CHAPTER 2

A PERSPECTIVE AND COMPARATIVE REVIEW OF
OBJECT-ORIENTED DATA MODELS

2.1 Introduction

In this chapter a review of the research efforts in the area of object-oriented database modelling is presented [Bhalla 1991]. In Section 2.2, we discuss object-oriented concepts, abstraction mechanisms and their role in object-oriented databases. The advantages of object-oriented approach over record-based approach to data modelling is given in Section 2.3. An object-oriented data model is explained with examples in Section 2.4. A comparison of object-oriented and relational models is also given. In Section 2.5, a review of the salient features of sample prototypes is given. On-going efforts in creating object-oriented DBMSs are classified into three categories; (i) those that are directly based on the object-oriented paradigm; (ii) extensions to relational systems; (iii) experimental toolkits (or storage systems). This is followed by a comparison of these prototype models on the basis of criteria essential to database modelling in Section 2.6.
2.2 CHARACTERISTICS OF OBJECT-ORIENTED DATABASES

Object-oriented databases combine ideas from traditional database management systems, semantic data models, knowledge representation in Artificial Intelligence and object-oriented programming. Concurrency, persistence, etc., that come from traditional databases are not discussed here. From semantic data models and knowledge representation come data abstraction mechanisms and from object-oriented programming concepts of object identity, classes, encapsulation, etc. have been borrowed. In this section we discuss object-oriented concepts and abstraction mechanisms.

2.2.1 Object-Oriented Concepts

The object-oriented paradigm has gained wide acceptance as a unifying paradigm for the design of database systems, programming languages and knowledge-based systems. This paradigm has evolved with the advent of object-oriented programming languages. In this subsection, we discuss object-oriented concepts and their role in object-oriented databases [Atkinson 1989], [Bancilhon 1988], [Kim 1990], [Nierstrasz 1989].

2.2.1.1 Object and object Identity

Objects represent things or concepts from an application environment. Every object has a state and behaviour. The state of an object is the set of
attributes. An attribute refers to instance variables in object-oriented terminology. It corresponds to a column of a relation in relational databases. The behaviour of an object is the set of operations which operate upon the state of the object. Each object has a unique identity that does not change throughout its lifetime. The identity of an object is maintained by the system defined identifier. The object identifier is independent of values of attributes. In real world, we use names, such as Tom, Mary, and direct references such as pinpointing 'this person' to address a person. In object-oriented systems we use object identifier to refer to objects. In O₂ [Deux 1990] names can also be attached to objects, in addition to object identifiers.

2.2.1.2 Abstract Data Types and Classes

An abstract data type (ADT) describes the structure (attributes) and operations that are used to access the object. For example, we can define an ADT 'COMPLEX' whose attributes are 'real' and 'imag' parts. We can define operations such as finding complex conjugate and modulus of a complex number.

The concept of a class serves two purposes. First, it describes the structure of objects and operations that are used to access the object. This property is similar to ADT. Second, it represents a collection of objects having the same type. Objects belonging to a class are
called instances of that class (Figure 2.1).

Some object-oriented systems have only types [Fishman 1987], some have only classes [Kim 1989], and some have both types and classes [Deux 1990].

2.2.1.3 Class Hierarchy and Inheritance

The classes are organized in a hierarchy called class hierarchy. The notion of class hierarchy and inheritance provide a convenient way of organizing and describing relationships between objects. A class can be created from an existing class. The new class is called a subclass of the existing class which is called a superclass. A subclass inherits all properties (attributes and operations) from the superclass and it can have additional properties. For example, if there are two types of employees, manager and secretary, the new subclasses MANAGER and SECRETARY can be created as subclasses of class EMPLOYEE. The subclass MANAGER inherits all properties of EMPLOYEE and has additional property 'responsibility'. Similarly, the subclass SECRETARY inherits all properties of EMPLOYEE and has additional property 'typing-speed'. In this case all instances of a subclass are also instances of its superclass.
FIG. 2.1: CLASS - SUBCLASS LATTICE.
Object-oriented systems allow single inheritance or multiple inheritance. With single inheritance, a subclass inherits properties from only one superclass. The example given above illustrates single inheritance. In systems which support single inheritance, classes form a hierarchy. Multiple inheritance allows a class to inherit properties from more than one class. For example, a class STUDENT-EMPLOYEE inherits properties from classes STUDENT and EMPLOYEE. In systems which support multiple inheritance, classes form a lattice (Figure 2.1).

2.2.1.4 Messages and Methods

Object-oriented systems allow the state and behaviour of an object to be accessed or invoked only through messages. 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 or performing action (for example, 'draw' or 'print'). Thus, messages correspond to function calls and methods correspond to functions in conventional programming languages such as Pascal.

Some object-oriented databases [Fishman 1987], [Rowe 1987] do not support message passing scheme.
2.2.1.5 Class-attribute Hierarchy

In object-oriented systems, the domain of an attribute of a class is again a class. The domain class may be a primitive or a user-defined class. A primitive class such as integer, string or boolean has no attributes. A user-defined class has its own set of attributes and operations. For example, consider a class CIRCLE with attributes 'radius' and 'centre'. The domain of the attribute 'radius' is a primitive class, say, real. The domain of the attribute 'centre' is a class POINT with attributes 'x' and 'y' coordinates. The domain of these attributes is a primitive class. The hierarchy of classes arising from the fact that the domain of an attribute of a class is again a class is called class-attribute hierarchy.

2.2.2 Abstraction Mechanisms

Within the data model, abstraction mechanisms are used to form higher-level constructs. An abstraction mechanism also allows one to hide or ignore irrelevant details. These mechanisms are identified as Classification, Aggregation, Generalization and Association [Brodie 1984], [Smith 1977]. In this subsection, we describe these mechanisms and discuss how object-oriented concepts include these mechanisms.
2.2.2.1 Classification/instantiation

Classification is a form of abstraction in which a collection of objects with common properties form a class. The inverse of classification is instantiation. Between an object and a class there exists 'is-instance-of' relationship. Figure 2.2 shows two instances, (03,'Data Structure',3) and (04,'Artificial Intelligence',4) of class COURSE.

2.2.2.2 Aggregation

Aggregation refers to an abstraction in which component objects are aggregated to form a higher level object. Between component objects and their aggregate object there exists 'is-part-of' relationship. For example, PROJECT is an aggregate of pno, title and budget (Figure 2.3).

In object-oriented database systems, a class is an aggregation of its attributes. There exists 'is-part-of' relationship between an instance of a class and the values of the attributes of the class. For example, there exists 'is-part-of' relationship between objects '001', 'Simulation' and '20000' and the instance (001,'Simulation',20000) of class PROJECT.
FIG. 2.2: CLASSIFICATION

FIG. 2.3: AGGREGATION
2.2.2.3 Generalization/specialization

Generalization refers to an abstraction in which similar object classes are regarded as a generic object class. The inverse of generalization is specialization. There exists 'is-a' relationship between the specialized class and the generalized class. For example, the class PERSON is a generalization of classes EMPLOYEE and STUDENT. EMPLOYEE and STUDENT are specializations of PERSON (Figure 2.4).

The concept of superclass and subclass in object-oriented databases support this mechanism. A superclass is a generalization of the subclasses which inherit attributes from it. Conversely, a subclass is a specialization of the superclass from which it inherits properties.

2.2.2.4 Association

Association is a kind of abstraction in which a relationship between collection of objects is considered as a higher level set-object. Between the objects and the set-object there exists 'is-member-of' relationship. For example, objects French, German, Spanish and English are members of the set-object LANGUAGE (Figure 2.5).

In object-oriented databases this is generally supported as a well known abstraction mechanism called COLLECTION. In these systems, collection not only refers
FIG. 2.4: GENERALIZATION

FIG. 2.5: ASSOCIATION

(English, French, Spanish, Russian)
to sets but also to lists and bags. (Bags are multi-sets, i.e., sets that may contain duplicates.)

2.3 RECORD-BASED AND OBJECT-ORIENTED APPROACH TO DATA MODELLING

The object-oriented approach has several advantages over record-based approach. In this section some of the advantages are discussed.

In a relational system tuples can only be distinguished on the basis of values. In object-oriented systems a unique identifier is assigned to each object. This provides a means to support the relationship between objects and referential integrity constraints [Copeland 1984].

In object-oriented approach, an object retains its identity through arbitrary changes in its own state. Further, objects with common information can be modelled as two objects with a shared 'subobject' containing the common information. The sharing of objects reduces the update anomalies existing in relational data models [Maier 1989].

A relational model has been found to be lacking in semantic expressiveness for some applications. It is necessary to create relationship relations for modelling many-to-many relations. This results in a large number of relations which often have to be joined by queries. Also,
there is no supertype-subtype hierarchy. We cannot model the similarities between, say, employees and managers, so as to define common operations. Object-Oriented database systems are semantically more expressive than relational database systems.

Data structures in record-based systems do not support the actual structure of information in the real world. Thus, the structure of real-world information has to be simplified in the database schema and must be encoded into available data structures. Object-oriented database systems provide powerful data modelling capabilities through flexible data structuring [Kent 1979].

The data structuring capabilities of record-based systems do not adequately support the complexity and variations that occur in real data. Records of a given type are identical in structure. Every record of a given type must have the same field and a field must draw its value from the same type in each record. Object-oriented database systems allow variations in structuring objects to enable arbitrary data items as values [Kent 1979]. Also, in these systems the domain of an attribute is a type or any subtype of it.

The traditional database management systems do not support storage of multi-media data types [Derrett 1986], [Woelk 1987] such as text (used in variable length
strings), voice (used for digitized voice) and pictures (used in digitized images for graphic devices). The information systems of today are not only required to support these new data types, but are also required to provide operations which are appropriate for them. Object-Oriented approach allows new data types and suitable operations to be added to a DBMS.

Modelling complex design entities in relational systems, increases the levels of indirection between an entity and a subcomponent due to normalization. Re-assembling the components of an entity in the database requires taking joins involving few tuples from many different relations. Complex entities can be represented directly in object-oriented database systems with less encoding. To access such an entity requires fewer levels of mapping and the computation at each level is simpler [Maier 1986].

In the current record-based database systems, constraints on database updations except for simple constraints such as range checking have to be maintained by the applications. In object-oriented database systems the operations to maintain constraints can be put into the database management system itself.

The object-oriented framework provides better support for managing time and changes in databases [Dayal 1986]. A change in the value of an object is seen by all objects
which refer to it (referential transparency) [Zanilo 1986]. In relational systems, a change in key value of an entity is not propagated to other tuples that refer to it.

Most record-based systems do not provide a convenient or efficient way for users to access the history of database states. Object-oriented database systems can capture the history of database states as part of the data model, and support queries that access past states [Dayal 1986].

The database operations in an object-oriented data model can be stored within the database. This ensures that only one copy of each operation exists. Also, database access can be restricted through these operations. It reduces the risk of intentional corruption of data by programs [Derrett 1986]. The operations can be optimized to limit the number of calls to the underlying storage subsystems. Also, a database operation can be shared by many different applications [Lyngbaek 1987].

In traditional record-based database systems some re-compilation is required when schema of the database are modified. Some object-oriented systems support dynamic schema evolution [Banerjee 1987a].
2.4 OBJECT-ORIENTED DATA MODEL

The object-oriented data model discussed in this chapter consists of two parts: the conceptual schema and associated the data operations [Dittrich 1988], [Kim 1990], [King 1988], [Zaho 1988]. The conceptual schema represents the real world elements and relationships between them. Data operations include creation and manipulation of objects.

2.4.1 Conceptual Schema

The conceptual schema consists of objects and classes. All real-world elements are considered as objects. A primitive object such as an integer or string has no attributes. It only has a value, which is the object itself. More complex objects contain attributes through which they reference other objects which in turn, contain attributes. Each object is an instance of a class. A class acts as a typing mechanism. Class definitions are analogous to schema in traditional DBMSs, with the exception that classes associate operations with structure.

As discussed earlier, the domain of an attribute of a class is again a class. This leads to class-attribute hierarchy. If an attribute takes a single value, it is called a single-valued attribute, otherwise it is called a set-valued attribute. The value of a set-valued attribute is a set whose members are identifiers of instances of
the domain class of p. To illustrate how class-attribute hierarchy helps in data modelling, let us consider a database which contains information about departments in a typical university environment. For each department, the database contains code, head of department, courses offered and projects undertaken. The schema is shown in Figure 2.6. In this figure, a class is indicated by a box containing two sections. The first section contains the class name and the second section contains a list of attributes. A line with single arrow points to the domain class of the attribute. It also indicates that the attribute is single-valued. Similarly, a line with double arrows indicates that the attribute is set-valued. In this figure, the domain of attributes 'head', 'projects' and 'courses' of class DEPARTMENT are classes EMPLOYEE, PROJECT and COURSE respectively. The domain of attribute 'name' of EMPLOYEE is class EMPNAME. The classes identified alongwith their properties are given below:

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT</td>
<td>code : Integer</td>
<td>EMPLOYEE</td>
<td>idnumber : Integer</td>
</tr>
<tr>
<td></td>
<td>head : EMPLOYEE</td>
<td></td>
<td>name : EMPNAME</td>
</tr>
<tr>
<td></td>
<td>courses : set of COURSE</td>
<td>salary : Real</td>
<td></td>
</tr>
<tr>
<td></td>
<td>projects : set of PROJECT</td>
<td>designation : String</td>
<td></td>
</tr>
<tr>
<td>COURSE</td>
<td>courseno : Integer</td>
<td>PROJECT</td>
<td>projectno : Integer</td>
</tr>
<tr>
<td>Attributes</td>
<td>title : String</td>
<td></td>
<td>title : String</td>
</tr>
<tr>
<td></td>
<td>credits : Integer</td>
<td></td>
<td>budget : Real</td>
</tr>
</tbody>
</table>
FIG. 2:6: SCHEMA OF DEPARTMENT DATABASE.
It is well known that in relational models, the domain of an attribute is a primitive type. Also, in this model new relations have to be created for multi-valued attributes. The relational schema corresponding to the above object-oriented schema is given below:

DEPARTMENT(code#, headid#)
EMPLOYEE(idnumber#, name#, salary, designation)
EMPNAME(nameid#, title, firstname, middlename, lastname)
PROJECT(projectno#, title, budget)
COURSE(courseno#, title, credits)
DEPTPROJ(deptno#, projectno#)
DEPTCOURSE(deptno#, courseno#)

As discussed before, in object-oriented data models the classes are organized into a class hierarchy. At the top level of the class hierarchy is a class which has subclasses but no superclass. To illustrate the concept of class hierarchy and behavioural modelling, let us consider a database which contains graphical objects. The schema is shown in Figure 2.7. In this figure, a class is indicated by a box containing three sections. The first and second sections are same as in Figure 2.6. The third section contains a list of operations. Also, in this figure a double line point from a subclass to its
FIG. 2.7: SCHEMA OF GRAPHICAL DATABASE.
superclass. Each graphical object has attribute colour, and operations Draw and Fillpattern. The class GRAPHOBJ has three subclasses TRIANGLE, SQUARE and CIRCLE. The attribute colour, and operations Draw and Fillpattern are inherited by all subclasses. The Draw operation is quite specific to each subclass. It has to be re-implemented by the subclasses that inherits it. The subclasses add new attributes and operations. All instances of a class have the same behaviour because the same operations are applicable to them. The classes identified alongwith the properties are given below:

Class GRAPHOBJ  
Attributes  
colour : String  
Operations  
Draw  
Fillpattern  

Class TRIANGLE  
Superclass GRAPHOBJ  
Attributes  
vertices : set of POINT  
Operations  
Isosceles  
Innercentre  

Class CIRCLE  
Superclass GRAPHOBJ  
Attributes  
centre : POINT  
radius : Real  
Operations  
Area  
Enclose-origin  

Class POINT  
Attributes  
x : Real  
y : Real  

Class SQUARE  
Superclass GRAPHOBJ  
Attributes  
location : POINT  
side : Real  
Operations  
Area  
Perimeter  

The relational schema corresponding to the above object-oriented schema is given below:
An object-oriented data model supports simple objects and complex objects. Integer, string or boolean are simple objects. A complex object is made up of simple objects or other complex objects. A complex object is viewed as a tree, the object itself is the root node and its component objects are children of the tree. The leaf nodes of the tree are simple objects. The component objects are called dependent objects because their existence depends on the existence of the parent object. A complex object can be thought of as a structure or record in conventional programming languages such as C and Pascal. The hierarchy of classes to which complex object and its components belong is called a complex object hierarchy. In a complex object hierarchy the root node is a class to which the root object belongs. All nodes other than the root node are called component classes of the root class. For example, we define a complex object 'employee' as follows:
EMPLOYEE
{
  empno : integer
  name
    (title : string
         firstname : string
         lastname : string)
  qualification
    (academic
      (degree : string
       grade : string)
    professional
      (training : string
       duration : integer)
  )
}

In the above example, the complex object 'employee' consists of simple objects 'empno' and complex objects 'name' and 'qualification'. The complex object 'qualification', in turn, consists of complex objects 'academic' and 'professional' and so on. The root object 'employee' belongs to the class EMPLOYEE. The components 'name' and 'qualification' belong to classes NAME and QUALIFICATION respectively. In Figure 2.8, we show an example of a complex object, its components and classes to which the components belong. In this figure, class EMPLOYEE is an independent class. The classes NAME, QUALIFICATION, ACADEMIC, PROFESSIONAL are called dependent classes.
FIG. 2.8: COMPLEX OBJECT
2.4.2 Data Operations

In this section, we provide operations for creating and manipulating objects. We will use message syntax that is similar to Smalltalk [Goldberg 1983]. However, functions of C++ [Eckel 1989] can also be used.

Operations are performed by sending messages. A message is an expression of the form:

\(<\text{receiver} > <\text{selector} > (<\text{arg1} > <\text{arg2} > .. <\text{argn} >)\)

where

\(<\text{receiver} > \) is the object to which the message is sent;
\(<\text{selector} > \) is the name of the message. The name of the message is identical to the method to be invoked; and
\(<\text{arguments} > \) are the list of objects. Since a message returns an object, an argument can be a message expression.

Some important operations incorporated in object-oriented data models are given below. For exact syntax on the use of these operations the users can refer to the respective models. Names enclosed in"<>" indicate that they are to be specified by the users. Messages are written in uppercase letters. Keywords are preceded by a colon [Goldberg 1983].
(i) To create a class:

```
<classname> CREATE-CLASS (:superclass {<superclassname>})
 (:attributes {<attributel (:domain <domain1>)
                  <attribute2 (:domain <domain2>)
                  ........
                  <attributek (:domain <domaink>)}
```

(ii) To create an instance of a class:

```
<class> CREATE-INSTANCE :attributel value1
                  :attribute2 value2
                  ........
                  :attributek valuek
```

(iii) To get a set of objects satisfying a boolean condition:

```
<class> SELECT Boolean-Expression
```

(iv) To delete objects satisfying a boolean condition:

```
<class> DELETE Boolean-Expression
```

(v) To delete a particular object:

```
<object-id> DELETE-INSTANCE
```

(vi) To assign a value to an attribute of a class:

```
<class> :at <attribute> :put <value>
```

(vii) To get the value of an attribute:

```
<class> attribute
```

In addition to the above operations the following operations for manipulating objects are also supported.

(i) C MEMBER x returns a boolean which is true if x is a member of class C;

(ii) C INSTANCES returns set of instances of class C;

(iii) S COUNT returns the number of members in set S.

(iv) x IDENTICAL y returns a boolean which is true if x
and y are identical objects;
($(v)$)

$(x \text{ EQUAL } y)$ returns a boolean which is true if $x$ and $y$ are equal objects;
($(v)$)

(see remark 2.4.2.1)

Remark 2.4.2.1: Two objects are equal if they have same attributes and the values of the corresponding attributes are equal. If two objects have same identifier then we say that they are identical, in fact they are same objects.

2.5 STATE OF THE ART OF CREATING OBJECT-ORIENTED DATABASES

A number of research efforts are in progress in creating object-oriented databases. Some prototype efforts such as ORION, OZ+, CACTIS and IRIS aim at creating advanced DBMSs based on the object-oriented paradigm. Another direction of object-oriented database research is being concentrated on the extensibility of relational systems. Prototype efforts such as POSTGRES and STARBURST attempt to provide extensions to the relational systems. In the third category of efforts such as EXODUS, system designers are experimenting using a "toolkit" approach, i.e., software tools and libraries to generate a database system specific to the application domain. A representative collection of these prototype efforts in the above mentioned categories is discussed below.
2.5.1 Systems based on object-oriented paradigm

2.5.1.1 ORION

The prototype object-oriented database system, ORION [Banerjee 1987], [Kim 1989], [Kim 1990a], is being developed at MCC. Objects are supported as instances of classes. Classes are organized into a class lattice. This supports generalization. Instance variables and methods are inherited through the class lattice.

In addition to the generalization hierarchy, a class may be involved in class-attribute hierarchy and complex object hierarchy. In class-attribute hierarchy the value of an attribute \( p \) of class \( C \) is an instance of class \( C' \) if \( C' \) is the domain of \( p \) and it is a primitive class. If \( C' \) is a non-primitive class then the value of \( p \) is an identifier of an object belonging to class \( C' \). The values of the attribute collectively form an aggregate object. This supports aggregation.

In a complex object hierarchy, the attribute of a class \( C \) owns the object (instance) belonging to the dependent class. A dependent object can be owned by one parent object, but can be referenced by any number of objects. The component classes of a complex object hierarchy can be used by the user as any other user-defined class. Also, a component class can have more than one parent. That is, if two objects \( o' \) and \( o'' \) belonging to classes \( C' \) and \( C'' \) contain objects \( d' \) and \( d'' \)
respectively, then \(d'\) and \(d''\) may belong to the same class. The system ensures that \(d'\) and \(d''\) are not identical objects.

2.5.1.2. OZ+

The research in the object-oriented database system, OZ+ [Weiser 1989], is being carried out at University of Toronto, Canada. The objects are supported as instances of classes. All instances of a class share the same instance variables and rules. An object’s operations are referred to as rules. Rules are also treated as objects.

Rules consist of conditions and actions. They are invoked by messages. For each rule some classes are specified and a rule can accept messages from instances of these classes only. A rule for which no such class is specified invokes itself and performs its actions when all its conditions are true. Such rules are referred to as self-triggering rules.

Classes are organized into a class hierarchy. This supports generalization. Instance variables and rules are inherited through class hierarchy. Single inheritance is supported.

Complex objects are supported as list, set, array and tree types. In a complex object of tree type, the root object is an independent object and all its children are dependent objects. The root class of complex object
hierarchy is an independent class and all other classes are dependent classes. A dependent class cannot be used by the user as in case of ORION. Therefore, a dependent object is owned by a parent and cannot be referenced by any other object.

2.5.1.3 IRIS

The IRIS data model [Derrett 1986], [Fishman 1987], [Lyngbaek 1986] is developed at Hewlett-Packard Laboratories. The data model is based on three constructs, objects, types and functions. Objects are supported as instances of types. Properties of objects and relationships among objects are expressed in terms of functions. Objects can be accessed and manipulated only by means of functions [Derrett 1985] defined on object types. Objects serve as arguments to functions and may be returned as results of functions.

As an example, a function 'head' may be defined on objects of type Department as follows:

head : DEPARTMENT → EMPLOYEE

The function head(Mathematics), then returns the head of the Mathematics department.

Functions may be set-valued. For example, the function

projects-of : DEPARTMENT → PROJECT

returns a set of projects undertaken by a department.
Types are organized into a type hierarchy which supports generalization and specialization. A given type may have multiple supertypes. Attributes and functions are inherited through generalization hierarchy. Types are also treated as objects. IRIS does not support classes. It supports functions which allow a user-created object to be added to a user-defined type and all its supertypes. It also supports function which returns all instances of a given type.

2.5.1.4 CACTIS

The CACTIS research project [Hudson 1989] is being undertaken at the University of Colorado. CACTIS development is based on two areas of database research, namely, semantic data modelling and object-oriented data modelling.

The data in the database is held in an attribute graph which is structured like a conventional network model. Each node in the graph is an instance of a particular type of data. Objects are supported as instances of types.

The object types are organized into a type hierarchy. This supports generalization. Types also describe the number of relationships and type of relationships that instances of a type have with instances of other types. Objects are therefore connected recursively by typed relationships, thereby supporting aggregation and complex
objects.

The model allows the value of an attribute of a type \( t \) to be derived by a function defined on the type \( t \) and also by functions defined on attributes of type \( t' \), if \( t' \) is a relationship type connected to type \( t \). Derived attributes capture the concept of behavioural modelling.

The data model allows constraints to be attached to attributes. A constraint is implemented as a derived attribute whose value is boolean. This indicates whether the constraint has been violated.

\[ \text{2.5.2 Extensions of Relational Systems} \]

\[ \text{2.5.2.1 POSTGRES} \]

The POSTGRES project [Rowe 1987], [Stonebraker 1988], [Stonebraker 1990] is being undertaken at University of California. It supports the concept of object identifier because each tuple is assigned a unique identifier. The value of an attribute of a tuple can be an identifier of another tuple. POSTGRES is extensible because it allows user to build new attributes as ADTs. That is, the system allows a column of a relation to be an ADT. Also the domain of an attribute of a relation can again be a relation. This is similar to class-attribute hierarchy. Objects are supported as relations with these special attributes or a tuple of such a relation.
It allows users to define functions whose operands can be primitive data types, ADTs, or relations. Another kind of functions available in POSTGRES are known as POSTQUEL functions. These functions are defined as a collection of commands in POSTQUEL query language. A POSTQUEL function enables tuples to be returned as the value of the function. Another advantage of this function is that it can be specified as the domain of an attribute of a relation. This enables the value of an attribute to be fetched from other relations, thereby supporting complex objects.

The system supports relation hierarchy. This supports generalization. A relation inherits all attributes from its parents. Key specifications are also inherited. Functions defined on a relation are also inherited through relation hierarchy.

2.5.2.2 STARBURST

The STARBURST project [Haas 1990], [Schwarz 1986] is being undertaken at IBM Almanden research centre. The goal of the project is to examine different ways by which the traditional relational database systems can be extended to meet the requirements of new applications and technologies. The project is considering several areas for extensibility. In this subsection we discuss abstract data types and complex objects.
The system supports three kinds of ADTs. These are scaler types, structured types and complex types. New operations can be defined on these types. Collection of fields can be treated as a unit. For example, the representation of 'date' type may be a collection of values, month, year and day. It is possible to reference the entire quantity as one unit. This supports aggregation. Multi-dimensional arrays form structured constructs for storing data. The project is examining different ways so that an entire array or a portion of on array can be referenced efficiently.

The system allows arbitrary collection of tuples from one or more relations to be referred as a unit, thereby supporting complex objects. The project is considering several approaches to handle complex objects. The first is to store complex objects as un-interpreted data in long fields. The second is to allow a structured set of records to be returned as a result of a query. The last is to allow records to have fields that are themselves relations.
2.5.3 Toolkit-Based Approach

2.5.3.1 EXODUS

The research project EXODUS [Carey 1986] is being undertaken at the University of Wisconsin. This prototype effort has been categorized as the "DBMS generator" approach. The goal of the project is to implement a storage system to support ADTs, access methods, operations and version control.

Storage objects are the basic unit of data. A storage object is an un-interpreted byte sequence of any size. Each storage object is assigned a unique identifier. A class defines storage objects consisting of same fields. A field of an object may be a primitive type such as integer, real, or an ADT such as a complex number or a constructed type such as array, record, set or bag. Constructed types can be accessed through predefined operations available on them. Classes are organized into a class lattice. A class inherits fields from its superclasses. This supports generalization.

The system provides a procedural interface. It supports procedures that enable reading and writing a subrange of bytes of a storage object. Also, there are procedures that allow insertion and deletion of bytes from a specified offset. The interface also include procedures to create and destroy objects. To aid application specific DBMS developers, libraries of useful routines for
the extensible components of the system are provided.

2.6 COMPARISON OF THE SYSTEMS

Object-oriented DBMSs aim to support advanced data modelling features. These support structures and operations associated with the representation, and storage of complex objects and their relationships. The design goals and approach to implementation of various systems differ from each other. Various authors [Hull 1987], [Peckham 1988] have attempted to provide some basis of comparison for semantic models. On similar lines, a reviewed comparison of object-oriented models is summarized in Table 2.1.
<table>
<thead>
<tr>
<th>Application Environment</th>
<th>Approach to Implementation</th>
<th>Representation of objects</th>
<th>Constructs for building complex data</th>
<th>Structural Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD, CAM, AI; Office Information Systems</td>
<td>The storage sub-system provides access to objects on disk, manages the allocation and deallocation of segments of pages on disk.</td>
<td>Supports objects as instances of classes.</td>
<td>List, array.</td>
<td>Class structure; Supports generalization and aggregation.</td>
</tr>
<tr>
<td>Support</td>
<td>It utilizes existing relational DBMS to manage objects in disk memory. Class definition is encoded into a set of relations. Objects are encoded into relational tuples.</td>
<td>Supports objects as instances of classes and rules.</td>
<td>List, set, array and tree.</td>
<td>Class structure; Supports generalization and aggregation.</td>
</tr>
<tr>
<td>IRIS</td>
<td>Object-Oriented data model is implemented by an object manager, built on top of relational storage sub-system.</td>
<td>Supports objects as instances of types.</td>
<td>Supports complex numbers and arrays.</td>
<td>Type structure; Supports generalization and specialization and aggregation.</td>
</tr>
<tr>
<td>CACTIS</td>
<td>Objects on mass storage are structured as a header, a block of relationship pointers, and a block of storage for attribute values. The header of each object contains index into the</td>
<td>Data is held in an attribute graph which is structured like a conventional network model. Each node in the graph is an instance of a data type.</td>
<td>Supports arrays and records.</td>
<td>Class structure; Supports generalization.</td>
</tr>
<tr>
<td>ORION</td>
<td>Extends the relational system with ADTs and functions as a fundamental data type.</td>
<td>Objects are represented as tuples whose attributes may be ADTs.</td>
<td>Supports arrays.</td>
<td>Relation hierarchy; Supports generalization and aggregation.</td>
</tr>
<tr>
<td>ORION</td>
<td>Supports a storage object manager at the low-level, which provides capabilities for reading, writing and updating storage objects.</td>
<td>Objects are represented as tuples whose attributes may be ADTs.</td>
<td>Supports arrays.</td>
<td>Objects composed of arbitrary collection of records from one or more relations; Supports aggregation.</td>
</tr>
<tr>
<td>ORION</td>
<td>The basic unit of data is the storage object (referred by an identifier) whose fields may be ADTs.</td>
<td>Objects are represented as tuples whose attributes may be ADTs.</td>
<td>Supports arrays.</td>
<td>Aggregation is supported by using ADTs.</td>
</tr>
</tbody>
</table>

Table 2.1: SUMMARY OF OBJECT-ORIENTED DATA MODELS
<table>
<thead>
<tr>
<th>Behavioural Abstraction</th>
<th>ORION</th>
<th>OZ+</th>
<th>IRIS</th>
<th>CACTIS</th>
<th>POSTGRES</th>
<th>STARBURST</th>
<th>EXODUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects respond to a set of messages defined over the class. The implementation of a class provides methods which respond to the messages.</td>
<td>Objects can only be accessed through the operations referred to as rules defined over classes. Rules are invoked by messages.</td>
<td>Objects can only be accessed through the operations defined on object types. Operations are implemented as methods which respond to messages.</td>
<td>Each object responds to a set of messages defined in the object class. The implementation of a class provides methods which respond to messages.</td>
<td>Functions are globally available.</td>
<td>Functions are the main constructs for referring to objects.</td>
<td>Algorithms are associated with large storage objects for various data operations.</td>
<td></td>
</tr>
</tbody>
</table>

| Inheritance | Class hierarchy through which attributes and methods are inherited. Multiple inheritance is supported. | Class hierarchy through which attributes and rules are inherited. Single inheritance is supported. | Type hierarchy through which attributes and functions are inherited. Multiple inheritance is supported. | Type hierarchy through which attributes and functions are inherited. Multiple inheritance is supported. | Relation hierarchy through which attributes and functions are inherited. Multiple inheritance is supported. | Class hierarchy through which storage objects inherit fields. Multiple inheritance is supported. |

| Data Operations | Operations are implemented as methods written in Common Lisp. | Operations are implemented as conditions and actions. | Operations are implemented as functions written in C programming language. | Operations are implemented as functions written in C programming language. | Functions are supported as database objects. Supports user-defined functions and integrity constraints. | Relational query language has been extended to allow use of functions in different contexts. |

| Salient features | Dynamic schema evolution, multi-media information management. | Rules may be self-triggering, i.e., it performs actions when all its conditions become true. | Functions have specifications for each of their arguments and result parameters which indicate a minimum and maximum object participation. | Ability to attach constraints to attributes. These are implemented as triggers. | Supports data base procedures for enforcing integrity constraints. | The prototype aims at managing large storage objects. A rule based query optimizer generator has been included in the design. |

Table 2.1 (cont.): SUMMARY OF OBJECT—ORIENTED DATA MODELS