
        strcat(str,",2) and left (");
        strcat(str,loperand);
        strcat(str, ",2) = left (");
        strcat(str, roperand);
        strcat(str,",2)");

}
if (strncmp(operator,"north",5)==0 || strncmp(operator,"south",5)==0) {
        strcat(str," ");
        strcat(str,"left(");
        strcat(str,loperand);
        if (strncmp(operator,"north",5)==0)
                strcat(str,",2) > left(");
        else
                strcat(str,",2) < left(");
        strcat(str,"\n");
        strcat(str,roperand);
        strcat(str,",2) and right (");
        strcat(str,loperand);
```

```
        strcat(str,",2) = right(");
        strcat(str,roperand);
        strcat(str,",2)");
}
if (strncmp(operator,"overlap",7)==0 )
{
        strcat(str," left(");
        strcat(str,loperand);
        strcat(str,",2 ) <");
        strcat(str,xb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-3)");
        strcat(str,",2 ) <");
        strcat(str,yb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-6),2 ) >");
        strcat(str,xt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-9),2)>");
        strcat(str,yt);
}
if (strncmp(operator,"contained_in",12)==0)
{
        strcat(str," left(");
        strcat(str,loperand);
        strcat(str,",2 ) >=");
        strcat(str,xb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-3)");
```

```
        strcat(str,",2 ) >=");
        strcat(str,yb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-6),2 ) <=");
        strcat(str,xt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-9),2)<=");
        strcat(str,yt);
}
if (strncmp(operator,"abuts_left",10)==0)
{
        strcat(str," left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-3)");
        strcat(str,",2 ) <");
        strcat(str,yt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-6),2 ) =");
        strcat(str,xb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-9),2)>");
        strcat(str,yb);
}

if (strncmp(operator,"abuts_right",11)==0)
{
        strcat(str,"left(");
```

```
        strcat(str,loperand);
        strcat(str,",2) =");
        strcat(str,xt);
        strcat(str," left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-3)");
        strcat(str,",2 ) <");
        strcat(str,yt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-9),2)>");
        strcat(str,yb);
}
if (strncmp(operator,"abuts_top",9)==0)
{
        strcat(str," left(");
        strcat(str,loperand);
        strcat(str,",2 ) <");
        strcat(str,xt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-3)");
        strcat(str,",2 ) <");
        strcat(str,yt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-6),2 ) >");
        strcat(str,xb);
}

if (strncmp(operator,"abuts_bottom",12)==0)
{
        strcat(str," left(");
```

```
        strcat(str,loperand);
        strcat(str,",2 ) <");
        strcat(str,xt);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-6)");
        strcat(str,",2 ) >");
        strcat(str,xb);
        strcat(str," and left(right(");
        strcat(str,loperand);
        strcat(str,",size(");
        strcat(str,loperand);
        strcat(str,")-9),2 ) =");
        strcat(str,yb);
}
if (strncmp(operator,"above",5)==0 ||
      strncmp(operator,"below",5)==0 ||
    strncmp(operator,"right_of",8)==0 ||
      strncmp(operator,"left_of",7)==0)
{
        strcpy(axis,token);
        if (strncmp(operator,"above",5)==0)
        {
                strcat(str," left(right(");
                strcat(str,loperand);
                strcat(str,",size(");
                strcat(str,loperand);
                strcat(str,")-3),2)");
                strcat(str,">=");
                strcat(str,axis);
        }

        if (strncmp(operator,"below",5)==0)
        {
                strcat(str," left(right(");
                strcat(str,loperand);
                strcat(str,",size(");
                strcat(str,loperand);
                strcat(str,")-9),2)");
```

```
                strcat(str,"<=");
                strcat(str,axis);
        }
        if (strncmp(operator,"left_of",7)==0)
        {
                strcat(str," left(right(");
                strcat(str,loperand);
                strcat(str,",size(");
                strcat(str,loperand);
                strcat(str,")-6),2)");
                strcat(str,"<=");
                strcat(str,axis);
        }
        if (strncmp(operator,"right_of",8)==0)
        {
                strcat(str," left(");
                strcat(str,loperand);
                strcat(str,",2)");
                strcat(str,">=");
                strcat(str,axis);
        }
}

if (strncmp(operator,"before",6)==0 || strncmp(operator,"after",5)==0)
{
        strcat(str," ");
        strcat(str,"right(");
        strcat(str,loperand);
        strcat(str,",1)");
        if (strncmp(operator,"before",6)==0)
                strcat(str," < ");
        else
                strcat(str," > ");
        for(i=0;i<=4;i++)
                if (strncmp(roperand,geo_time[i].geo_time_name,8)==0)
                        strcat(str,geo_time[i].geo_time_code);
}
if (strncmp(operator,"between",7)==0) {
        strcat(str," right(");
        strcat(str,loperand);
        strcat(str,",1)");
```

```
                    strcat(str,">= ");
                    for(i=0;i<=4;i++)
                        if (strncmp(roperand,geo_time[i].geo_time_name,8)==0)
                            strcat(str,geo_time[i].geo_time_code);
                    get_token();
                    strcpy(logicalopr,token);
                    strcat(str,logicalopr);
                    strcat(str," right(");
                    strcat(str,loperand);
                    strcat(str,",1)");
                    strcat(str," <= ");
                    get_token();
                    strcpy(operand2,token);
                    for(i=0;i<=4;i++)
                        if (strncmp(operand2,geo_time[i].geo_time_name,8)==0)
                            strcat(str,geo_time[i].geo_time_code);
                }
                get_token();
                skip_white(&cmd_lptr);
                strcpy(logicalopr,token);
                if (strncmp(logicalopr,";",1)==0) break;
                else strcat(str,token);
        }; /* 13 dec */

        fp1= fopen("instance.f","w");
        fprintf(fp1,"%s",str);
        fprintf(fp1,"\n\\go\n");
        fclose(fp1);
        call_ingres();
}

/*---------------------------complex_objects-----------------*/

complex_objects()
{
        FILE *fp,*fopen();
        char constant[20],comparator[20],range_var[20],cl_name[20];
        int i=0;
## static int i1,j;
## char dc2[10][21];
## char att_name[21];
```

```
## int n, finished;
## char *attrptr;
## char a_name[21],a_type[21];
        *str='\0';
        attr_ctr=0;
        clearscreen();
        printf(" ************************************************************ \n");
        printf(" *                                                          * \n");
        printf(" * object get range_var qual;                               * \n");
        printf(" *                                                          * \n");
        printf(" *                                                          * \n");
        printf(" *                                                          * \n");
        printf(" *                                                          * \n");
        printf(" ************************************************************ \n");

        get_line();
        cmd_lptr = ln;
        get_token();  /* get object */
        strcpy(cl_name,token);
        get_token();  /* get get    */
        get_token();  /* range_var  */
        strcpy(range_var,token);
        get_token();  /* lparen     */
        get_token();  /* range_var  */
        while (token_type != RESERVED  ) {
                get_token();
                if (token_type != RESERVED ) {
                        strcpy(dc1[attr_ctr++].attr_name,token);
                        /* attr_ctr++;*/
                }
        }
        printf("%d \n",attr_ctr);
        printf("%d \n",attr_ctr);
        for (i=0;i<=attr_ctr;i++)
        {
                printf("%s\n",dc1[i].attr_name);
        }
        strcpy(comparator,token);
        get_token();
        strcpy(constant,token);
## ingres oodb
```

```
## range of r is cl_cattr
        attr_ctr--;
        for (k=0;k<attr_ctr;k++)
        {
                j=k;
                strcpy(att_name,dc1[k].attr_name);
                attrptr=att_name;

## retrieve (a_name=r.cattr_name,dc2[j]=r.cattr_type )
##  where r.cattr_name = att_name
        }
        strcat(str,"range of r is ");
        strcat(str,cl_name);
        strcat(str," retrieve (r.oid) where ");
        strcat(str,range_var);
        strcat(str,".");
        strcat(str,dc1[0].attr_name);
        strcat(str," = ");
        strcat(str,dc2[0]);
        strcat(str,".oid");
        for (k=1;k<attr_ctr;k++) {
                strcat(str," ");
                strcat(str," and ");
                strcat(str,dc2[k-1]);
                strcat(str,".");
                strcat(str,dc1[k].attr_name);
                strcat(str,"=");
                strncat(str,dc2[k],strlen(dc2[k]));
                strcat(str,".oid");
        }
        k=attr_ctr;
        k--;
        strcat(str," and ");
        strcat(str,dc2[k]);
        strcat(str,".");
        strcat(str," ");
        strcat(str,dc1[attr_ctr].attr_name); /* attr_ctr */
        strcat(str,"=");
        strncat(str,dc2[k],strlen(dc2[k]));
        strcat(str,"_");
        strcat(str,dc1[attr_ctr].attr_name);
```

```
        strcat(str,".");
        strncat(str,dc2[k],strlen(dc2[k]));
        strcat(str,"_");
        strcat(str,"oid");
        strcat(str," and ");
        strncat(str,dc2[k],strlen(dc2[k]));
        strcat(str,"_");
        strcat(str,dc1[attr_ctr].attr_name);
        strcat(str,".");
        strcat(str,constant);
        strcat(str,"_");
        strcat(str,"oid");
        strcat(str,"=");
        strcat(str,constant);
        strcat(str,".");
        strcat(str,"oid");

        for (i=0;i<=attr_ctr;i++){
        }
        /* strcat(str," ");
strcat(str,comparator);
strcat(str,constant); */
        printf("%s ",str);

        fp=fopen("instance.f","w");
        fprintf(fp,"%s",str);
        fprintf(fp,"\n\\go\n");
        fclose(fp);
        call_ingres();
}

right_index(string,substring)
char *string, *substring;
{
        int i,j,k;
        int loc = -1;
        for(i=0; *(string+i) != EOS; i++)
                for (j=i,k=0; (*(substring +k) != EOS) &&
                    ( *(substring +k) == *(string+j));k++,j++)
                        if ((*substring + k+1) == ' ' || *(substring +k+1) ==EOS) {
                                loc = i + strlen(substring);
```

```
                        break;
                }
        return (loc);
}


/*--------------------skip_white-----------------*/
static void skip_white(s)
char **s;
{
        while (iswhite(**s))
                (*s)++;

}
/*----------------------skip_lparen---------------*/
static void skip_lparen(s)
char **s;
{
        while (islparen(**s))
                (*s)++;
}

/*----------------------skip_rparen---------------*/
static void skip_rparen(s)
char **s;
{
        while (isrparen(**s))
                (*s)++;
}

/*----------------------skip_lcurlb----------------*/
static void skip_lcurlb(s)
char **s;
{
        while (islcurlb(**s))
                (*s)++;
}

/*--------------------skip_rcurlb-----------------*/
static void skip_rcurlb(s)
char **s;
```

```
{
        while (isrcurlb(**s))
                (*s)++;
}

/*----------------------skip_comma--------------------*/
static void skip_comma(s)
char **s;
{
        while (iscomma(**s))
                (*s)++;
}

/*-----------------------skip_equal-------------------*/
static void skip_equal(s)
char **s;
{
        while (isequal(**s))
                (*s)++;
}

/*-------------------get_token-------------------------*/
get_token()
{
        register char *temp;
        token_type = 0;
        tok = 0;
        temp = token;
        if (*cmd_lptr == '\0' ) { /*  end of file */
                *token =0;
                tok = FINISHED;
                return (token_type = DELIMITER);
        }
        while (iswhite(*cmd_lptr)) ++cmd_lptr;  /* skip over white spaces */
        if (strchr( "+-/%*^=;(),{}<>",*cmd_lptr)) { /* delimiter */
                *temp = *cmd_lptr;
                cmd_lptr++; /* advance to next position */
                temp++;
                *temp = 0;
                /* token_type = DELIMITER; */
                return (token_type = DELIMITER);
```

```
        }
        if (isdigit(*cmd_lptr)) { /* number */
                while (!isdelimit(*cmd_lptr)) *temp++ = *cmd_lptr++;
                *temp = '\0';
                /* token_type = NUMBER;*/
                return (token_type = NUMBER);
        };
        if (isalpha(*cmd_lptr)) {
                while (!isdelimit(*cmd_lptr)) *temp++ = *cmd_lptr++;
                token_type = STRING;
        }
        *temp = '\0';
        if (token_type == STRING) {
                tok = look_up(token);
                if (!tok) token_type = VARIABLE;
                else token_type = RESERVED;
        }
        if (isquote(*cmd_lptr)){
                *temp++ = *cmd_lptr++;
                while(!isquote(*cmd_lptr)) *temp++ = *cmd_lptr++;
                *temp++ = *cmd_lptr++;
                *temp = '\0';
                token_type= STRING;
        }
        return token_type;
}


/*-----------------------putback()------------*/

/* return a token to input stream */
void putback()
{
        char *t;
        t = token;
        for(;*t;t++) cmd_lptr--;
}

/*-------------------look_up(s)------------------*/

/* look_up a token in the token table */
```

```
look_up(s)
char *s;
{
        int i,j;
        char *p;
        /* convert to a lower case */
        p=s;
        while (*p) {
                *p = tolower(*p);
                p++;
        }

        /* see if token is in table */
        for (i=0; *table[i].reserved;i++)
                if (!strcmp(table[i].reserved,s)) return table[i].tok;
        return 0;
}

/*-----------------isdelimiter----------------------*/
/* return true if c is delimiter */

isdelimit(c)
char c;
{
        if (strchr(";,+-/%<>(){}=",c) || c==' ' || c=='9' || c=='\r' || c==0)
                return 1;
        return 0;
}




/*----------------------------get_word-----------------*/
static char *get_word(cp)
char *cp;
{
        int wl=0;
        int fst=0;
        skip_white(&cp);
        while(*cp && *cp!='\n' && iswhite(*cp)==0 && *cp !=',' )
```

```
        {
                if (wl==NAMELEN && fst==0) {
                        err_flag=1;
                        fst++;
                } else
                        word[wl++]= *cp++;
        }
        word[wl]='\0';
        return(cp);
}

/*----------------------------get_line-----------------*/
static void get_line()
{
        *ln='\0';
        while(*ln=='\0' || *ln=='\n')
        {
                if (fgets(ln,420,stdin)==0)
                {
                        exit(1);
                }
                lnctr++;
        }
}

/*-------------------------replace_string---------------*/
static char *str_replace(str0,str1,str2)
char *str0;
char *str1, str2[];
{
        char *str3, *str4;
        *str4='\0';
l1:
        str0=get_word(str0);
        if (strncmp(word,";",1)==0 || tok == FINISHED ) goto l2;
        if (strncmp (str1,word,2)==0)
        {
                if (obj_typ == VARIABLE ) {
                        strcat(str4,"\" ");
                        strcat(str4,str2);
                        strcat(str4," \"  ");
```

```
                } else strcat(str4,str2);
                goto l1;
        }
        else;
        strcat(str4,word);
        strcat(str4," ");
        goto l1;
l2:
        return(str4);
}

/*-------------------------error-------------------------*/
error(i)
int i;
{
        switch (i) {
        case 1 :
                txt = " ";
                break;
        case 2 :
                txt = "  ";
                break;
        default:
                break;
        }

}
/*-------------------------clearscreen----------------*/
static void clearscreen()
{
        system("clear");
}

/*-------------------------quit-----------------------*/
quit()
{
        exit(0);
}
```