NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A CONCEPTUAL APPROACH TO OBJECT-ORIENTED DATA MODELING

by

Gerald Byron Barnes

September, 1994

Thesis Advisor: C.Thomas Wu

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE Sep 1994	3 REPORT TYPE AND DATES COVERED Master's Thesis, Final

4 TITLE AND SUBTITLE A CONCEPTUAL APPROACH TO OBJECT-ORIENTED DATA MODELING	5 FUNDING NUMBERS
6 AUTHOR(S) Gerald B. Barnes	
7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8 PERFORMING ORGANIZATION REPORT NUMBER
9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10 SPONSORING/MONITORING AGENCY REPORT NUMBER

11 SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT *(maximum 200 words)* Object-oriented data modeling is starting to replace the relational model for many recently emerging database applications. The complex nature of these databases precludes mapping of their data directly into a tabular relational structure. Current object-oriented data modeling lacks the standardization and mathematical soundness of the relational model. This thesis addresses this problem by proposing a conceptual data model called OPERA (Object Paradigm / Entity-Relationship Approach). OPERA incorporates the static features of the Entity-Relationship Model with the dynamic properties of object-orientation. In addition to OPERA, an object-oriented extension to the graphical query language GORDAS (Graph-Oriented Data Selection) is proposed. To demonstrate the effectiveness of the proposed model, we will model a United States combat systems support database, the EWIRDB (Electronic Warfare Integrated Reprogramming Database). We map the EWIRDB from its basic relational format to an object schema and then to an OPERA graph. Finally, this conceptual schema is mapped to a GORDAS schema graph and queries are performed on the database. OPERA is conceptually superior to the ER Model and its object-oriented variant, the Enhanced Entity-Relationship Model (EER) Model. We demonstrate this by representing methods as relationship types, which the ER and EER models are incapable of. OPERA also aids in query formulation for visual query languages such as GORDAS by providing a query graph mapping template.

14. SUBJECT TERMS Object-Oriented Data Model, Enhanced Entity-Relationship Model, Conceptual Data Model, Electronic Warfare Integrated Reprogramming Database(EWIRDB), Object Paradigm/Entity Relationship Approach(OPERA), Graphical Object-Oriented Data Selection(GORDAS)	15. NUMBER OF PAGES 98
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release, distribution is unlimited

A Conceptual Approach to Object-Oriented Data Modeling

by

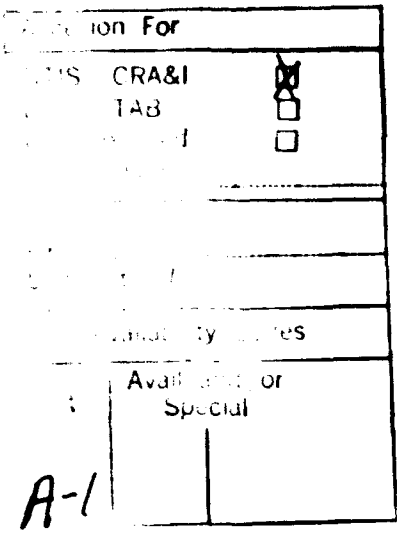
Gerald Byron Barnes
Lieutenant, United States Navy
B S , University of Alabama, 1981

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1994



Author: _____
Gerald B. Barnes

Approved by: _____
C. Thomas Wu, Thesis Advisor

_____
Craig W. Rasmussen, Second Reader

_____
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

Object-oriented data modeling is starting to replace the relational model for many recently emerging database applications. The complex nature of these databases precludes mapping of their data directly into a tabular relational structure. Current object-oriented data modeling lack [illegible]dardization and mathematical soundness of the relational model. This thesis addresses this problem by proposing a conceptual data model called OPERA (Object Paradigm / Entity Relationship Approach). OPERA incorporates the static features of the Entity-Relationship (ER) Model with the dynamic properties of object-orientation. In addition to OPERA, an object-oriented extension to the graphical query language GORDAS (Graph-Oriented Data Selection) is proposed.

To demonstrate the effectiveness of the proposed model, we will model a United States combat systems support database, the EWIRDB (Electronic Warfare Integrated Reprogramming Database). We map the EWIRDB from its basic relational format to an object schema and then to an OPERA graph. Finally, this conceptual schema is mapped to a GORDAS schema graph and queries are performed on the database.

OPERA is conceptually superior to the ER Model and its object-oriented variant, the Enhanced Entity-Relationship (EER) Model. We demonstrate this by representing methods as relationship types, which the ER and EER models are incapable of. OPERA also aids in query formulation for visual query languages such as GORDAS by providing a query graph mapping template.

ACKNOWLEDGMENTS

Many individuals offered assistance to me during the course of writing this thesis. I would like to thank Professor Craig Rasmussen, who not only spent a great deal of time reviewing this work as second reader, but also made mathematics fun instead of just demanding. A special thanks goes out to Professor Tom Wu, who was a dedicated mentor from start to finish and whose sense of humor was only rivaled by his intellect.

To my most trusted ally, Joyce, thank you for the last three years of support you have given me while I pretended to toil.

I. INTRODUCTION

A. BACKGROUND

A *data model* is a set of concepts which may be used to describe the structure of a database, and also a set of operations which may be performed on it (Elmasri, 1989). Up until the early 1960's, the only model for storing data was the traditional file system. With the advent of the hierarchical and network data models in the mid to late 1960's, the idea of a separate *database management system* (DBMS) evolved as a set of programs which could enable users to create and maintain a set of related data called a database.

In 1970 the relational data model was introduced, and has become a standard for many DBMS applications. A relational DBMS stores data in a logical tabular arrangement, and its implementation is rather straightforward, since relational tuples and file records are conceptually the same. In addition, the relational model allowed several high-level query languages to be developed in the 1970's, including SQL, QBE, and QUEL, which were an improvement over the data manipulation language (DML) of the network and hierarchical models. By 1976, a conceptual model was introduced, the Entity-Relationship (ER) Model, which could be used to map high-level database schemas directly into a relational, hierarchical, or network implementation schema.

Although relational database management systems (RDBMS) have generally been adequate for traditional business applications, they have proven insufficient for others. Multimedia, computer-aided manufacturing and design (CAD/CAM) and artificial

intelligence knowledge databases all have complex modeling and storage requirements which the typical RDBMS cannot satisfy. The primary reason for this is that the relational model forces segmentation of related data. That is, data which are spread out among several relations (i.e., tables) lose their meaning as the database becomes more complex. This is incompatible with advanced modeling concepts such as attribute inheritance and aggregation found in object-oriented systems. For this reason, a new data model, the object-oriented model, was developed for such applications.

Object-oriented data models are superior to the relational model in terms of their abstraction capabilities. Unlike the relational model, data which describe a real-world entity is consolidated in a single database object, which makes object-oriented modeling more compatible with advanced database semantics. Unfortunately, object-oriented modeling suffers from the fact that there is no standard through which all complex databases may be represented.. The concept of the relation (Rosen, 1991), which is the basis for RDBMS implementation models, and also integral to the ER Model of Chen (1976), has become popular as a conceptual tool for database design. What is needed for the object-oriented data model is a similar benchmark which provides a conceptual framework for object-oriented database design and modeling, and which is based on an established mathematical theory.

Since Chen's original paper on the ER model was published , there have been a number of research papers discussing extensions to the basic model. Smith (1977) introduced the abstract concepts of aggregation and generalization. Scheuermann (1979)

showed that ER relationship types could be modeled as higher-level aggregate objects. Dos Santos (1979) derived complex data types from Chen's entity sets using mathematical constructors. Elmasri (1989) incorporates these and other enhancements to the ER model in an excellent discussion.

Hughes (1990) provides a good overview of object-oriented data modeling. The graphical representation of object schemas used in this thesis is based largely on Bertino (1993). A thorough examination of a representative object-oriented data model (ORION) is found in Banerjee (1987). Bertino (1992) discusses the impact of the object-oriented data model on query language design. The conceptual query language used in this thesis is an extension of that proposed in Elmasri (1981).

A number of papers have been devoted to merging the ER and object-oriented models. Kappel (1988), Lazimy (1989), Navathe (1988), and Gorman (1990) all propose conceptual models which incorporate the static features of the ER model with the dynamic ones of the object-oriented paradigm. The model developed in this thesis incorporates ideas from these papers, but proposes a different method for representing the behavioral aspect of the object model.

There are several issues addressed by this thesis. First, a general data model for an object-oriented database schema must be established . Given this model, an abstract device should be developed to represent it in an understandable and meaningful way. We accomplish this by studying whether an existing or modified conceptual data model may be mapped to an object-oriented schema. Assuming this can be done, the utility of an

object-oriented conceptual model must be examined. To do this, we see if the model serves a purpose or has some application.

The objective of this research is to extend the ER model into a high-level graphical representation for the object-oriented model, and thereby establish a common mathematical foundation for the design of object-oriented database management systems (OODBMS). In this thesis, a data model is proposed for representing object-oriented schemas as abstractions. The model is graphical in nature; as such, it is a visual aid in understanding the semantics of an object database. Constraints are modeled in a diagrammatic fashion instead of a procedural one, and operations are discussed from the viewpoint of data retrieval only (queries). The example military database which is chosen for modeling is classified; however, only those elements which are unclassified have been used for this study. It is assumed that the reader has a basic understanding of the ER model.

B. ORGANIZATION

This thesis is composed of seven chapters. Chapter I is an introduction containing information on the reasons for the research and the methods for accomplishment. Chapter II discusses the Enhanced Entity-Relationship (EER) Model as an object-oriented extension of the ER model. Chapter III provides a general overview of object-oriented data modeling. Chapter IV proposes a conceptual model called OPERA (Object Paradigm / Entity-Relationship Approach) for representing an object-oriented database schema. Chapter V compares different query languages for the object-oriented data model and

proposes an extension to a graphical query language (GORDAS) to support object queries. Chapter VI uses the results of Chapters II through V to map a military application database, the Electronic Warfare Integrated Reprogramming Database, to an object schema and then an OPERA diagram. Chapter VII concludes with a review of thesis objectives and recommendations for further research.

II. THE ENHANCED ENTITY-RELATIONSHIP MODEL

The Enhanced Entity-Relationship Model (Elmasri, 1989), or EER, is an extension of the ER model which includes all of the concepts of Chen's model plus the following:

1. Superclass
2. Subclass
3. Specialization
4. Generalization
5. Category
6. Attribute Inheritance

The notion of abstract classification is demonstrated in the EER model in terms of superclasses and subclasses. An entity type in the ER model is represented as a superclass in the EER model. A particular superclass may be a subclass of another entity type (which may be a superclass in its own right). Hence, a relationship exists between the two, called a *superclass/subclass relationship*. The significance of this relationship lies in the fact that all characteristics which describe the superclass also pertain to all of its subclasses; however, a particular subclass may exhibit characteristics which are in addition to those of its superclass , but also exclusive of other subclass types. To illustrate these concepts, refer to Figure 1. This diagram represents a database containing information about a company's employee base and how vehicles are assigned to it. Right away we notice a significant difference from the ER model - the representation is hierarchical in nature. Another difference is more subtle - the entity types EMPLOYEE, MANAGER, and ENGINEERING MANAGER are now referred to as superclasses or subclasses,

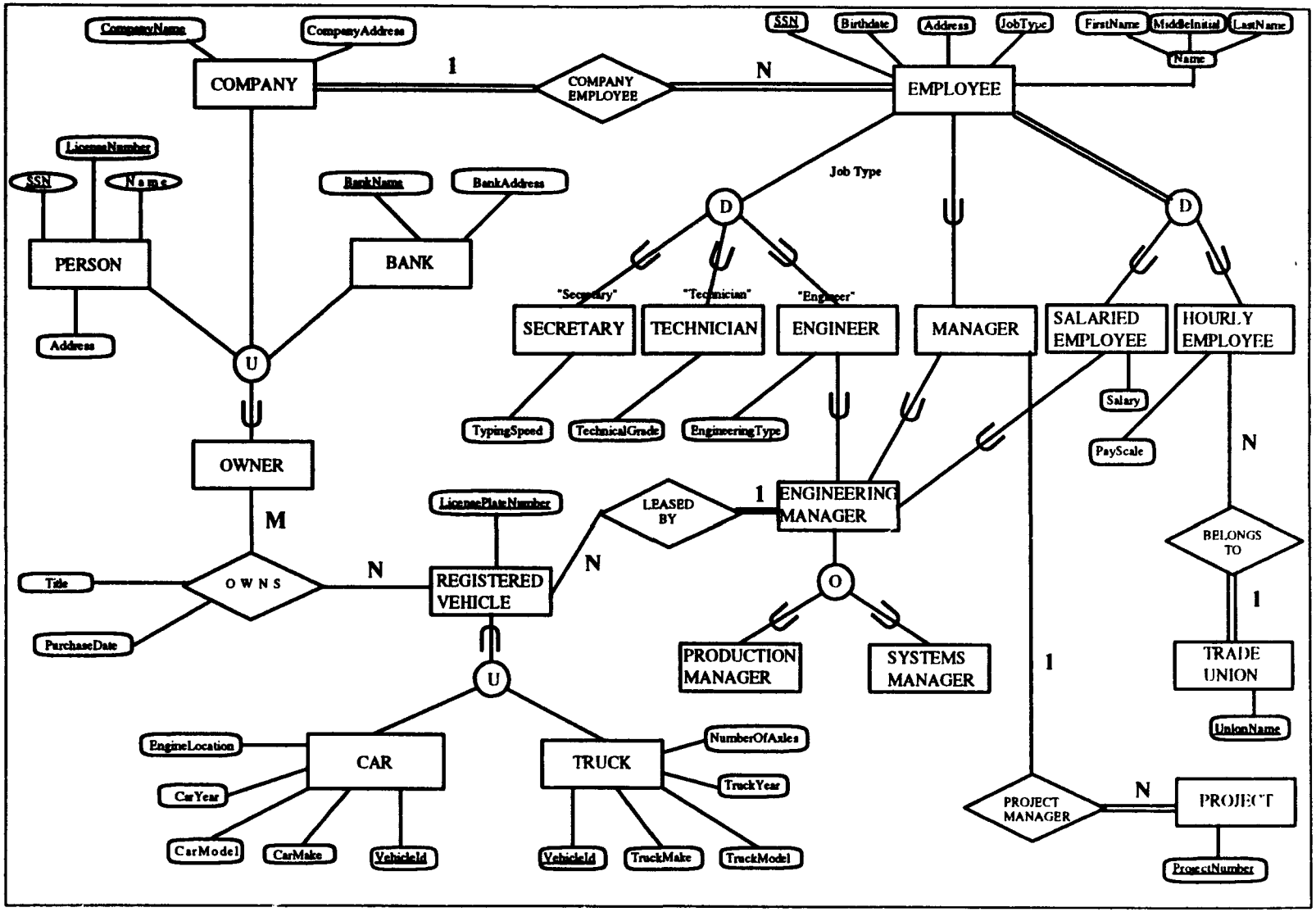


Figure 1. EER Schema

depending on their involvement in a specific relationship (the symbol {U} means subclass) For example, EMPLOYEE is a superclass of MANAGER, while MANAGER is a superclass of ENGINEERING MANAGER. In addition, PRODUCTION MANAGER and SYSTEMS MANAGER are both subclasses of ENGINEERING MANAGER. This dualism would be hard to represent in standard ER form. The important observation about such a hierarchy is that all subclasses descended (i.e., derived) from the root class (in this case, EMPLOYEE), inherit all of its characteristics. This feature is known as *attribute inheritance*. Hence, a SECRETARY is described by all the EMPLOYEE attributes (SSN, Name, Job Type, Address, Birthdate) as well as its own specific one (Typing Speed).

The inheritance paradigm is also applicable to relationships: if a superclass/subclass is related to another superclass/subclass not in its hierarchy, that relationship is also inherited by all its successors. To illustrate, the EMPLOYEE class participates in a relationship named COMPANY EMPLOYEE with another class (in a separate hierarchy) called COMPANY. This relationship, as shown in Figure 1, shows that a member of the COMPANY class may contain one or more members of the EMPLOYEE class, but must contain at least one. Conversely, an EMPLOYEE must belong to only one COMPANY. Since SECRETARY, TECHNICIAN, ENGINEER, MANAGER, SALARIED EMPLOYEE, and HOURLY EMPLOYEE are subclasses of EMPLOYEE, they all participate in this relationship as well. An important aspect of this model is that a subclass, even though it has an independent existence with respect to some other classes in the hierarchy, cannot exist as a member by itself. If a class is a subclass of another class, it is

also a member of that superclass This is the *subset hierarchy definition* of superclass/subclass relationships an entity A is a subset of entity B if every occurrence of entity A is also an occurrence of entity B (Teory, 1986).

Specialization, as applied to the EER model, is the process of further classifying a set of objects (i.e., a superclass) into more specialized subclasses. This specialization is usually defined by some distinguishing characteristic of the entities which comprise the superclass. To illustrate, in Figure 1 we have a superclass, EMPLOYEE, which contains two specializations, one based on job type, another on method of payment. The set {SECRETARY, TECHNICIAN, ENGINEER} contains entities which are employees but are identified in a specialized role based on the job performed. Likewise, {SALARIED EMPLOYEE, HOURLY EMPLOYEE} is a set of instances specialized based on their type of income. The other subclass, MANAGER, is not a specialization, but a simple subclass of EMPLOYEE. One interesting thing to observe about this figure is the relationship BELONGS TO. In the ER model, this would have to be represented as a relationship between EMPLOYEE and TRADE UNION; this is not as expressive as the EER depiction. In the EER model, because of specialization, we know exactly what group of employees may belong to such an organization: those who are hourly employees, and no others.

Generalization in the EER model is simply the inverse process of specialization; it is the process of grouping several classes into a higher-level abstract class that includes objects in all these classes. Both specialization and generalization are similar; the former is

a top-down conceptual refinement and the latter is a bottom-up conceptual synthesis Otherwise, they are equivalent

As seen in Figure 1, a class may be shared by one or more higher-level classes. Such a class is called a *shared subclass*. For example, the subclass ENGINEERING MANAGER is shared by ENGINEER, MANAGER, and SALARIED EMPLO... Each of the superclass/subclass relationships involving a shared subclass is distinct. ENGINEER/ENGINEERING MANAGER, MANAGER/ENGINEERING MANAGER, and SALARIED EMPLOYEE/ENGINEERING MANAGER are separate relationships, and as such ENGINEERING MANAGER inherits the attributes (i.e.,characteristics) of its three superclasses all the way to the root of the hierarchy (EMPLOYEE). A structure which contains a shared subclass is called a *specialization lattice*; if no shared subclasses are present, a *specialization hierarchy* exists.

There are some situations where a lattice is not desired, that is, instead of a shared subclass, we need to model a relationship where one subclass is related to multiple superclasses in a single superclass/subclass relationship. How can this be done? The *category* provides the answer. In Figure 1, vehicle registration for certain company employees is modeled by the LEASED BY and OWNS relationships. These show that only company employees who are engineering managers may lease a registered vehicle, which in turn belongs to some other entity. The owner of a vehicle may be a person, bank, or a company, but does not have to be all three. To represent this relationship properly, the category OWNER is established, which is a subclass of the union of its superclasses

PERSON, BANK, and COMPANY. This union of superclasses is depicted graphically by a connecting node containing the letter "U". The important detail to note about categories is that they provide for *selective inheritance*; that is, a category only inherits attributes from the superclass to which it is related. To illustrate, if a particular owner happens to be a person, the attributes for BANK will not be passed along to that instance of OWNER. This was not possible with a shared subclass, since attributes of all the superclasses for a shared subclass are inherited. While a category is the union of its related superclasses, a shared subclass is the intersection of its superclasses; each has their modeling applications.

As was discussed earlier, a most important component of any data model is its set of built-in constraints. In addition to the constraints which are part of the ER model, the EER model contains the following additional constraints on specialization/generalization:

1. Subclass Membership
2. Disjointness
3. Completeness

As seen earlier, more than one specialization may exist within a hierarchy. The specialization structure may be refined further to specify membership in a particular subclass. There are three categories of subclass membership constraints which accomplish this: *predicate-defined*, *attribute-defined*, and *user-defined*. With predicate-defined subclasses, membership is determined by the value of some superclass attribute. The value of the superclass EMPLOYEE attribute Job Type is known as the *defining predicate* of the subclass, since it defines whether an entity will belong to the SECRETARY, TECHNICIAN, or ENGINEER subclasses. The defining predicate condition is a

constraint which is indicated graphically by placing its value along the arc leading from the connecting node to the subclass. A second type of subclass membership constraint is the attribute-defined constraint. This is similar to the predicate-defined constraint, except that membership is determined by the value of the same superclass attribute for all subclasses. This attribute is called the *defining attribute* of the specialization (or generalization), and is depicted by showing its name on the arc leading from the superclass to the connecting node. For the schema of Figure 1, the attribute Job Type of EMPLOYEE is the defining attribute for {SECRETARY, TECHNICIAN, ENGINEER}. A third type of subclass membership constraint is *user-defined*. With user-defined membership, there is no built-in condition which determines that an entity will belong to a subclass; the database user sets this constraint himself. Such a constraint is specified individually for each entity entered into the database.

In order to make the EER model complete, constraints must be specified on membership within a specialization. In our example schema, the absence of any such constraints might allow a member of the EMPLOYEE superclass to simultaneously exist as both a SALARIED EMPLOYEE and HOURLY EMPLOYEE. This is not an accurate reflection of the real world, for no employee of any company is salaried and paid on an hourly wage scale at the same time. To resolve this dilemma, an additional constraint on *disjointness* is provided. This constraint specifies whether duplicate membership is allowed in a specialization. The disjointness constraint has two possible values, either disjoint or overlapping. Disjointness is indicated graphically by the presence of the symbol (D) in a

connecting node, and overlapping subclasses are indicated by the presence of the symbol (O). Subclasses which are overlapping within a specialization imply that the same entity may coexist in distinct subclasses, whereas disjoint subclasses must not allow this. In Figure 1, the ENGINEERING MANAGER superclass contains two overlapping subclasses: PRODUCTION MANAGER and SYSTEMS MANAGER. This tells us that an engineering manager may be in charge of systems, production, or some combination of the two. Similarly, the EMPLOYEE superclass has a specialization with three disjoint subclasses: SECRETARY, TECHNICIAN, and ENGINEER. This reflects the fact that secretary, technician, and engineer are mutually exclusive job descriptions.

Finally, *completeness* must be specified for the model. This is an extension of the participation constraints imposed by the ER model. The completeness constraint in the EER model determines superclass entity participation in the superclass/subclass relationship. Such a constraint may be total or partial. A total specialization constraint mandates that every superclass entity be a member of some subclass, whereas partial specialization allows an entity not to belong to any subclass. Figure 1 illustrates these concepts. There is a total participation constraint between EMPLOYEE and the specialization {SALARIED EMPLOYEE, HOURLY EMPLOYEE}, indicated graphically by a double-lined arc to the connecting node (the same graphic notation as ER existence dependency). There is also partial participation between EMPLOYEE and {SECRETARY, TECHNICIAN, ENGINEER}. Do these constraints accurately reflect the real world existence of database entities? Yes, because it is logical that an employee

may be a secretary, technician, or engineer, or possibly none of the three. It is also apparent that a paid employee must be either given a salary or compensated by the hour

The EER model is now complete. But what is its real advantage? One view is that the EER model enhances the database designer's ability to capture the real data requirements accurately because it requires one to focus on greater semantic detail in the data relationships. Also, abstraction techniques, such as generalization, provide useful tools for integration of user views to define a global conceptual schema (Teory, 1986). These tools are essential for complex designs such as CAD databases. Because of the iterative nature of the (CAD) design process, designers cannot give a complete description of the design at once; they provide a partial description, later completed by repeated refinements. The iterative and tentative nature of the design process implies several descriptions of the design object in the database at any time, and previous states of the design must be available to designers working on the later states (Berzins, 1987).

III. THE OBJECT-ORIENTED DATA MODEL

A. INTRODUCTION

The EER is a high-level or *conceptual* data model. As such, it models some aspect of the real world in a way which is similar to a person's perception of it. At the opposite extreme are *physical* data models, which describe how data is stored and arranged on some physical medium. An *implementation* data model is a combination of the two; while providing a reasonable representation for a higher-level schema, it also is not far removed from a low-level structural model. (Elmasri, 1989)

The relational model is an implementation model, since its logical tabular arrangement of data equates nicely to a physical record structure. Like the relational model, the object-oriented model provides structure and meaning to a database, but goes much further. The distinguishing feature of such a model is its ability to easily represent the *dynamic* nature of a database, that is, the operations which may be performed upon it.

In this chapter we examine the basic concepts of object-orientation. Then, using these ideas, an object-oriented schema will be designed for a real-world database. Finally, the EER and object-oriented data models are compared.

B. BASIC CONCEPTS

The fundamental concept supporting object-oriententation is that of *object*. An object represents a unique entity in the real world. It has its own identity independent of any characteristics it may possess via an *object identifier* (OID). In terms of database

storage, an object may be differentiated from any other object by means of an OID. Hence an object does not need a unique characteristic (i.e., key attribute) to uniquely identify it. the OID is a system-defined value.

An object may have both a *state* and a *behavior*. The state of an object is determined by the values of its *instance variables* (object properties). The behavior of an object is provided by its *methods*. Methods are simply blocks of code which manipulate or return the state of an object (Banerjee, 1987). They are implemented via *encapsulation* and *information hiding*. A method is encapsulated when another object may access it only through a common interface. This interface is called a *message*, which when sent by one object invokes execution of a method on another. A method may return the state of its object to another object by sending a return message. An object may also hide information from other objects by having *private* methods which are innaccessible to them. A *public* method may be accessed by any object.

Objects having similar properties and methods are grouped together in *classes*. This concept, called *classification*, is an advantage both from a modeling and physical storage viewpoint. For example, if a number of objects use the same method to return or change their state, it would be wasteful to encode a procedure for each object. Instead, when a method on an object is invoked, a common definition stored in a *class object* is used for all objects belonging to a particular class. Similarly, *class properties* and *class methods* may be defined which do not apply to any particular object, but to a group of objects.

For an object to exist, it must be created. *Instantiation* is the process by which a new object is created by sending a message to a class object. The class object contains one or more methods called *constructor methods* specifically for this purpose. In addition to constructor methods, other methods exist (Hughes, 1991) which return or change the state of an object. *Accessor methods* return the current state of an object, whereas

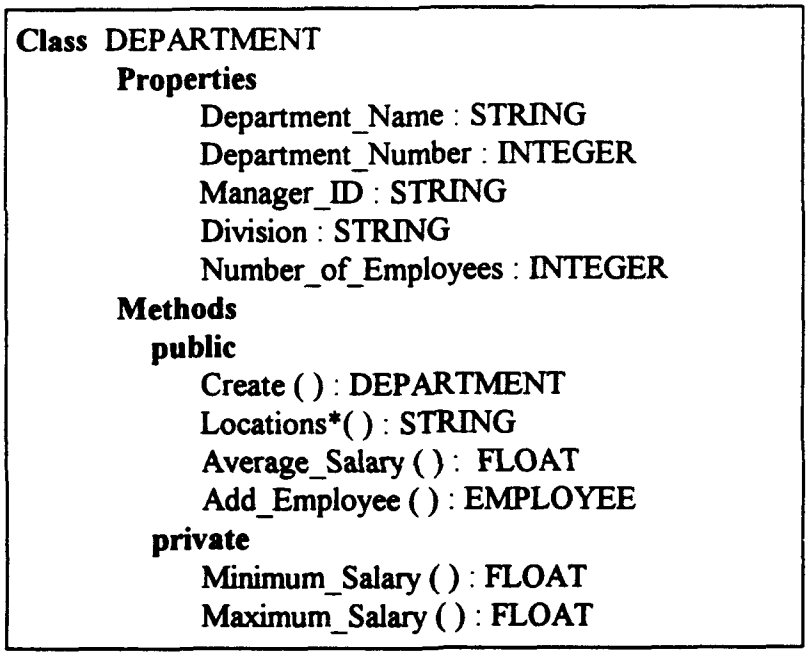


Figure 2. The Class DEPARTMENT

transformer methods change its state and return a new object (i.e., the same object with a different state). *Destructor methods* are similar in form to constructor methods, but opposite in function. That is, they remove object instances from existence.

To illustrate the above concepts, refer to the example object-oriented schema of Figure 2. This schema models an entity in the real world of business, a company

department. Since company departments can be described as having a similar representation for state and behavior, they are modeled as a class DEPARTMENT. An object belonging to this class has certain characteristics, such as a name, a manager, and a controlling division. It also has certain behaviors, or methods, which may be accessible or not. For example, the constructor method *Create* may be invoked by an appropriate user to create a new DEPARTMENT object. The accessor method *Location* returns a set of one or more objects of type STRING (for example, different city names) indicating where the company department exists. Another accessor method, *Average_Salary*, computes the mean wage paid to all employees in the department. *Add_Employee* is a transformer method which changes the state of the department by creating an object of class EMPLOYEE and thereby modifying the instance variable *Number_of_Employees*. Finally, two private methods are used by the class object to update the minimum and maximum salary each time a new employee is added to the department.

Another distinguishing feature of the object-oriented model is that of *inheritance*. It is the most powerful concept of object-oriented programming (Bertino, 1993). Like the concept of classification, inheritance is an effective storage and modeling technique. The basic idea of inheritance is the superclass/subclass relationship, discussed in Chapter II. The subclass assumes the properties and methods of its superclass (or superclasses) and may define specific ones for itself. Suppose, for example, that the class DEPARTMENT of Figure 2 could be logically separated into more than one type of object. Even though these objects are similar in that they are all departments, they are functionally different in

certain distinguishing ways. A department might specialize in production, sales, or marketing. A production department may have a specific requirement to maintain information concerning total time spent in producing a product or service. Accounting data maintained by the sales department may consist of total and average sales during a

```
Class PRODUCTION_DEPARTMENT
        Superclass
          DEPARTMENT
        Methods
          public
              Weekly_Product_Hours ( ) : FLOAT
              Weekly_Service_Hours ( ) : FLOAT
Class SALES_DEPARTMENT
        Superclass
          DEPARTMENT
        Methods
          public
              Total_Weekly_Sales ( ) : FLOAT
              Average_Sale_Price ( ) : FLOAT
Class MARKETING_DEPARTMENT
        Superclass
          DEPARTMENT
        Methods
          public
              Average_Customer_Age ( ) : FLOAT
              Customer_Males ( ) : INTEGER
              Customer_Females ( ) : INTEGER
              Married_Customers ( ) : INTEGER
```

Figure 3. Specialization of DEPARTMENT

specific time interval. Marketing, on the other hand, may need to know survey information such as average age, sex, and marital status of customers. Figure 3 illustrates how the schema of Figure 2 may be expanded to model these refinements. This schema format,

derived from Hughes (1991) and Bertino (1992), defines a subclass by specifying its superclass within the schema description. For example, SALES_DEPARTMENT is a subclass of the superclass DEPARTMENT, inheriting the properties and methods of DEPARTMENT while specifying two additional methods, *Total_Weekly_Sales* and *Average_Sale_Price*, which are unique to SALES_DEPARTMENT.

In addition to the basic concepts of the object-oriented model, there are other more advanced modeling techniques which greatly enhance the expressiveness and utility of the model. The ORION data model (Banerjee, 1987) proposes three major extensions : *composite objects*, *schema evolution*, and *versions*. A composite object is an object which has properties whose domain consists of objects of another class; such domain objects are called *component objects*. Component objects may be composite objects themselves, leading to a *composite object hierarchy*. With such a hierarchy, it is possible to model extremely complex real-world objects in a natural fashion. An excellent discussion of issues involving composite objects may be found in Kim (1989).

Schema evolution allows for a database schema to be dynamically defined and modified while maintaining integrity and consistency of the database. ORION achieves this by satisfying a set of *schema invariants*, which are object-oriented semantic integrity constraints. For example, any change to a database class structure must result in a directed acyclic graph, or tree structure. This is the *class lattice invariant*. The *full inheritance invariant* ensures that a class must inherit the instance variables and methods of its superclasses. In addition to invariants, which maintain the integrity of a database, rules for

schema evolution exist which ensure that consistency is maintained. Among these are *default conflict resolution* and *property propagation* rules, which establish how property name clashes are resolved in multiple inheritance schemas and how properties are propagated throughout the class hierarchy when a schema instance variable is modified.

Versions are alternate copies of the same object which are extremely important in design databases. Through versions a history is maintained of how a specific version of an object was created. Such version histories have significant implications in managing a complex design project. Although beyond the scope of this thesis, Bertino (1993) provides a good discussion of version modeling.

C. AN OBJECT-ORIENTED SCHEMA

A real-world database schema can be modeled with object-oriented concepts. Suppose we have a database containing information describing medical services provided by a health care facility (such as a hospital). A hospital may be divided into functional units, or wards. A ward may specialize in a certain type of patient care, for example, surgical, obstetric, or pediatric. Each ward has certain characteristics in common : a name, number of employees assigned, and number of patients assigned. In addition, surgical wards may keep track of the number of operations per day and postoperative infection rate. Obstetric wards may need information on babies born per day, neonatal unit staffing, and number of cesarean sections performed per day. A pediatric ward could keep a record of average patient age and total incidence of bone fractures. With these requirements in

mind, it is natural to model a ward as a superclass with subclasses for each of its three specializations.

Each ward consists of a collection of patients. The data maintained for these people might be address, social security number, name, phone number, blood type, sex, age, and type of illness. Thus a patient is an object in its own right, and is modeled as a class.

One or more illness types are assigned to each patient. All illnesses are identified by a name and primary treatment (such as drug therapy or surgery). Illnesses may either be terminal or non-terminal. For terminal illnesses, cumulative patient deaths are recorded. For non-terminal ones, average time for patients to recover is maintained. With this in mind, an illness would be modeled as a superclass for curable and incurable diseases.

Finally, medical care is provided by trained hospital personnel. The medical staff consists of doctors, nurses, and medical aides. For all medical personnel, the following data is kept : home phone, name, sex, address, social security number, birthdate, educational degree, workshift, and age. In addition, it is known what patients are assigned to each staff member, who they supervise, what other staff members (if any) supervise them, and what ward they are assigned to. Care providers who are physicians are also identified by their annual malpractice premium and name of medical specialty (i.e., cardiologist, radiologist). Nurses have the additional property of licensing source (i.e., RN or LPN). Medical aides are further described by the number of cumulative training hours they have received in CPR, physical therapy, and patient hygiene.

In producing an object-oriented schema for the hospital database, we use a format similar to that of Figures 2 and 3. To make the model more concise, the distinction between public and private methods is not shown, and constructor/destructor methods are not given for each class :

Class MEDICAL_STAFF
  Properties
    Name : STRING
    Address : STRING
    Home_Phone : STRING
    Sex : CHARACTER
    SSN : STRING
    Birthdate : DATE
    Workshift : INTEGER
    Degree : CHARACTER
    Ward_Assigned : WARD
    Supervisor : MEDICAL_STAFF
    Supervisees* : MEDICAL_STAFF
  Methods
    Age ( ) : INTEGER
    Patients_Assigned* ( ) : PATIENT

Class PHYSICIAN
Superclass MEDICAL_STAFF
  Properties
    Malpractice_Premium : FLOAT
    Specialty : STRING

Class NURSE
Superclass MEDICAL_STAFF
  Properties
    Licensing : STRING

Class MEDICAL_AIDE
Superclass MEDICAL_STAFF
  Properties
    CPR_Hours : FLOAT
    Physical_Therapy_Hours : FLOAT

Patient_Hygiene_Hours : FLOAT

Class WARD
Properties
Medical_Staff_Assigned* : MEDICAL_STAFF
Methods
Number_of_Employees ( ) : INTEGER
Number_of_Patients ( ) : INTEGER
Patients_Assigned* ( ) : PATIENT

Class SURGICAL_WARD
Superclass WARD
Methods
Operations_Per_Day ( ) : INTEGER
Post_Op_Infection_Rate ( ) : FLOAT

Class OBSTETRIC_WARD
Superclass WARD
Properties
Neonatal_Unit_Capacity : INTEGER
Methods
Births_Per_Day ( ) : INTEGER
Cesareans_Per_Day ( ) : INTEGER

Class PEDIATRIC_WARD
Superclass WARD
Methods
Total_Bone_Fractures ( ) : INTEGER
Average_Patient_Age ( ) : FLOAT

Class PATIENT
Properties
Name : STRING
Address : STRING
Room_Number : STRING
Home_Phone : STRING
Sex : CHARACTER
SSN : STRING
Birthdate : DATE
Blood_Type : STRING

Ward_Assigned : WARD
Medical_Staff_Assigned* : MEDICAL_STAFF
Illnesses* : ILLNESS

Methods
Age ( ) : INTEGER

Class ILLNESS
Properties
Name : STRING
Primary_Treatment : STRING
Methods
Affected_Patients* ( ) : PATIENT

Class TERMINAL_ILLNESS
Superclass ILLNESS
Methods
Cumulative_Patient_Deaths ( ) : INTEGER

Class NON-TERMINAL_ILLNESS
Superclass ILLNESS
Methods
Average_Recovery_Time ( ) : FLOAT

In the hospital schema, some characteristics are modeled as methods instead of properties. This would depend on the implementation; in general, information which is subject to frequent change may be modeled by methods for efficiency of implementation. For example, the *Age* method is a derived attribute of classes MEDICAL_STAFF and PATIENT. The ages of patients and employees may be directly calculated from their birthdates, which do not change (and hence are properties). Similarly, the *Patients_Assigned* method, which returns a set of objects of class PATIENT, is better modeled as a method, since it is a dynamic characteristic. Also note that *Cumulative_Patient_Deaths*, the method for subclass TERMINAL_ILLNESS, and

Average_Recovery_Time, the method for subclass NON-TERMINAL_ILLNESS, are both class methods, since they tabulate data for groups of ILLNESS objects. On the other hand, *Total_Bone_Fractures* is an object method for PEDIATRIC_WARD, since it only calculates data for a specific object (the pediatric ward).

D. CONCLUSION

The object-oriented model and the EER model are similar in many respects. Both model object-oriented concepts such as inheritance, superclass/subclass relationships, generalization, and specialization. In the EER model, an entity represents a unique member of a superclass or subclass. The analogous construct in the object-oriented model is the object. EER entity types have attributes; the object-oriented equivalent is the property (i.e., instance variable).

Nevertheless, there are noticeable differences. First of all, the EER is a conceptual model. It represents the real world in a way which is very natural to understand, graphically showing relationships and constraints among entity types. Conversely, the object-oriented model is an implementation model. As such, it gives more detail as to how objects are represented in a database. That is, data types of class properties are displayed in a schema. Uniqueness of an entity is represented by a key attribute in the EER model; the OID performs this function for object-oriented model. In addition, there is no EER equivalent for the composite object. The greatest difference between the two models, however, is how database dynamics are illustrated. Whereas the object-oriented model has methods, the EER has no such capability. Hence, the EER provides us with an excellent

conceptual vehicle for describing a database, while the object-oriented model allows us to model both state and behavior. In the next chapter, we will merge these two capabilities.

IV. A CONCEPTUAL OBJECT-ORIENTED MODEL

A. INTRODUCTION

In the last chapter the object-oriented data model was introduced and compared to the EER model. In this chapter, both models are combined into a high-level graphical model called OPERA (Object Paradigm/Entity-Relationship Approach). This model is capable of expressing database constraints, operations, and logical data relationships between object classes. OPERA proposes to bridge the gap between the entity-relationship and object-oriented data models by integrating the mathematical relation and the method in a visual representation of a database. Once formulated, OPERA will be used in the remainder of the thesis to aid in object-oriented query formulation (Chapter V) and to model and query a complex military database (Chapter VI).

B. THE GRAPHICAL OBJECT-ORIENTED SCHEMA EXPRESSION

Before merging the object-oriented and EER models, it is helpful to express both of them graphically. Since the EER is already a conceptual schema, only the object-oriented implementation model needs to be converted. This modification, called the Graphical Object-Oriented Schema Expression (GOOSE), is a visual object-oriented schema adapted from Bertino (1993). It makes the transformation to an OPERA diagram easier by pictorially displaying links between classes and, unlike the Bertino model, incorporates EER constraints on superclass/subclass relationships.

Figure 4 shows the basic schema template. A class (or subclass) is modeled as a rectangular block. This is similar to the EER representation. Within the block, the properties of the class (i.e., attributes of the entity type) are modeled as in the object-oriented implementation schema. Each property is linked to a primitive domain , such as INTEGER or STRING, by a colon (:), or to a complex domain, such as a

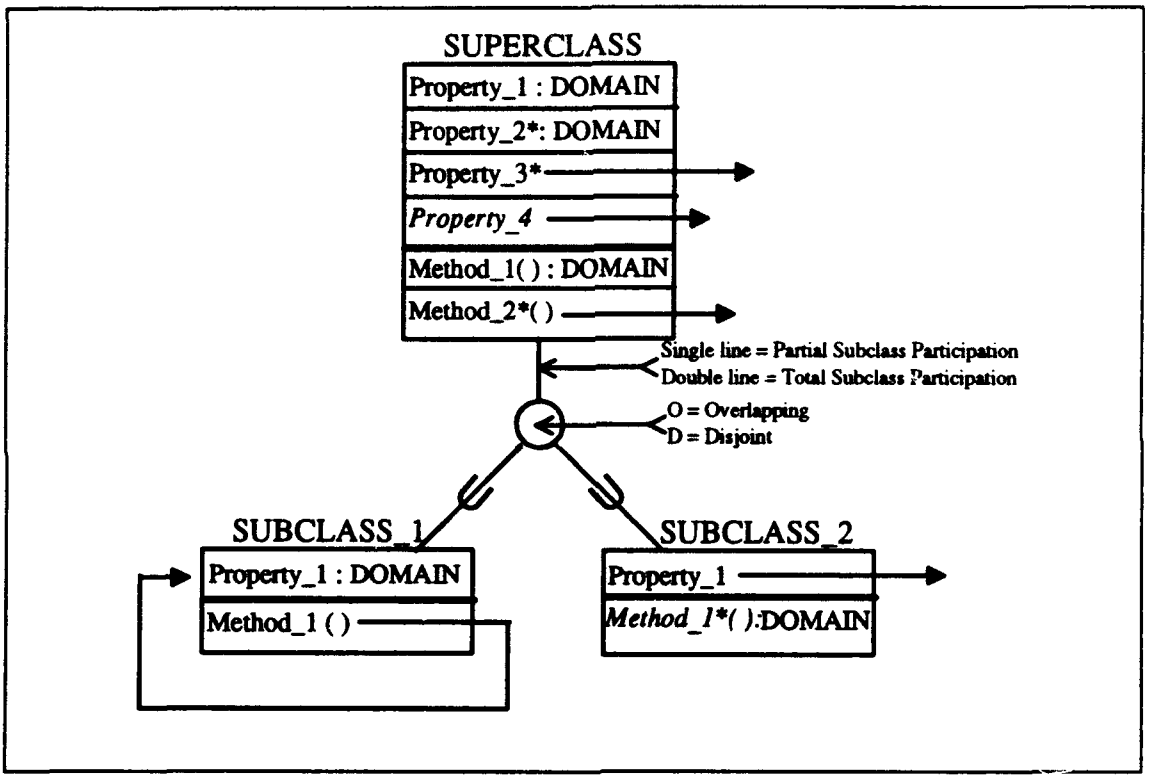


Figure 4. GOOSE Template

composite object class, by an arrow (→). Properties may be single-valued or multivalued, that is, they may return a single object or a set of objects. Single-valued and multivalued property domains may be primitive or complex; multivalued properties are denoted by an asterisk (*). In addition, class properties are italicized. Methods are represented in a

similar way , but are separated from properties by a bold line. The parenthesis following a method name represents its functional nature. That is, method activation is modeled as a procedure call. As SUBCLASS_1 illustrates, a domain link may be recursive in nature. In other words, the domain of a method (or property) may consist of objects belonging to that method's (or property's) class. Finally, constraints and the IS_A relationship are modeled as in the EER.

We are now ready to model an object-oriented implementation schema in a GOOSE diagram. The hospital database schema of Chapter III will be used for this purpose. Recall that in this schema, there are four superclasses (MEDICAL_STAFF, WARD, PATIENT, ILLNESS) and eight subclasses (PHYSICIAN, NURSE, MEDICAL_AIDE, SURGICAL _WARD, OBSTETRIC_WARD, PEDIATRIC_WARD, TERMINAL_ILLNESS, NON-TERMINAL_ILLNESS). The methods and properties can be mapped directly from the object-oriented schema, whereas the constraints may be derived from the verbal description of the database. The complete GOOSE schema is illustrated in Figure 5.

C. OPERA

The OPERA model is a graphical representation of the state and behavior of an object-oriented database. Since it combines the entity-relationship and object-oriented data models, the EER and GOOSE diagrams will be integrated into a common description. To facilitate this, the behavioral aspect of the object-oriented approach, the method, must be presented conceptually.

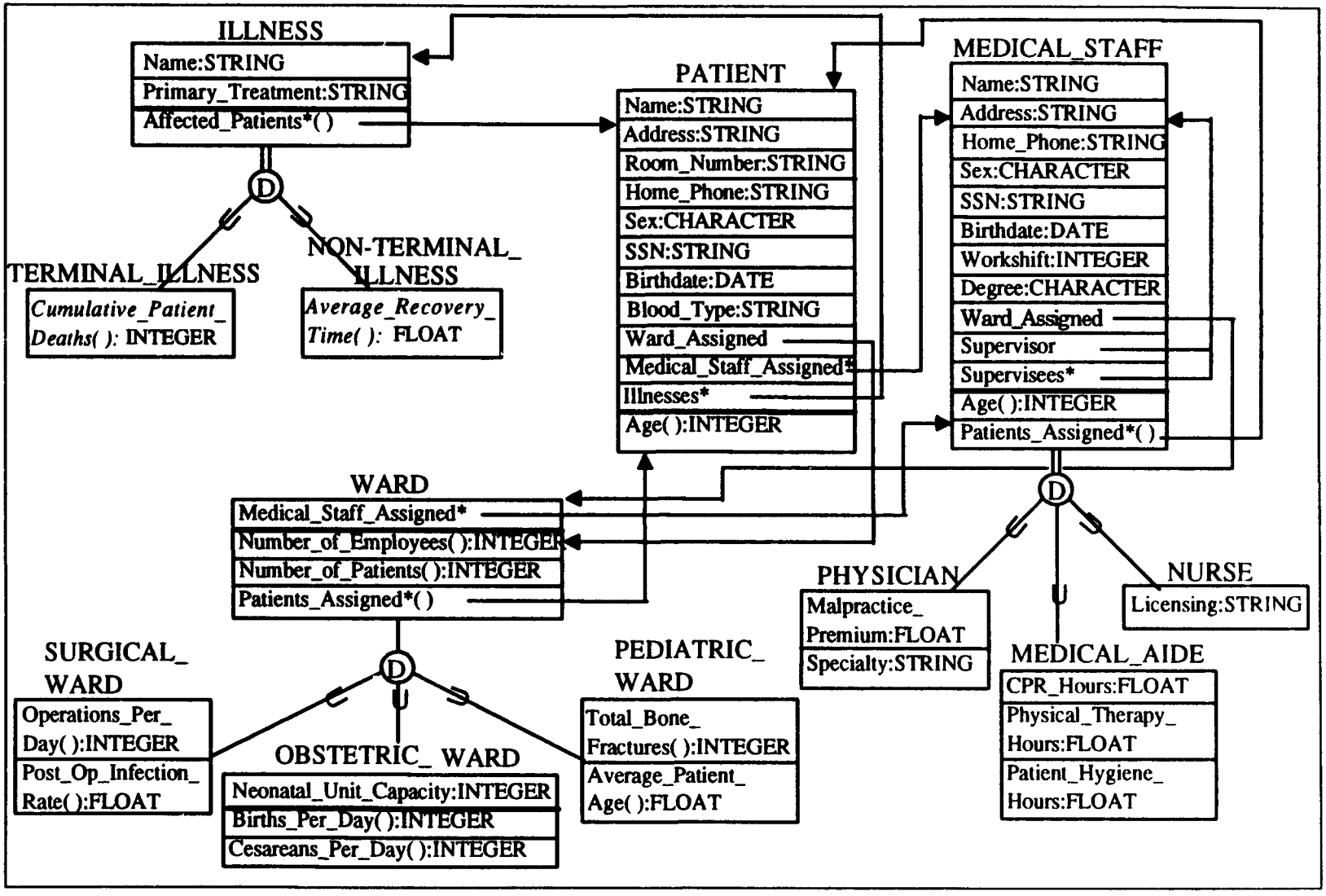


Figure 5. Hospital Database GOOSE Diagram

Recall from the discussion of Chapter III that a method models a functional capability. That is, when an object sends a message to another object, a method (identified by the message syntax) may be executed by the object receiving the message. In return, the receiving object may send a return message, which may contain state information or invoke a method on the sending object. In mathematical terms, this is a logical mapping from one set to another. Suppose we have two sets A and B as shown in Figure 6. Set A

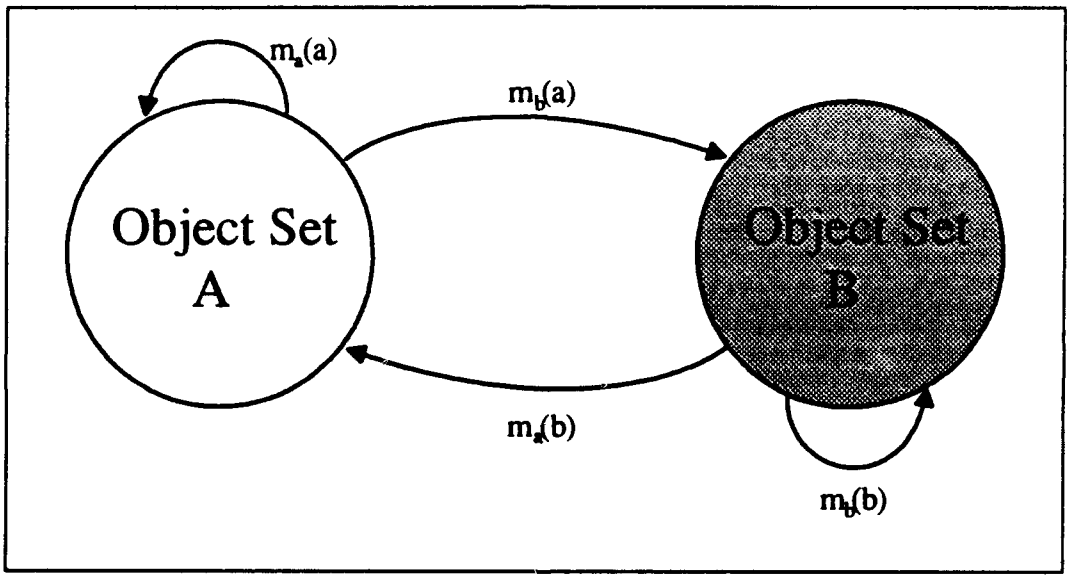


Figure 6. Set Mapping with Methods

consists of n elements, and set B consists of m elements. That is, $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$. A method may be defined as a mapping from one set (i.e., class) to another. In the case where a method returns the state of one element (i.e., object) to another, we may model this transformation as $m_r(s)$, where $r \in R$ and $s \in S$, with R the set of receiving elements and S the set of sending elements. This situation is depicted in Figure 6 by $m_b(a)$, which represents a method executed by an object of A (in response to a

message from an object of B) with a return of some state information from the A object to the B object. Similarly, B may map into A. Note also that a method may map one set into itself. We may model this as $m_s(S)$, where $s \in S$, and S is any set (i.e., class) which has some method (m_s) which returns elements of its containing set. In Figure 6, we see this type of method execution with $m_b(b)$, which indicates a change of state for set B or some other information returned only to set B.

We have indicated the mathematical equivalence of the *set* and *class*. Likewise, the terms *element* and *object* are considered the same. A mathematical set consists of unordered, discrete elements. This corresponds to the class, which consists of distinct objects (each with a unique OID) which are logically unordered (although physically ordered in a database). Using this premise, we will now show the mathematical equivalence of the method and relation.

Suppose there are two sets X and Y. The set X is said to be a *subset* of Y if and only if every element of X is also an element of Y. We may express this as:

$$X \subseteq Y \text{ iff } \forall x(x \in X \rightarrow x \in Y).$$

Now suppose there are two sets Y and Z. The *cartesian product* of Y and Z, denoted $Y \times Z$, is the set of all ordered pairs (y,z) where $y \in Y$ and $z \in Z$. This may be expressed as:

$$Y \times Z = \{ (y,z) \mid y \in Y \wedge z \in Z \}.$$

Given that a set is equivalent logically to an object class, and a set element is equivalent to an object, we may substitute the classes A and B of Figure 6 for the sets Y and Z above. We may think of a method as an ordered pair of objects, since a method $m_a(b)$ obtains

state information from object *a* and sends it to object *b*, forming an ordered pair (a,b). Since (a,b) represents a subset of A×B, the method may be though of as:

$$m_a(b) \subseteq A \times B.$$

However, given two sets A and B, a *binary relation* from A to B is a subset of A×B. Therefore, the method is mathematically equivalent to the relation.

We are now in a position to develop a high-level, conceptual representation for an object-oriented implementation schema. Starting with a GOOSE diagram and a miniworld, that is, a verbal description of the database and its constraints, we may proceed to an OPERA diagram as follows:

1. For each GOOSE superclass, construct an OPERA superclass by mapping the superclass name into an EER entity set.
2. For each GOOSE subclass, construct an OPERA subclass by mapping the subclass name into an EER entity set.
3. For each GOOSE superclass/subclass relationship, map constraints directly into the OPERA diagram, thus forming a link between OPERA superclass entity sets and subclass entity sets.
4. For each domain link in a GOOSE diagram, form an EER relationship type in the OPERA diagram. Label the relationship type with the names of the classes at the origin and destination of the link. If the domain link is recursive (same origin and destination class), label the relationship type with a role name.
5. For GOOSE methods and properties with primitive domains, no relationship type is mapped to the OPERA diagram. Such methods and properties are not explicitly represented in OPERA.
6. For GOOSE methods and properties with single-valued domain links, initially label the corresponding OPERA relationship type cardinality as 1:1. If the inverse of this method or property exists on another class with a multivalued domain link, or if the miniworld description explicitly states a many-to-one cardinality, relabel the OPERA relationship type cardinality as N:1.

7. For GOOSE methods and properties with multivalued domain links, initially label the corresponding OPERA relationship type cardinality as 1:N. If the inverse of this method or property exists on another class with a multivalued domain link, or if the miniworld description explicitly states a many-to-many cardinality, relabel the OPERA relationship type cardinality as M:N.
8. After determining the relationship cardinality in steps 6-7, assign participation constraints between OPERA class entity sets based on the miniworld description. Partial and existence dependency are shown as in the EER model.

We are now in a position to generate an OPERA diagram for the hospital database schema, whose miniworld was described in Chapter III. From this description and the GOOSE diagram of Figure 5, the OPERA schema is derived following the mapping scheme in steps 1-8. The conceptual view of the hospital database is shown in Figure 7.

D. CONCLUSION

The GOOSE diagram is an intermediate step between the object-oriented implementation schema of Chapter III and the high-level OPERA schema derived from it. GOOSE serves primarily as a conceptual aid in visualizing an implementation schema. However, it also incorporates object-oriented characteristics of the EER model, which eases the task of mapping to OPERA. Although OPERA does not offer an improvement over the GOOSE diagram in terms of gross database description, it simplifies a schema by abstracting out implementaion details (such as domain representations) and transforming domain links into relationship types.

Although the OPERA and EER schemas look alike, they are significantly different. For example, an EER representation shows all attributes of each class, including key attributes, which uniquely identify an entity. In OPERA, attributes (i.e., properties and

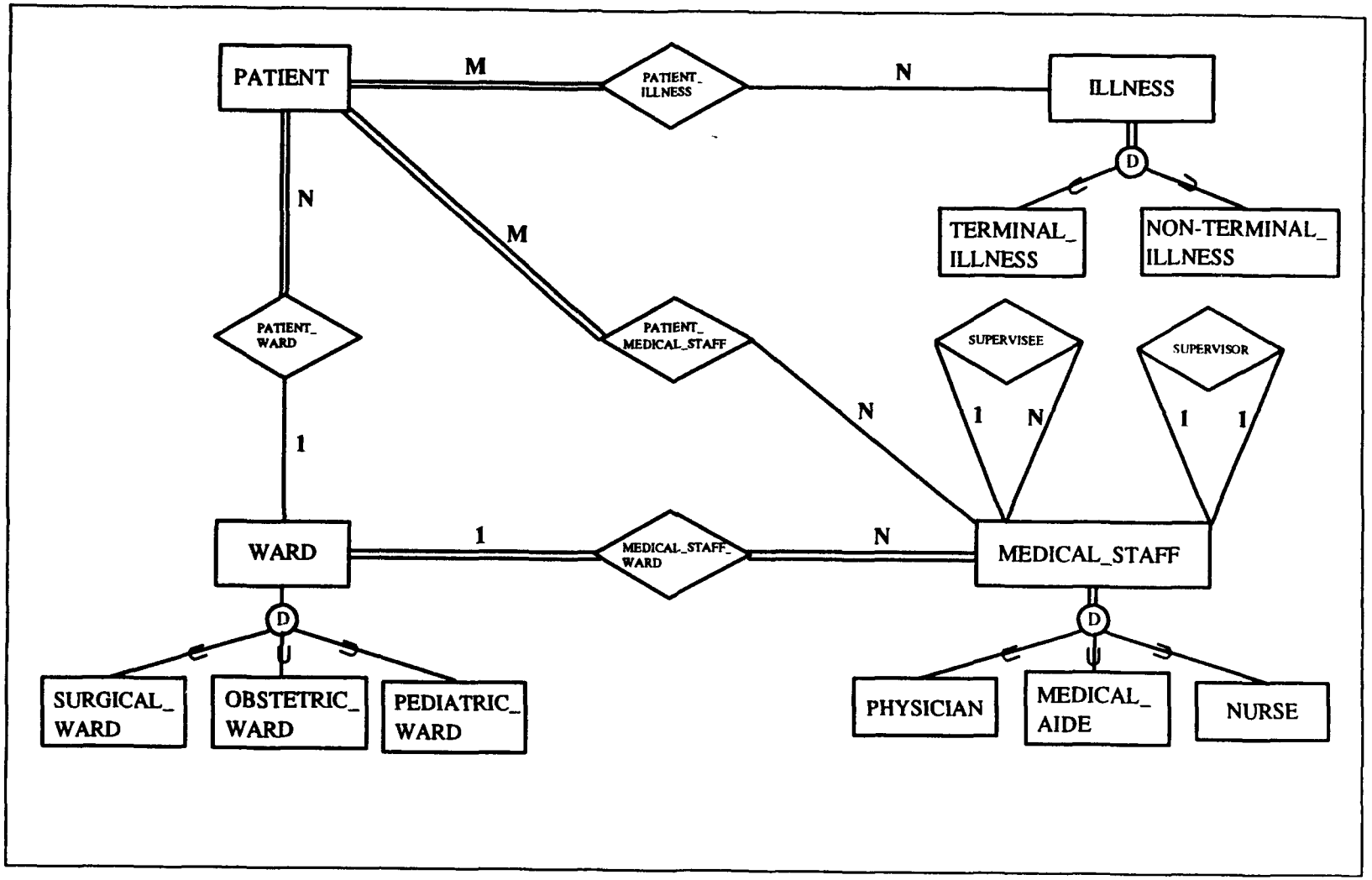


Figure 7. Hospital Database OPERA Diagram

methods) are not explicitly shown. In fact, there are no key attributes, since each object is uniquely identified by an implicit OID. In this way OPERA abstracts a database to the highest conceptual level, without regard to lower level details of implementation.

Another significant difference between the two models is in the treatment of methods. In the EER, methods are not represented. In OPERA, a method which has a complex domain is modeled as a relationship type-owing to its theoretical equivalence to the mathematical relation. Methods which have primitive domains are treated like primitive properties; they are both considered to be details of implementation, and therefore not explicitly modeled in OPERA. As we will see in the next two chapters, this higher level of abstraction is beneficial when choosing an appropriate query language for the object-oriented model and in describing relational databases in object form.

V. OBJECT-ORIENTED QUERIES

A. INTRODUCTION

A very important part of any data model is the set of operations which may be performed on it. These operations may exist to construct a database schema (e.g., CREATE) or modify it (e.g., DELETE). In general, these are system-defined and not easily modified; in fact, there is usually no need to. Another type of operation which is required by a database user is the query. Queries exist primarily for the purpose of extracting information from a database. An effective query language has several characteristics, among which are conciseness, semantic integrity, and efficiency of data retrieval. In this chapter, queries are examined from a semantic viewpoint; this is deemed important for object-oriented data models, which rely on ease of understanding to model real-world entities effectively.

Several different approaches to object-oriented queries will be examined. First, a mathematical model upon which many object query languages are based, called object-oriented predicate calculus (OOPC), will be reviewed. Next, two generic object query model languages, GOMql and GOMsql, are studied. Then, a query language developed for the ORION prototype object-oriented database system is examined. Finally, a query model based on the entity-relationship data model, GORDAS, is expanded to incorporate object queries. A sample object schema is queried using the different query languages and the semantic efficiency of these approaches is compared.

B. OBJECT-ORIENTED PREDICATE CALCULUS (OOPC)

One of the criticisms of object-oriented data models is the lack of a standard mathematical foundation. The relational data model, on the other hand, has no such problem. The mathematical concept of relation provides its basis and, through relational calculus, supports the associated query languages (such as SQL). OOPC (Bertino, 1992) attempts to bridge this mathematical gap by extending the relational predicate calculus to object-oriented queries..

An OOPC query has the following syntax : {Target clause; Range clause; Qualification clause}. The target clause specifies the data to be retrieved by the query. It may be a variable representing an object in the schema or a property of an object. The form it takes is either {x} or $\{x.A_i\}$, where x is a variable bound explicitly to a class in the range clause, and A_i is a property of the class bound to x. The range clause indicates an explicit binding of a variable to a schema class. This scope of this binding is both the target clause and qualification clause. For example, the range clause {x/C} means that any variable x in a query formulation represents an object of class C. The qualification clause contains the selection criteria, i.e., predicates, which specify which object is chosen for retrieval; the data from this selected object (or objects) is returned in the form of the target clause. The format for the qualification clause is { Θ Range clause (Qualification clause)}, where Θ is either the universal ($\forall$) or existential ($\exists$) quantifier. (Qualification clause) is a Boolean combination of predicates connected by the logical and ($\wedge$), or ($\vee$) or not ($\neg$).

An example query is the best way to illustrate the use of OOPC. Referring to the schema of Figure 8, a user of the database might need to know the salary of all presidents of companies which have a manufacturing division. The query could be formulated as:

Query 1: *{x.salary; x/Employee; ∃ y/Company (x.name = y.president.name ∧ y.divisions.name = "manufacturing")}*

Note the use of the dot operator (.) in the query. A path expression such as {y.divisions.name} functions as an implicit join; that is, the classes **Company** and **Division** are examined without need of an explicit equality comparison between a common property (as is the case in a relational query). Also, this path expression returns a set of objects in the course of navigation - {y.divisions} returns a set of objects which are divisions of a single company. Each of these, in turn, returns its name, which is then compared for equality with "manufacturing". To make this query more efficient (and more meaningful), OOPC allows for a quantified path expression. This is helpful when returning sets of objects in the course of query navigation; in the case of existential predicates, it allows the query to resolve without returning all possible set objects. For example, the above query may be changed to :

Query 1A: *{x.salary; x/Employee; ∃ y/Company (x.name = y.president.name ∧ y.∃ divisions.name = "manufacturing")}*

The semantics are clearer here; for every company, its divisions are searched until one with the name "manufacturing" is found. Once found, the predicate evaluates to true, and the query may be resolved.

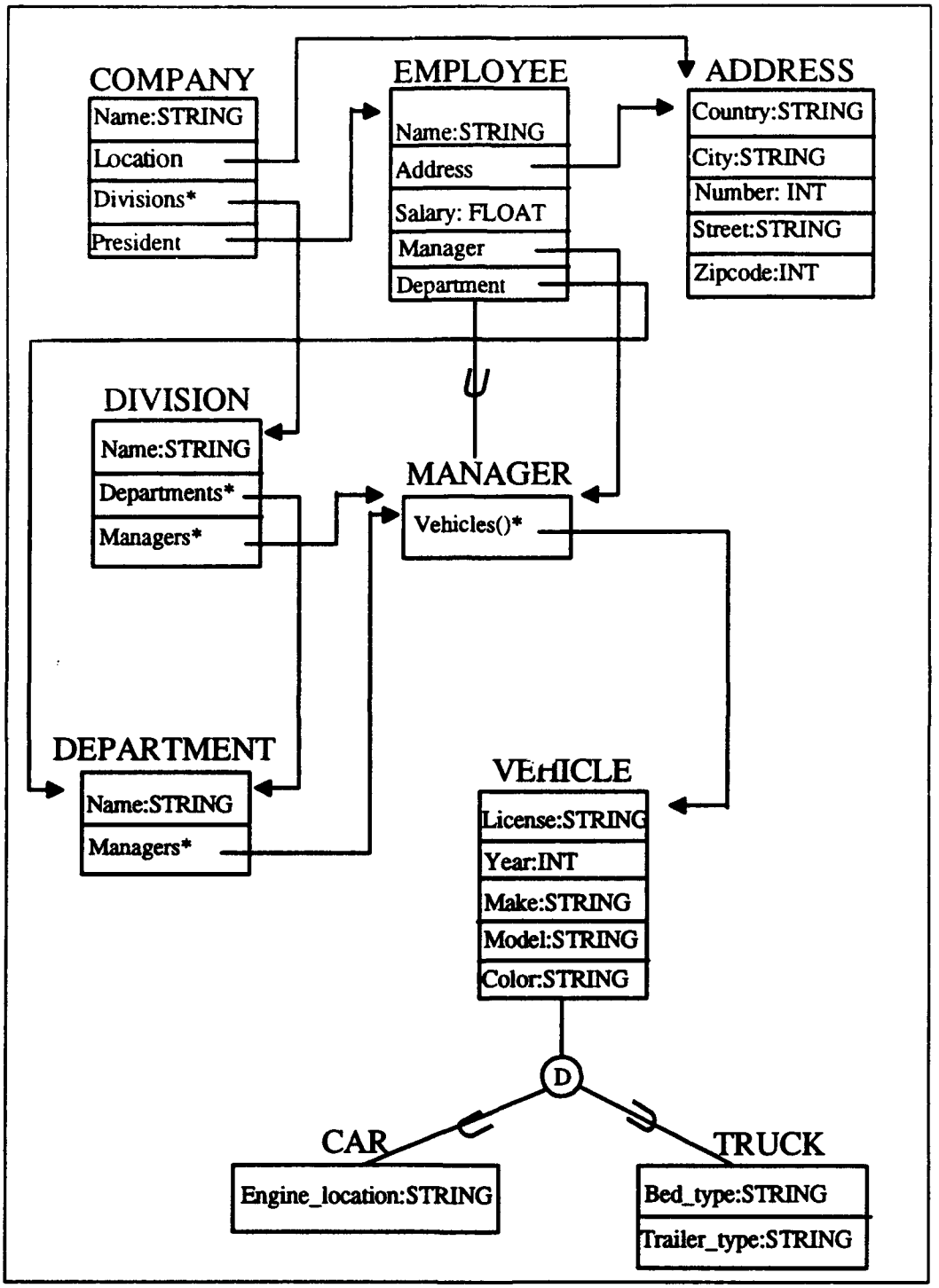


Figure 8. Company Database Schema

Methods, like properties, must be incorporated into an object-oriented query language. In OOPC, they both have the same form in a query path expression. For example, a requirement may exist to determine the name and address of all department managers who earn more than $50,000 per year and drive a company car with a year model prior to 1985. A possible query might be:

Query 2: *{x.name,x.address; x/Manager; ∃ y/Department (x ∈ y.managers ∧ x.salary > 50000 ∧ x.∃ vehicles( ).year < 1985)}*

This query illustrates the problem of how data is retrieved from a subclass specified in a schema. Manager is a subclass of Employee; x could be bound to this superclass in the range clause, but the query semantics would suffer (in addition to retrieving the wrong result - all employee names and addresses vice managers). However, x can be bound to Manager and still project the name and address properties in the result, since these properties are inherited from the Employee class. Another issue raised by this query is how to restrict the set of managers to those of departments. This is done in OOPC via the membership predicate {x ∈ y.managers}. Finally, this example shows how methods are handled in a query path expression - just like properties. This should be expected, since a method (vehicles( ) in this case) returns an object (or set of objects), and a property domain consists of an object (or set of objects). Hence, from the viewpoint of query semantics, methods and properties are identical.

A feature characteristic of object-oriented data models is specialization of superclasses via subclasses. In Figure 8 the class **Vehicle** is specialized in the two

subclasses **Car** and **Truck**, which have some common properties, and some which are unique. Suppose a user needs to query the database to retrieve the license numbers of all red mid-engine vehicles and blue long-bed vehicles. A possible query might be :

Query 3: {x.license; x/Vehicle; (x.color = "red" ∧ x.engine_location = "mid") ∨ (x.color = "blue" ∧ x. bed_type = "long")

Given the schema of figure, will this query retrieve the correct data? Although clear in semantics, it is incorrect in execution. The problem arises from the fact that x is bound to Vehicle in the range clause; when a Car object is examined during query execution, an error occurs when the property bed_type is examined, because a Car object does not have such an attribute. The same problem occurs with Truck objects and the engine_location property. OOPC resolves this dilemma with the CLASS_OF operator:

Query 3A: {x.license; x/Vehicle; CLASS_OF(x) =
[Car : x.color = "red" ∧ x.engine_location = "mid";
Truck : x.color = "blue" ∧ x. bed_type = "long"]}

This construct is called an alternative predicate; the CLASS_OF operator applied to a predicate binds the predicate variable to the class designated at the head of the predicate clause. In the above example, x is bound to Car in the first predicate clause; in the second clause, x is bound to Truck. Hence, the CLASS_OF operator allows a type of polymorphism for binding variables within the qualification clause of the query. Although the semantics are obscured somewhat, the correct result is obtained.

C. GENERIC OBJECT MODEL (GOM) QUERIES

1. GOMql

GOMql (Kemper, 1994) is a high-level, stand-alone declarative query language based on the relational query language QUEL. It is very similar in form to OOPC, but its semantics are a bit clearer. The format of a GOMql query is :

range $v_1 : C_1 , \ldots , v_n : C_n$
retrieve $v_1 , \ldots , v_n$
where $P(v_1 , \ldots , v_n)$

In the range clause, the variables $v_1 , \ldots , v_n$ represent bindings to classes $C_1 , \ldots , C_n$. In the retrieve clause, $v_1 , \ldots , v_n$ are objects bound to classes in the range clause; these objects may be class properties or method results. The where clause contains the selection predicate(s) P which are evaluated to produce the query results specified in the retrieve clause. Path expressions of the form $v_1 .name_1 .name_2 ...name_n$, where v_1 is a variable bound in the range clause and $name_1 ...name_n$ are class properties or methods, may be elements of the retrieve and where clauses. The range, retrieve, and where clauses of GOMql query are analogous to the range, target, and qualification clauses of an OOPC query. For example, Query 1 (retrieve the salary of all presidents of companies which have a manufacturing division) is formulated in GOM as :

Query 4 :
range *e: Employee, c: Company, d: Divisions*
retrieve *e.salary*
where *e.name = c.president.name*
and exists d in c.divisions (d.name = "manufacturing)

Note the use of the **exists** and **in** operators; these perform the same function as the existential quantifier in OOPC. Like the existential quantifier, they must have as arguments variables which represent a set of objects. Here, d is a variable representing a set value; if one element of the set {c.divisions} is found which satisfies the equality predicate, it is selected. A similar operator, **forall**, exists in GOMql to function as a universal quantifier in queries.

Query 2 (determine the name and address of all department managers who earn more than $50,000 per year and drive a company car with a year model prior to 1985) may be written in GOMql in a similar fashion:

Query 5 : ***range*** *m: Manager, d: Department, v: Vehicles*
retrieve *m.name*
where *m* ***in*** *d.managers* ***and*** *m.salary > 50000*
and exists *v* ***in*** *m.vehicles (v.year < 1985)*

Note the use of the *implicit* existential quantifier **in**. This limits the managers bound to variable m to those also bound to variable d {m **in** d.managers}.

Query 3 (retrieve the license numbers of all red mid-engine vehicles and blue long-bed vehicles) posed a problem for OOPC in that the query semantics were impacted by the hierarchical nature of the object schema. In GOMql, the use of the **exists** operator greatly improves the meaning of this query, and eliminates the need for an explicit CLASS_OF operator:

Query 6 : ***range*** *v:Vehicle, c: Car, t: Truck*
retrieve *v.license*
where ***exists*** *c in v (c.color = "red"* ***and*** *c. engine_location = "mid")* ***or***
exists *t* ***in*** *v (t.color = "blue"* ***and*** *t.bed_type = "long")*

2. GOMsql

GOMsql (Kemper, 1994) is patterned after the standard relational query language SQL and the object query language O_2SQL. It has three parts to its query format, like OOPC and GOMql, and its format is :

select v_1 ,..... ,v_n
from v_1 ***in*** {$Class_1$| *Nested query expression*$_1$}, ...v_n ***in*** {$Class_n$|*Nested query expression*$_n$}
where $P(v_1, \ldots, v_n)$

The **select** clause is the equivalent of the OOPC target clause; the variables $v_1 \ldots v_n$ represent class properties or method results, and may be a path expression. The **from** clause is the range clause of OOPC, and like it, establishes bindings between variables $v_1 \ldots v_n$ and $Class_1 \ldots Class_n$ via the **in** operator. However, a binding may also be established from a variable to a *nested query expression* (which returns a collection of objects). The **where** clause contains selection predicates (as in the OOPC qualification clause) and may also contain nested queries.

To demonstrate the use of a nested query binding in the **from** clause, suppose a query is needed to retrieve all managers from the Plymouth Division of Chrysler Motor Company who earn more than $50,000 (refer to the schema of Figure 8). Such a query might be written as :

```
Query 7 :   select   m
            from     m in
                        (select d.managers
                         from d in
                              (select c.divisions
                               from c in Company
                               where c.name = "Chrysler")
                         where d.name = "Plymouth" )
            where    m.salary > 50000
```

Nested query expressions are evaluated from the innermost to outermost as in relational SQL. Although such nesting is possible, and may be useful at times in the **from** clause, a shorter query can usually be formulated using path expressions. Query 7 could be changed to read:

```
Query 7A :  select   m
            from     m in Manager, c in Company, d in Division
            where    (c.name = "Chrysler") and (d in c.divisions )
                       and (d.name = "Plymouth") and (m.salary > 50000 )
```

This version of the query is more concise than Query 7, and is easier to understand.

Queries 1, 2, and 3, previously written for OOPC and GOMql, are now written as Queries 8,9, and 10 for GOMsql:

```
Query 8 :   select   e.salary
            from     e in Employee, c in Company, d in Division
            where    e.name = c.president.name  and
                     c.divisions in
                        (select d
                         from d in Division
                         where d.name = "Manufacturing")
```

```
Query 9 :    select   m.name, m.address
             from     m in Manager, d in Department, v in Vehicle
             where    m in
                        (select d.manager
                         from d in Department)
                      and  m.salary > 50000
                      and
                      v in
                        (select m.vehicles
                         from m in Manager
                         where m.vehicles.year < 1985)

Query 10 :   select   v.license
             from     v in Vehicle, c in Car, t in Truck
             where    v in
                        (select c
                         from c in Car
                         where c.color = "red" and c.engine_location=
                         "mid")
                      or
                      v in
                        (select t
                         from t in Truck
                         where t.color = "blue" and t.bed_type = "long")
```

These three examples all use nested queries in the outer **where** clause in order to select a set, or collection, of objects(which is equivalent to a class). It is from this temporary collection that objects bound in the **from** clause are examined for membership.

D. ORION QUERY LANGUAGE

Banerjee (1988) proposes a formal model of queries under the object-oriented data model for ORION. In this model, a query is implemented as a series of messages sent to objects in an *aggregation hierarchy*. An aggregation hierarchy exists in an object-oriented schema when the domain of some class property has as its domain another class, and this

class in turn has a property domain which is also a class; such a schema may be defined recursively in terms of *composite objects*. In Figure 8 several aggregation hierarchies are apparent. Objects of class **Company** are composite, having three properties with class domains : Location, Divisions, and President. Each of these properties has one or more associated aggregation hierarchies. For example, a hierarchy for Divisions is :

Divisions➔Departments➔Managers➔Vehicles( )➔License

This hierarchy is the basis for the ORION query language syntax, and is the logical eqivalent of a path expression in other query languages.

An ORION query is a series of messages which return a result in the form of a set of objects belonging to a single class. This restriction has ramifications on the allowable queries which may be formulated, as will be shown later. The format of the query, based on the Smalltalk object-oriented programming language, is :

(*Receiver Object* *Selector* *Iteration Variable* *Query Expression*)

Receiver Object is either a set object which refers to a collection of objects (i.e., a class), or a single object. *Iteration Variable* is a variable (preceded by a colon) which binds the *Receiver Object* instances to the same variable appearing in the *Query Expression*. *Selector* is a message sent to *Receiver Object* ; it returns a set of instances of *Receiver Object* based on the evaluation of *Query Expression*. This expression returns true or false based on the resolution of predicates, and consists of code blocks which may contain other queries.

To illustrate the use of the ORION query syntax, suppose the database represented by the schema of Figure 8 must be searched to determine all companies located in New York City which have a president who resides in the same city. The query may be written as :

```
Query 11 : (Company select  :C ( :C City = "NewYork"
                                 and  ( :C President Address City = "New York" )
                                  )
           )
```

The resolution of this query is explained as follows. The **select** message is sent to the Company object, which is bound to variable :C. In the query expression, to each object bound to :C, the "City" message is sent, and this returns the value of the City property of Company (which is an object). This object is sent the message "=" with the argument "New York" ; the result is a Boolean value (True/False) depending on the value of City. The message "and" is in turn sent to this Boolean object, with its argument the result of the next code block. If both code blocks in the query expression evaluate to true, the instance of Company bound to :C is selected for the result set of the query; the query is resolved when iteration of :C over all objects in Company is complete.

In order to compare the semantics of the ORION query language with OOPC, GOMql, and GOMsql, the three queries wriiten for these languages will be translated into the Smalltalk-like syntax of ORION. Recall Query 1, which requests the salary of all presidents of companies which have a manufacturing division :

```
Query 12 : (
            (Company select :C ( :C Divisions some :D
                                   ( :D Name = "Manufacturing")
                                 )
            )
            President Salary
          )
```

Note the use of the **some** selector message in the query expression. This selector returns true, and hence **select** chooses :C for the result set, if there is at least one instance of a division (bound to :D) which has a name attribute domain equal to the string value "Manufacturing". This is ORION's version of the existential quantifier; a similar message selector **all** functions as the universal quantifier. Finally, it is interesting to observe that the inner query returns objects of class Company, and the message "President Salary" is sent to this set object (a subset of all objects of class Company) to return the result of the outer query.

As mentioned previously, the restriction on object types in the result set of an ORION query can impact the possible queries which may be devised. For example, Query 2, which requires both the name and address of all department managers who earn more than $50,000 per year and drive a company car with a year model prior to 1985, cannot be written in ORION because of the single object class limitation for the result set. However, it may still be formulated as two separate queries. The query returning the name is :

```
Query 13 : (
        (Department Managers select :M ( :M Salary > 50000
                                        and ( :M Vehicles some :V
                                            (:V Year > 1985)
                                        )
                                    )
        )
        Name
    )
```

The query returning the address is identical, with the "Address" message substituted for the "Name" message.

Query 3, retrieve the license numbers of all red mid-engine vehicles and blue long-bed vehicles, could be proposed as :

```
Query 14 : (
        (*Vehicle select :V ( :V Color = "red"
                            and ( :V Engine_location = "mid")
                            or  (:V Color = "blue")
                            and ( :V Bed_type = "long")
                            )
        )
        License
    )
```

In this query, the asterisk operator (*) further defines the function of the **select** message to retrieve all instances of Vehicle and its subclasses. This poses the same problem that occured with Query 3 in OOPC; if on a particular iteration of the query, the Vehicle object bound to :V is sent the "Engine_location" message, an error may occur in processing the query if an object of class Truck (for which Engine_location is undefined) is queried. To solve this problem, the query must be divided into separate queries :

```
Query 14A : (
              ( Car select :C ( :C Color = "red"
                                and ( :C Engine_location = "mid")
                              )
              )
              License
            )

Query 14B : (
              ( Truck select :T ( :T Color = "blue"
                                and ( :T Bed_type = "long")
                              )
              )
              License
            )
```

E. GORDAS

GORDAS (Elmasri,1981) is a formal high level query language based on Chen's entity-relationship (ER) model. It is unique from a semantic standpoint in that no variables are required in formulating queries; as will be seen, this is one reason queries in GORDAS are somewhat easier to understand than some other languages. Also, queries in GORDAS are derived directly from a high-level conceptual representation of a database schema. As a result, a more natural language query interface is possible. Unfortunately, the language is not designed specifically for object-oriented database schemas. To solve this problem, an extension of the base conceptual model is proposed.

GORDAS is well-described by its name (Graph-Oriented Data Selection). The first step in making GORDAS queries is to construct a high-level conceptual schema from an implementation schema. This conceptual schema will be the OPERA schema developed in Chapter IV. This schema is then mapped into a directed graph called the *schema graph*, or

SG, which represents the intension of the database to be modeled. GORDAS queries are derived directly from the SG.

First, the GOOSE schema diagram of Figure 8 must be transformed into an OPERA model representation. Figure 9 is the result of this mapping. The schema graph (Figure 10) is formed from the OPERA schema as follows :

1. For each superclass entity type in the OPERA schema, construct a corresponding node in the SG. Each such node is labeled (ES, <entity set name>, <:>, <attribute names>). This is the entity set ES_1.
2. For each subclass entity type in the OPERA schema, construct a corresponding node in the SG. Each node is labeled (ES, <entity set name>, <:>, <attribute names>). Only attribute names specific to the subclass are named. This is the entity set ES_2.
3. The entity set $ES = ES_1 \cup ES_2$.
4. For each relationship type in the OPERA schema, construct a corresponding node in the SG. Each such node is labeled (RS, <relationship set name>, <:>, <attribute names>). This is the relationship set RS.
5. Nodes are numbered consecutively beginning at one. Two nodes may not have the same number.
6. Color all relationship nodes identically; color all entity nodes identically, but different from relationship nodes.
7. For each superclass/subclass ES node, construct an undirected link labeled by the subset symbol ($\subset$).
8. For each entity-relationship pair, construct a link which is directed from the ES node to the RS node. Label this link (<far entity name, near entity name>,<:>, <constraint c_1,c_2>)). <Far entity name> refers to the entity participating in RS in the direction of the link, and <near entity name> refers to the entity participating in RS in the direction opposite the link. <Constraint c_1,c_2> limits the number of relationship instances for a participating ES nodes to the range (c_1,c_2). The default value for (c_1,c_2) is (0, ∞). This is the directed edge set D_E.
9. Form a directed, colored graph G = (V,E), where V consists of all nodes n such that ($n \in$ ES) $\vee$ ($n \in$ RS), E consists of all edges e such that $e \in D_E$, and no edge e exists such that two nodes n of the same color are directly conected.

Steps 1,2,3,6,and 7 are extensions to the GORDAS model which allow for object-oriented modeling (superclass/subclass inheritance) and greater clarity of representation (graph coloring).

Before defining the sematics of a GORDAS query, it must be mentioned that whereas the intension of a database is modeled by an SG, the extension may also be modeled by a directed graph called the *database graph* (DBG). This graph is derived from the SG; its labeling is similar, with the notable exception that attribute values are shown to indicate a particular instance of a node in the SG. In addition, a DBG uses the node numbering scheme of the SG to group RS instance nodes. The mapping of SG to DBG is beyond the scope of this paper; more details are found in Elmasri (1981).

In order to construct a GORDAS query, one must be able to traverse the SG (and concurrently, the DBG). This is realized via the *path expression*. A path expression may be of the form [N] or [P of N], where N is a node from the SG and P is a *prefix* of the path expression. A prefix may be an attribute name, another node, or a *connection name*. A connection name determines the direction of traversal in the path expression; it is indicated by an edge label in the SG. A path expression evaluates to a single value or a set of values from the DBG.

To see how a path expression works, refer to Figure 10. The path expression [COMPANY] would specify the single node COMPANY in the SG, and all instances of this node in the DBG. The path expression [division of COMPANY] would specify a set of DIVISION nodes for each COMPANY node. In such a path expression, the directed

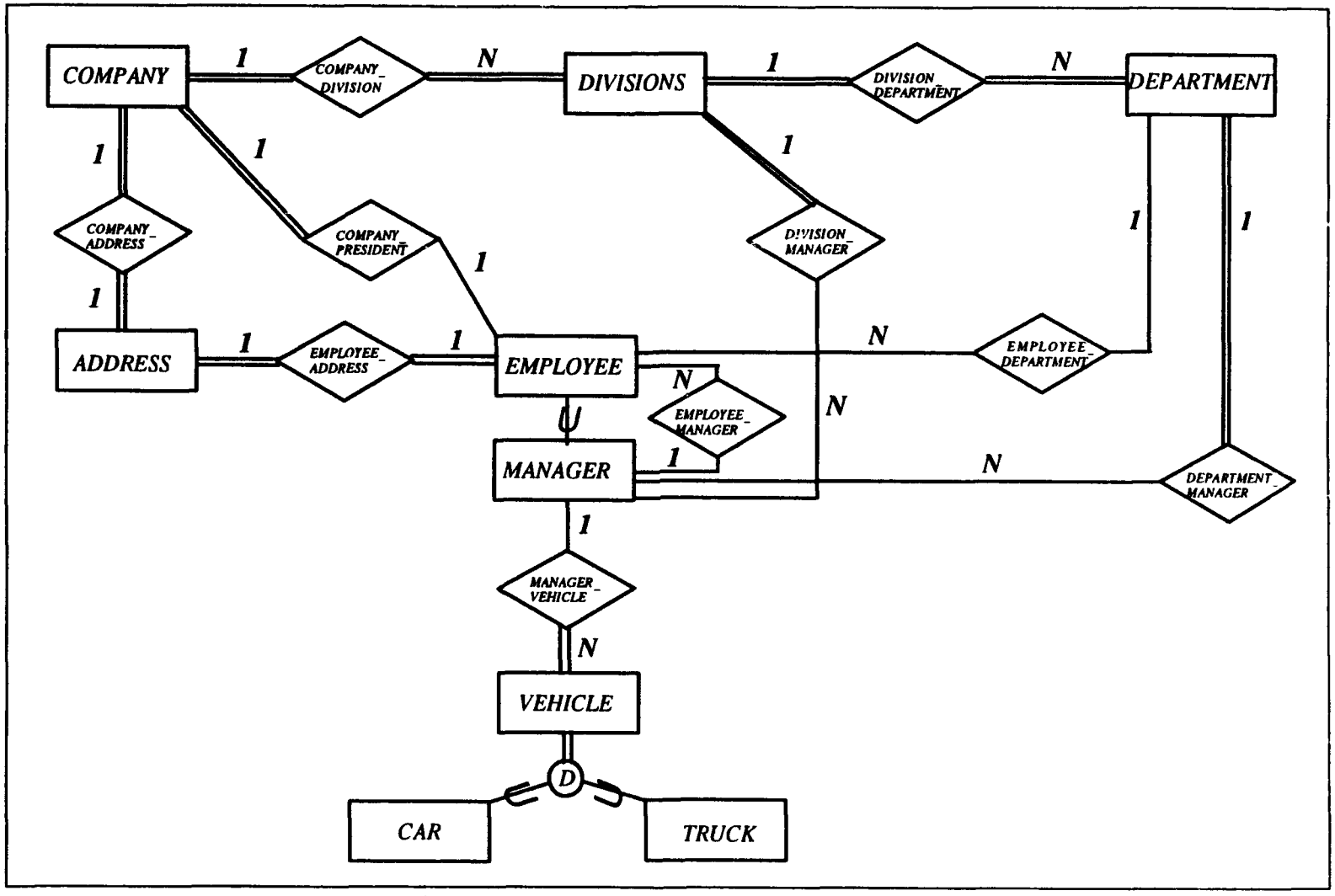


Figure 9. Company Database OPERA Schema

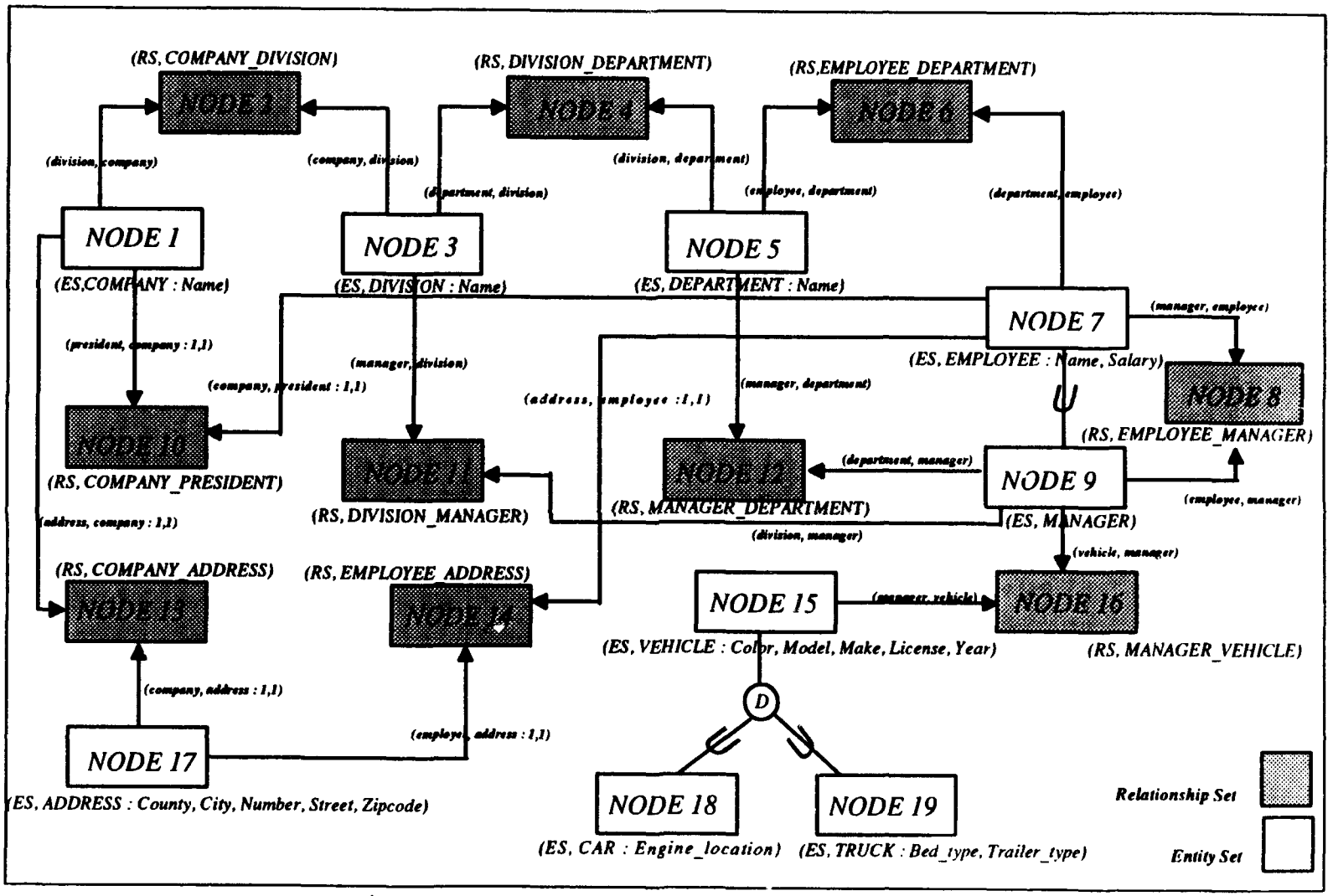


Figure 10. Company Database GORDAS Schema Graph

graph is followed from the *last reached node* COMPANY to the RS node COMPANY_DIVISION via the connection name *division*, and followed from the RS node via the same connection name to DIVISION. Every path expression must be well-specified; that is, a connected path must exist from the last reached node in the path expression (the node at the end of the expression) to any other node reached by traversing the path expression in a direction away from the last reached node.

Path expression may also contain predicates. For example, the expression [Name of employee : (Salary > 50000) of DEPARTMENT] would indicate the names of all instances of the set EMPLOYEE who make more than $50,000 per year and belong to a set DEPARTMENT. Note how the path is traversed : starting at the last reached node DEPARTMENT, the connection name *employee* is followed to EMPLOYEE, where the predicate (Salary > 50000) is applied. The Name attribute is then used with this restricted set to specify the instances desired. Note the two prefixes in the expression : <Name > and <employee : (Salary > 50000)>. Thus path expressions may contain nested prefixes.

A GORDAS query is specified on a schema graph SG, and retrieves information from the corresponding database graph DBG . The query is formed from two clauses : the **GET** clause and the **WHERE** clause (Elmasri, 1981, p.60). The GET clause contains one or more path expressions representing SG nodes N_1 , N_2 , ... , N_n. The cartesian product C = N_1 x N_2 x ... x N_n is taken, and to this set is applied the predicate P of the WHERE clause. Each element c_i of C is returned as the result set $\{c_1 , c_2 , ..., c_k \}$, where $1 \leq k \leq n$ and each c_i satisfies P.

The three queries which have been written in OOPC, GOMsql, GOMql, and ORION may now be drafted in GORDAS. First, Query 1, which requests the salary of all presidents of companies which have a manufacturing division :

Query 14 : ***GET*** *Salary of president of COMPANY*
WHERE *division [of COMPANY]* ***INCLUDES***
(***GET*** *division of COMPANY*
WHERE *name = "manufacturing")*

Note the use of brackets in the WHERE clause. The phrase [of COMPANY] is added for clarity, but is not required since *division* is bound to COMPANY in the GET clause. The predicate of the WHERE clause uses an operator **INCLUDES** to compare two sets (one of which is a nested query) for membership in the result of the query.

Query 2 (get the name and address of all department managers who earn more than $50,000 per year and drive a company car with a year model prior to 1985) is stated as:

Query 15 : ***GET*** *<Name, Address> of manager of DEPARTMENT*
WHERE *(Salary of manager [of DEPARTMENT] > 50000)*
AND
(vehicle of manager [of DEPARTMENT] ***INCLUDES***
GET *vehicle of manager of DEPARTMENT*
WHERE *Year of vehicle of manager [of*
DEPARTMENT] < 1985

In this query, note that the attributes *Name* and *Address* of the ES node MANAGER are inherited from the ES node EMPLOYEE. This represents an extension to the GORDAS query language to allow for subclass inheritance of attributes in processing of query statements.

Query 3 (get the license numbers of all red mid-engine vehicles and blue long-bed vehicles) is written in GORDAS as :

Query 16 : ***GET*** *License of Car, License of Truck*
WHERE
(Engine_location of Car = "mid" ***AND*** *Color of Car = "red")*
OR
(Bed_type of Truck = "long" ***AND*** *Color of Truck = "blue")*

In this query, the GET clause contains two path expressions. Together these represent the cartesian product of all DBG nodes which are cars or trucks (i.e., vehicles).

F. CONCLUSION

Several object-oriented query languages have been examined in this chapter. The main criterion for comparison of these languages is their relative ease of understanding; that is, semantic clarity. GORDAS is clearly the easiest to understand. A good example of this is Query 16. Even someone with no knowledge of query language syntax could determine what the purpose of the query is. On the other hand, OOPC expresses this (as Query 3A) in an arcane way, using the CLASS_OF operator to make it work. GOMql uses the **exist** operator (Query 6) and a parenthetical structure which is not as easy to evaluate logically. GOMsql (Query 10) is more lengthy to formulate than the GORDAS version, and nesting of queries does not help the meaning. Finally, in addition to being split into two separate queries (Query 14A, 14B), the ORION version is heavily dependent on Smalltalk syntax.

GORDAS was implemented during the 1980's as a stand-alone, high-level, non-procedural query language interface based on the Entity Category Relationship Model (ECR), an extension of the ER model incorporating generalization, subclasses, and entity grouping based on relationship roles. Queries in this implementation are translated to a relational algebra internal form for processing (Elmasri, 1981).

The utility of GORDAS as a semantic vehicle for query formulation is primarily due to its natural language quality. This is a direct result of its mapping from a high-level schema to an associated query graph, which is in turn used to write queries. Although the semantics of GORDAS are more clear than other query languages, its overhead is much greater; a big effort must be put into developing a query graph in order to write queries. Still, for an object-oriented schema, queries are more concise and understandable with this language.

VI. MODELING THE EWIR DATABASE

A. INTRODUCTION

The Electronic Warfare Integrated Reprogramming Database, or EWIRDB, is the primary Department of Defense (DOD) approved source for technical parametric and performance data on noncommunications emitters (National Air Intelligence Center, 1994). Its primary purpose is to provide an up-to-date and accurate source of information for reprogramming United States electronic warfare (EW) combat systems such as radar warning receivers, combat identification systems, electronic jammers, anti-radiation missiles, and other target sensing systems. A variety of information is included in the EWIRDB, including parametric data on radars, jammers, navigational aids, identification friend or foe (IFF) equipment, and numerous noncommunications electronic emitters. Secondary objectives of the EWIRDB include support of EW systems research, development, test, and evaluation; modeling and simulation; combat operations planning; and EW tactics and training.

The EWIRDB was developed initially by the U.S. Air Force in the 1970's but has become a joint service product, involving input by all branches of the U.S. Armed Forces, DOD, the National Security Agency (NSA), and various other intelligence agencies. EWIRDB data is both *observed* and *assessed*. The observed parameters are obtained from the KILTING database maintained by NSA. Assessed data is provided by two sources, Scientific and Technical Intelligence (S&TI) Centers and the Air Force

Information Warfare Center (AFIWC). DIA (Defense Intelligence Agency) is responsible for maintenance of the S&TI assessed data while AFIWC maintains the United States Noncommunications Systems Database (USNCSDB). The three sources (KILTING, S&TI Assessed Data, USNCSDB) are merged into the EWIRDB, which is maintained by the National Air Intelligence Center (NAIC). Figure 11 illustrates this process.

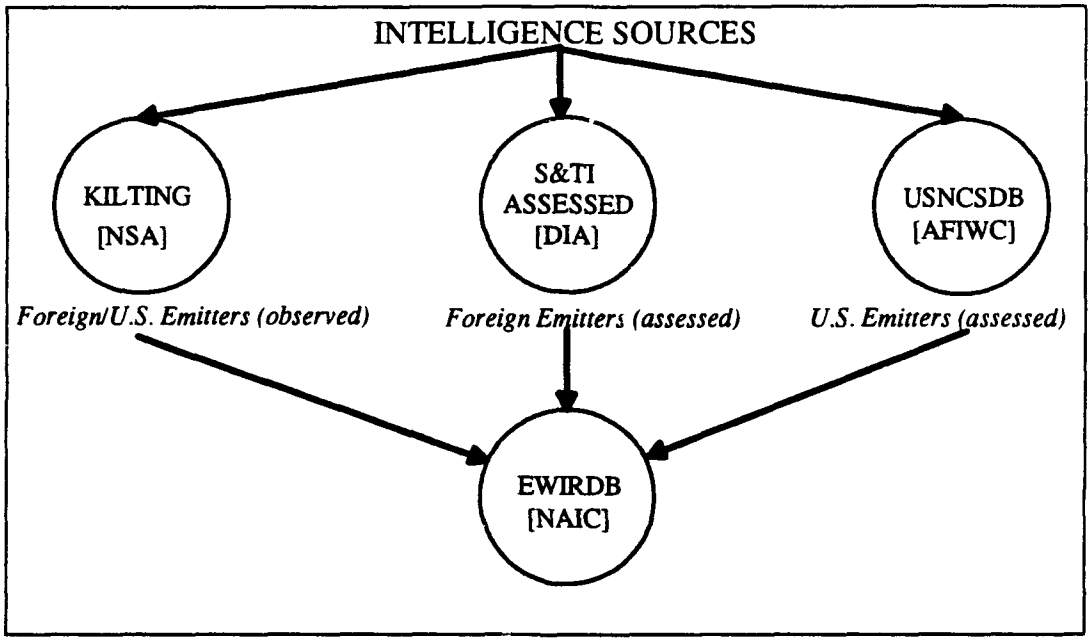


Figure 11. EWIRDB Synthesis

Although extremely important to our nation's warfighting capabilities, the EWIRDB is not easy to comprehend. Its format is difficult to interpret, assessed/observed codes are not standard for all record types, and the database is generally described in terms of the physical storage structure. The focus of this chapter is to examine the form and content of the database, represent it as an object-oriented schema, map this schema into a high-level model of the database, and create a GORDAS schema graph for EWIRDB queries.

B. EWIRDB STRUCTURE

1. Storage Structure

Parametric data associated with electronic emitters is represented as a logical tree, although the actual storage struc[illegible]record type. This *parametric tree* orders a long list logically and hierarchically in a way that proceeds from broad characteristics through levels of successively finer ones (National Air Intelligence Center, 1994). Parametric data exists in *subfiles* of the tree structure. Subfiles are major groupings, or subtrees, within the parametric tree which contain logically related data.

Emitter parametric data may exist in one of three tree types : (1) Pulsed/ Continuous Wave (P/CW), (2) Receiver Performance Assessment (RPA), or (3) Electronic Countermeasures (ECM). The PCW tree is used primarily for signal identification, the RPA tree is used in electronic countermeasures design, and the ECM tree is used for radar electronic counter-counter measures (ECCM) applications. The edges, or branches, of all three tree types are indexed by a number and additionally, a one or two character subfile code. Branches which only function to connect subfiles together are called *superheaders* and carry a numerical designation only. Figures 12, 13, and 14 show the top-level logical storage hierarchies for the three parametric tree types.

2. Record Format

The EWIRDB product contains many types of information in addition to emitter parametric data. The standard storage and distribution format for this data is the Technical Electronic Intelligence Reference File (TERF). The TERF format consists of six different record types, which are designated S00, S01, S02, S03, S04, S05, and S06. A

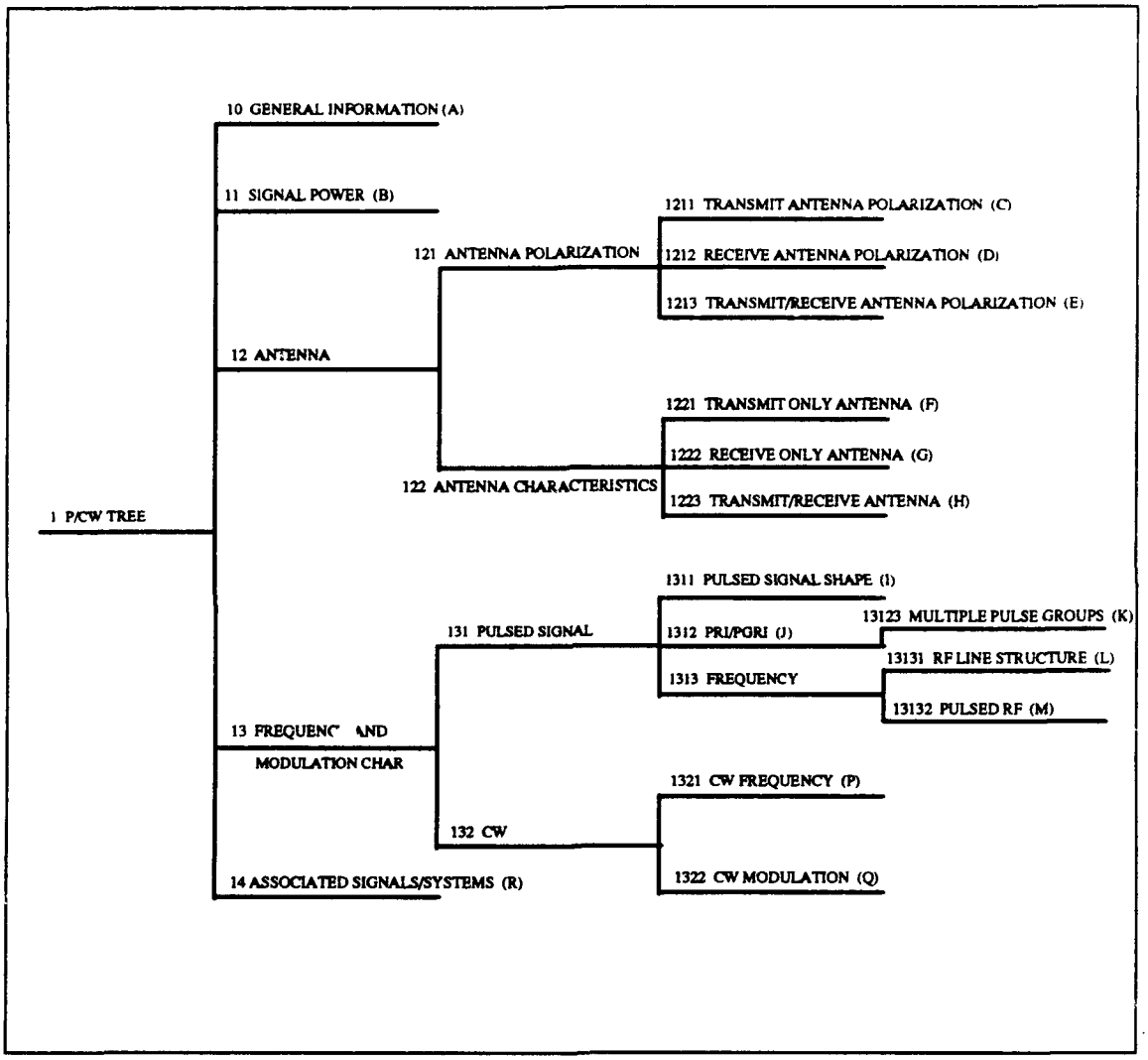


Figure 12. P/CW Parameter Tree

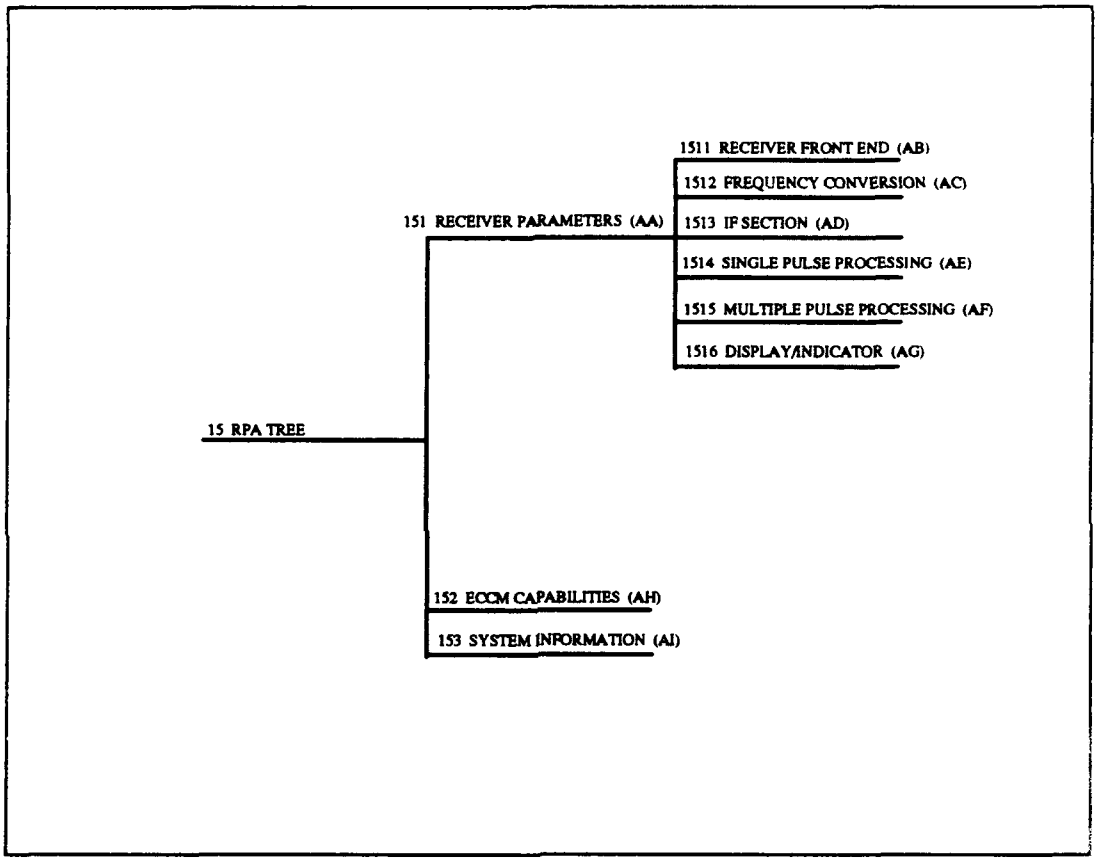


Figure 13. RPA Parameter Tree

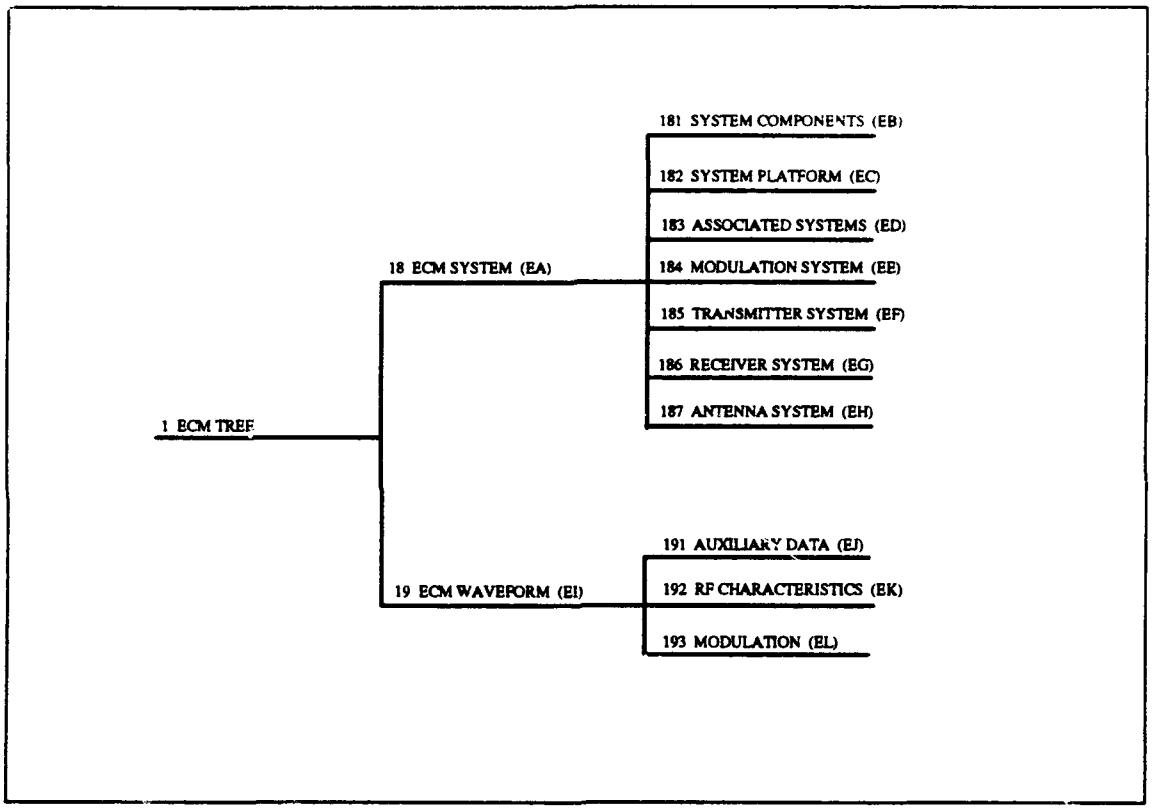


Figure 14. ECM Parameter Tree

single emitter file may consist of multiple instances of each record type, and also record types from different sources. For example, an emitter may have a number of SO2 records, and each of these may be derived from the KILTING, S&TI, or USNCSDB sources. Figure 15 shows a sample emitter file with all record types and data sources. Before proceeding to map the record types into an object-oriented schema, a description of each type is needed.

a. Classification Record (S00)

This record is unique in that there may be only one per emitter file It defines the overall classification for the emitter, which is the highest classification of any component record from the three EWIRDB input sources. The first three data fields in the S00 are common to all the other record types. The *Record Type* is a string denoting the type of TERF record (S00, ... , S05). The *Source Designator* is a character which indicates the EWIRDB input source (K = KILTING, E = S&TI, U = USNCSDB). The *Notation* is a five character alphanumeric known as the ELNOT (electronic intelligence notation) which is unique to each emitter. The remaining two fields in the S00 are the *Classification,* which shows the classification of the emitter file, and *Retrieval Date*, which indicates when data was provided from KILTING (observed data only).

b. Emitter Name Record (S01)

At least one emitter name record must be provided per emitter file. After the first three data fields is the *Emitter Name*. This is string value which is the common name for the emitter hardware (such as a radar name). Next are two fields which differ depending on whether the data source is observed or assessed. For KILTING data, the

SAE Code and *Date of Last Significant Change* follow, and for S&TI/USNCSDB, the *S&TI Code* and *Multiple Source Review Date* occur. *SAE Code* is a four-character code which indicates the agency responsible for the ELNOT, and *Date of Last Significant Change* is a six-digit date code (YYMMDD) which is the date of last review of the source KILTING database. *S&TI Code* and *Multiple Source Review Date* are the assessed data analogs of *SAE Code/Date of Last Significant Change*. The last field, *Parametric Update Date*, is common to assessed and observed data and provides the most recent change to any S03, S04, or S05 record.

c. Subfile Header Record (S02)

Although this record contains no emitter parametric data, it is required for any subfile containing parametric data. In addition to the initial three data fields, all S02 records have the next three fields in common : *Subfile Tree Number*, a 12-digit number which is the parametric tree index for the subfile (see Figures 2-4); *Subfile Name*, a 25-character field containing the associated EWIRDB subfile name; and *Subfile Code*, containing the one or two-character code for the parametric tree subfile. In observed records only, the *Technical Date* precedes the *Subfile Code* and indicates when the last change was made to any S03 record in the subfile.

d. Parametric Data Record (S03)

The S03 record is the most useful, since it actually contains the emitter measurement names and indexes into the parametric tree where the data is stored. For both observed and assessed data, the first nine fields are identical. After the record type, source code, and ELNOT, come the following fields : *Tree Number*, a 15-digit number

which is the index into the parametric tree; *Suffix Code*, a two-character code used to identify operating modes of the emitter; *Measurement Name*, a character string which is the name of the value stored at the parametric index; *Units*, a string describing the scale of data or format of text entered in the tree; *Lower/Upper Value Text*, which, for numeric entries, gives the possible range of values which may be entered, and a single text string otherwise.

The next two fields differ depending on the data record source. For assessed data, *Confidence Level* and *S&TI Code* follow; for observed data, *Measurement Accuracy* and *Measurement Accuracy Units*. *Confidence Level* is a one-digit code which is a system analyst's confidence in the parametric data; *S&TI Code* is the same as in the S01 record. *Measurement Accuracy* and *Measurement Accuracy Units* are seven and three-digit fields which describe the accuracy (if available) for the parametric data units.

Reference Number Comment Number, and *Reserve Mode* are common to observed and assessed data. *Reference Number* (R = KILTING, A = S&TI, F = USNCSDB) and *Comment Number* (C = KILTING, K=S&TI, N = USNCSDB) are used to connect the S03 record to an S04 reference record or to an S05 comment record. Both are four-character fields, the first character being the code for the data input source. *Reserve Mode* is a numerical value showing (1) that an emitter mode is a wartime reserve mode (WARM) and (2) the system analyst's confidence level in this assignment.

The next field is either *Classification* (assessed) or *Intelligence Source* (observed). *Classification* provides a one-character code for classifying data in the

Lower/Upper Value or *Text* fields; *Intelligence Source* (one-character code) is only used if multiple intelligence sources were used to derive S03 records.

The last three fields in the S03 record are *Subfile Code*, *Releasability KILTING Preferential Rating*, and *Date of Last Update*. The *Releasability* field, applicable to assessed data, is a two-character code representing the countries which may have access to the S03 data. *KILTING Preferential Rating* is a one-digit system analyst rating for relative importance of different KILTING data entries. Both *Subfile Code* and *Date of Last Update* are used by all record sources. *Subfile Code* is identical to the S02 entry and *Date of Last Update* reflects the last significant change to the parametric data.

e. Reference Data Record (S04)

S04 records are used to store information on references which are connected to parametric data record (S03) entries. These reference records enable system analysts to trace parametric data back to source documents. All six fields are shared by assessed and observed records; however, the last field has a different format for both types.

After the record type, source, and ELNOT are three fields which are identical for all source types. The *Reference Number* field is the same as the S03 entry, and serves as a link to the S03 data. *Line Number* is a three-digit number which signifies the line of reference text. *Reference Text* is a string which describes the particular reference, but its format varies. For assessed data, this field contains information on the published reference; the last line for each reference also contains a classification of the

reference text data and its releasability to other nations. For KILTING records, the reference text is a formatted field which contains specific information on each successive line, such as document number, document title, report date, and producer of the document. The last entry is the report classification, which is a text string.

f. Comments Record (S05)

This record contains two primary types of information : (1) amplifying descriptions of data contained in the S03 records, and (2) suffix tables. A suffix table is a matrix with rows and columns of suffix codes which may be mathematically manipulated to calculated all possible operating modes of an emitter. A detailed examination of suffix codes/tables is beyond the scope of this paper, but is described in National Air Intelligence Center (1994).

The first three fields of the S05 are identical to all other TERF records, The next three are unique to the S05. They are *Comment Number*, *Line Number*, and *Comment Text*. *Comment Number* is identical to the entry in the S03 field; it is the link to a parametric data record. In addition, it has a unique comment number ("0000") which is used for listing and describing the suffix table. *Line Number* is analogous to the same field in the S04 record. *Comment Text* contains the amplifying text (including suffix table). For assessed records only, a classification comment text line follows a series of comment lines and provides the classification of the text preceding it. The fields in this type of comment are *Classification Code* and *Releasability Code* (which may be blank if non-applicable).

```
S00KTPN19UNCLASSIFIED                                                                                                    940101
S01KTPN19AN/TPN-19                                                                                                    920604920604
S01ETPN19AN/TPN-19                                                                                                    NMIC940206940206
S01UTPN19AN/TPN-19                                                                                                    USF 930204930204
SO2KTPN19  000000000010   GENERAL INFORMATION   940106                                                                    A
S03KTPN19  000000000010.20++ EMITTER FUNCTION   (TEXT)   PA                          DEFR001C001   A   6   920604
S02ETPN19  000000000010   GENERAL INFORMATION                                                                           A
S03ETPN19  000000000010.14UA USER COUNTRY       (TEXT)   US UNITED STATES   2      NMIA001K013   UA       940126
S02UTPN19  000000000010   GENERAL INFORMATION                                                                           A
S03UTPN19  000000000010.15++ EMITTER DESCRIPTION (TEXT)  APPROACH          1        USFF002N001   UA       930924
S02KTPN19  000000000011   SIGNAL POWER            940106                                                              B
S03KTPN19  000000000111.10 PEAK POWER             DBW   83.5    83.5              CLCR001C002   B   4   920604
S04ETPN19A001001   JANE'S RADAR AND ELECTRONIC WARFARE SYSTEMS (1991-1992)
S04ETPN19A001002                                                                  ***(U/ )
S04KTPN19R001001   GCA RADARS, THEIR          IEEE PROCEDINGS      740600   IEEE/UNC
S04KTPN19R001002   HISTORY AND STATE OF       VOL 62, NO. 6, JUNE
S04KTPN19R001003   DEVELOPMENT                1974, PP 705-716
S05KTPN19C000001   ANALYST'S SUFFIX TABLE
S05KTPN19C000002
S05KTPN19C000003     = A,
S05KTPN19C000004     = B,
S05KTPN19C000005   1 =   , S, S1, S2, S3,
S05KTPN19C000006   1 =   , T,
S05UTPN19N001001    AN/TPN-19 (FIXED) AND AN/GPN-22 (MOBILE) ARE GROUND BASED
S05UTPN19N001002    PRECISION APPROACH RADARS USED BY THE AIR FORCE AND EXPORTED
S05UTPN19N001003    TO SOME COUNTRIES.
S05UTPN19N001004    UP TO SIX AIRCRAFT CAN BE TRACKED AND CONTROLLED
S05UTPN19N001005    SIMULTANEOUSLY.
S05UTPN19N001006    THE RADAR CAN OPERATE IN LIGHT AND MODERATE RAIN CONDITIONS.
S05UTPN19N001007                                                                                          ***(U/ )
```

Figure 15. TERF Record Format

C. OBJECT-ORIENTED EWIRDB

The first step in developing a high-level conceptual model for the EWIRDB is to map the TERF record format into an object-oriented schema. This can be achieved by converting the basic record structure into a class structure. That is, each record type in the TERF is mapped into an object class. With such a structure, the importance of assigning unique ELNOTs to each emitter file is no longer significant in regard to physical storage, since unique object identifiers are assigned to each emitter file object. ELNOTS, however, retain their importance from a logical classification viewpoint.

Figures 16 and 17 depict the TERF format in the object form of a GOOSE diagram. This schema represents all the information contained in the various record types, but in a more meaningful way. Whereas the TERF format tends to model data in a low-level (physical) manner, the object schema provides an implementation (logical) model for representing the EWIRDB. As can be readily seen, the object schema groups properties (fields) which are common to both observed and assessed data in a single superclass for each record type, and lists properties unique to each data type in a separate subclass. The object model also converts some TERF fields into methods for efficiency of data retrieval. This schema can be directly mapped into a high-level conceptual (OPERA) schema, which in turn is used to develop a schema graph for GORDAS queries. First, it is helpful to examine how each record is mapped into the object class structure.

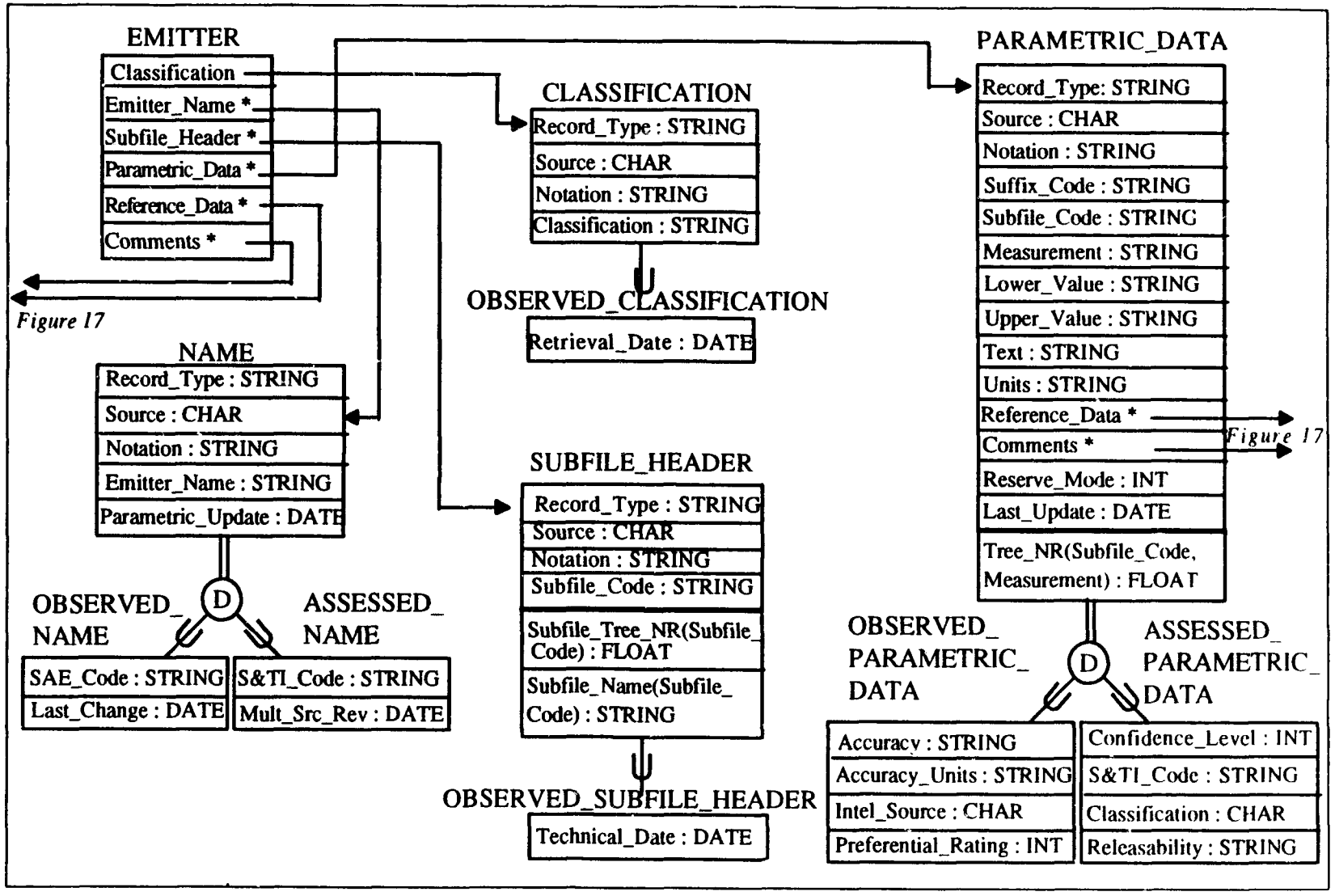


Figure 16. EWIRDB Object Schema

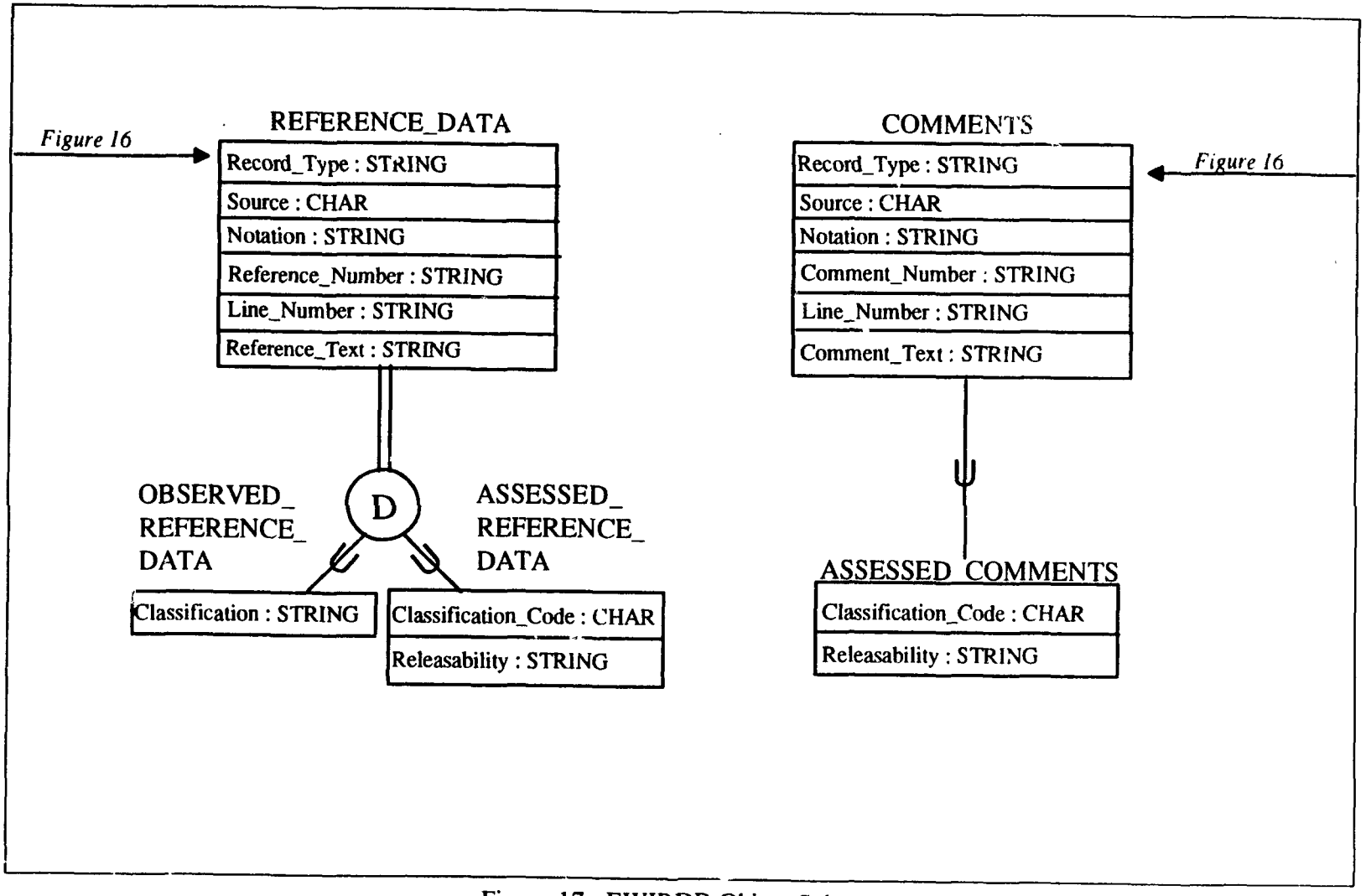


Figure 17. EWIRDB Object Schema

1. Emitter

This class does not correspond directly to a specific TERF record, but rather to an entire emitter file (multiple TERF records). The property *Classification* has as its domain objects of class *Classification*. It is also single-valued, reflecting the fact that an emitter file has one and only one classification record. In other words, its domain is a singleton set. The other properties of *Emitter* are multivalued, i.e., they return sets of objects of the following classes : *Emitter_Name*, *Subfile_Header*, *Parametric_Data*, *Reference_Data*, and *Comments*.

2. Classification

The *Classification* object class corresponds to the S00 record. It is modeled as a superclass with properties common to observed and assessed data, and also as a subclass *Observed_Classification* for the additional property unique to KILTING data. Of course, all KILTING data objects inherit the properties of *Classification*.

3. Name

This is the object counterpart to the S01 record type. Again, there are five properties corresponding to the S01 fields which are common to both data types, and two subclasses, *Observed_Name* and *Assessed_Name*, which have properties specific to KILTING or S&TI/USNCSDB source data.

4. Subfile_Header

The *Subfile_Header* class contains properties analogous to the S02 record fields, but differs in implementation. The subfile tree number and subfile name are returned by object methods instead of properties. By passing *Subfile_Code* as a parameter to *Subfile_Tree_NR* and *Subfile_Name*, the subfile codes and tree numbers for all

Subfile_Header objects may be retrieved by the same block of code. This allows all the tree numbers and names to be stored centrally and eliminates the need to duplicate them in a property for each object (or field for each record). A subclass *Observed_Subfile_Header* inherits these two methods (and four properties) of *Subfile_Header*, and includes an additional property.

5. Parametric_Data

This class corresponds to the S03 record, but is distinct in several ways. First, there are two subclasses, *Observed_Parametric_Data* and *Assessed_Parametric_Data*, with properties in addition to their superclass. Also, these subclasses inherit a method *Tree_NR*, which takes as parameters the subfile code and measurement name and returns the tree number for the parametric data. As with *Subfile_Header*, this technique saves storage space and allows all tree numbers to be stored once. Finally, instead of storing an explicit reference number and comment number, *Reference_Data* and *Comments* return sets of objects of the classes containing references and comments associated with objects of the *Parametric_Data* class.

6. Reference_Data

This class consists of objects which are domain elements of *Emitter* and *Parametric_Data*. The properties correspond to the S04 record fields, and two subclasses exist to show the differences in classification coding between assessed and observed data.

7. Comments

Similar to *Reference_Data*, this class exists in the domain of *Emitter* and *Parametric_Data* and contains properties which are the fields of the S05 record type. Only one subclass, *Assessed_Comments*, is required to model comment classification for assessed data objects.

D. THE CONCEPTUAL EWIRDB

The object-oriented schema developed in section C may now be mapped into a high-level conceptual model. The model used for this representation is the OPERA model of Chapter IV. The reasons for creating a conceptual model are (1) to provide a simplified expression of data requirements and relationships for the database end user and (2) to furnish the producers of the database a model for the conceptual design of the EWIRDB.

Figure 18 is the OPERA representation of the object schema shown in Figures 16 and 17. Recall that individual properties and methods are excluded in order to focus on the relationships between classes. One feature of the OPERA representation which is not evident in the object or TERF formats is modeling of *constraints*. For example, the OPERA schema clearly shows that an emitter file may be related to many subfile headers (in fact, it must contain at least one) but a subfile header may be related to many emitter files (or to none). In the OPERA schema such constraints are inherited by subclasses. For example, an *Observed_Reference_Data* object must be related to some *Emitter* object , but an emitter object may be related from zero to many *Observed_Reference_Data* objects, since an emitter may be discovered which has no documented references.

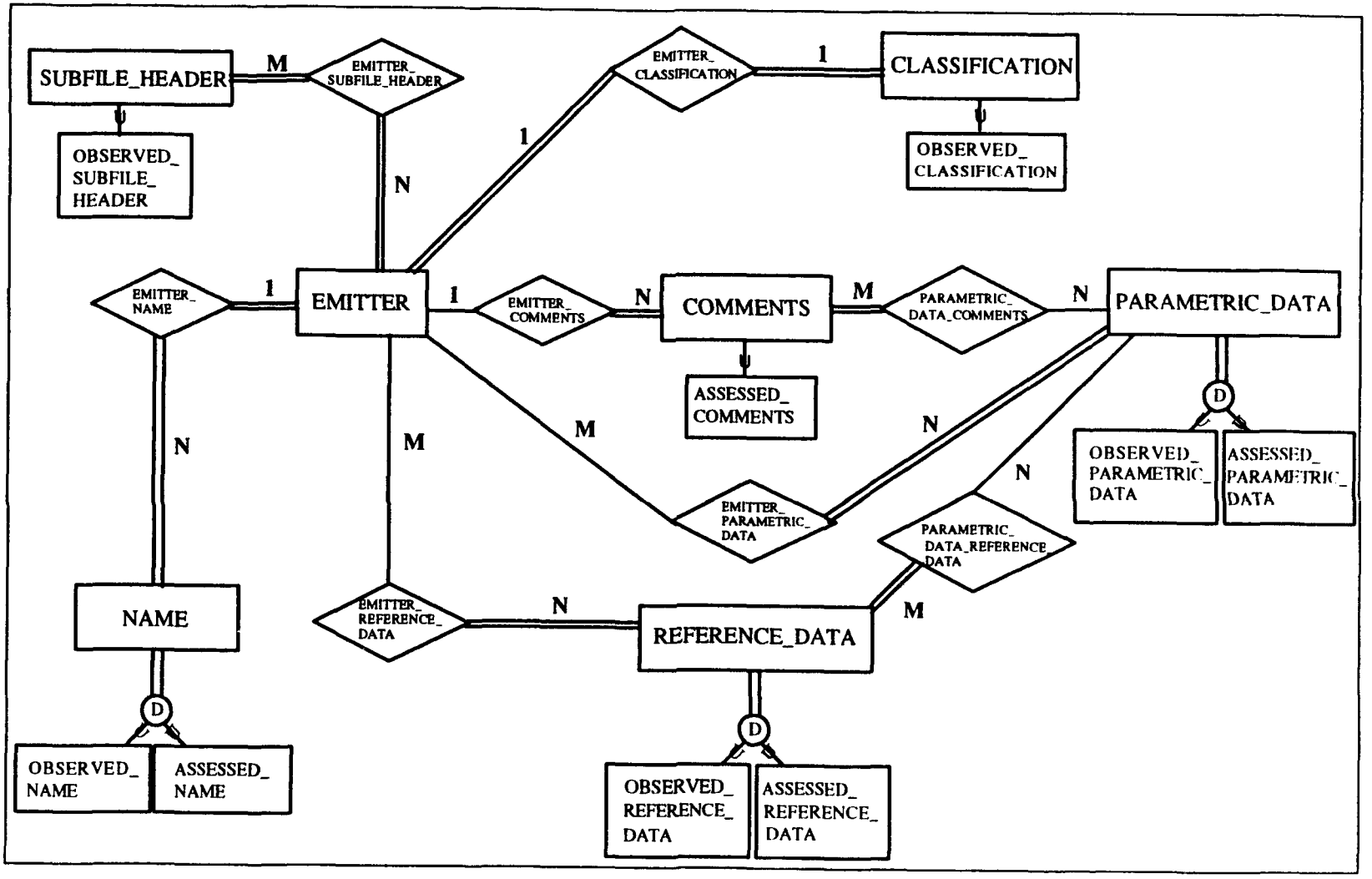


Figure 18. EWIRDB OPERA Schema

Similarly, an *Assessed_Comments* object must be related to at least one *Parametric_Data* object, since a comment cannot exist independently, and may be related to many *Parametric_Data* objects, as is the case with suffix table comments. As a final example, note the relationship *Emitter_Classification*. This clearly shows that an emitter file must have one and only one classification, and that a classification which exists in the database must be related to some emitter file.

E. GORDAS QUERIES

Using the mapping procedure of Chapter V, a GORDAS schema graph may be developed from the OPERA schema of Figure 18. This graph, which may be used to write queries against the EWIRDB, is illustrated in Figure 19.

One of the functions of the suffix code is to determine the possible operating modes of an emitter. The "++" suffix code indicates that a particular parameter has a value for all operating modes. Suppose that a requirement exists to retrieve from the EWIRDB the names of all emitters which have a suffix code of "++" for the parameter *scan angle coverage*. This information may be extracted by the following query :

Query 1 : GET *Emitter_Name of name of EMITTER*
WHERE *(Suffix_Code of parametric_data [of EMITTER] = "++")*
AND
(Measurement of parametric_data [of EMITTER] = "scan angle coverage")

As discussed in Chapter V, the use of brackets ([ ]) is optional. It is used here for clarity only.

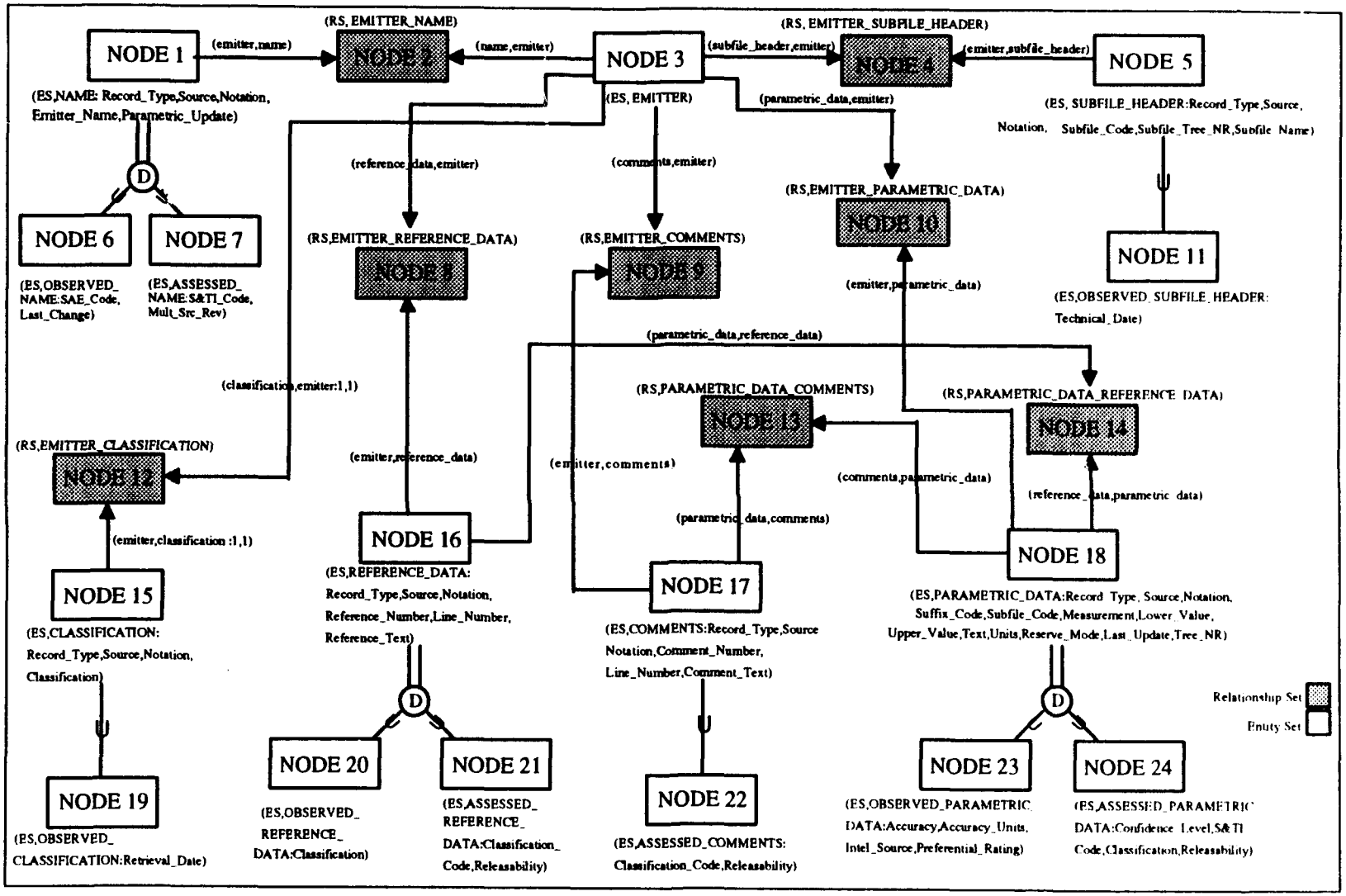


Figure 19. EWIRDB GORDAS Query Graph

Suppose a user of the EWIRDB knows the common name of an electronic emitter, but requires more specific information. For example, for the SPS-10 surface search radar, the ELNOT, parametric tree index, suffix table, and intelligence source for all observed data may be obtained with Query 2 :

```
Query 2 : GET <Notation, Tree_Nr, Comments, Intel_Source> of
                                    parametric_data of EMITTER
          WHERE  Source of parametric_data [of EMITTER] INCLUDES
                 GET Source of PARAMETRIC_DATA
                 WHERE Source [of PARAMETRIC_DATA] = "K"
          AND
                 (Emitter_Name of name [of EMITTER] = "SPS-10")
          AND
                 (Line_Number of comments of parametric_data
                   [of EMITTER] =  "C000")
```

Note how the suffix table is retrieved. Only comments with three trailing zeros belong to a suffix table; the leading digit specifies whether the source is observed or assessed. Also, observe the use of the nested query. This is needed to restrict the query to observed parametric data.

The next query retrieves the subfile names of all emitters with a technical date after March 3, 1987 and whose references are classified TOP SECRET :

```
Query 3 : GET Subfile_Name of subfile_header of EMITTER
          WHERE  Source of subfile_header [of EMITTER] INCLUDES
                 GET Source of SUBFILE_HEADER
                 WHERE Source [of SUBFILE_HEADER] = "K"
          AND
                 (Technical_Date of subfile_header [of EMITTER] > 870303)
          AND
                 (Classification of reference_data of parametric_data
                   [of EMITTER] =  "TOP SECRET")
```

Query 4 retrieves the names of all emitters whose classification exceeds CONFIDENTIAL :

Query 4 : GET *Emitter_Name of name of EMITTER*
WHERE *Emitter_Classification of classification [of EMITTER] = "SECRET"*
OR
Emitter_Classification of classification [of EMITTER] = "TOP SECRET"

Query 4 and Query 1 are similar in form and function. Unlike the other two queries, there is no restriction on what subclasses may be queried; emitter names associated with observed and assessed data sources are returned in the result.

The final query to be examined uses an aggregate function operator to determine the number of parametric data records with the best preferential rating or confidence level:

Query 5 : GET *COUNT of PARAMETRIC_DATA*
WHERE *Source [of PARAMETRIC_DATA]* ***INCLUDES***
*(**GET** Source of PARAMETRIC_DATA*
__WHERE__ Source [of PARAMETRIC_DATA] = "K"
__AND__ Preferential_Rating [of PARAMETRIC_DATA] = 6)
OR
*(**GET** Source of PARAMETRIC_DATA*
__WHERE__ (Source [of PARAMETRIC_DATA] = "E")
OR
(Source [of PARAMETRIC_DATA] = "U")
__AND__ Confidence_Level [of PARAMETRIC_DATA] = 1)

The two nested queries are used to restrict examination of the preferential rating to those objects whose source is KILTING , and restrict confidence levels to those found in objects whose source is S&TI or USNCSDB.

F. CONCLUSION

The EWIRDB is critical to the combat effectiveness of the U.S. Armed Forces. As the primary database for reprogramming electronic warfare system components, it must contain correct and timely information to ensure minimal loss of life during conflict. In peacetime, it is essential for maintaining combat readiness in the areas of tactics and training.

The EWIRDB, although effectively implemented, is inadequately modeled. In particular, National Air Intelligence Center (1994) does not contain a conceptual model of the database, but rather an implementation model. Since the EWIRDB is constantly evolving , a high-level representation would be useful to make design changes to the database. In addition, with the advantages object-oriented database management systems offer, a conceptual object schema would establish a foundation from which to eventually store the EWIRDB in the form of objects (instead of records). The schemas of Figure 16, 17, and 18 provide this basis.

In order for any database to be useful, it must have a query capability. The GORDAS query language was chosen because of the natural way the conceptual EWIRDB schema maps into the query graph, and the ease of query formulation. Other query languages may be more suitable from a processing viewpoint, but such consideration is beyond the scope of this paper.

VII. CONCLUSIONS AND RECOMMENDATIONS

In this thesis we have proposed a conceptual model, OPERA, which takes an object-oriented implementation data model and abstracts it to a level of simplicity exceeding that of the EER model. With this simplicity comes an improved understanding of the state and behavioral characteristics of an object-oriented database. We conclude, based on our examination of the EWIRDB, that such a conceptual refinement can be more meaningful than a record-based relational description.

Several issues were addressed by this thesis. As a starting point, the EER was presented as the foundation for proposing an object-oriented conceptual model. Then, a general data model for an object-oriented database schema was established, the implementation model of Chapter III. We determined that, by modifying the existing EER model to incorporate database behavior, we could represent the object-oriented model in an understandable and meaningful way. The result of this was the primary objective of the thesis : extending the ER model into a high-level graphical representation for the object-oriented model, called OPERA. In doing this, an intermediate schematic representation, GOOSE, was used to aid in transformation from an implementation to a conceptual schema. We also demonstrated the mathematical relation to be a basis for modeling database behavior in OPERA.

Object-oriented query languages were compared in Chapter V. An extension of the GORDAS query language, adapted for use with object-oriented schemas, was proposed.

By converting an OPERA schema to a GORDAS query graph, we showed how queries could be formulated which were easier to understand than the same ones in other object-oriented query languages.

Finally, we applied the OPERA model to a real-world database, the EWIRDB. This database was determined to be a good candidate for analysis because it is (1) critical to U.S. Armed Forces combat capability, (2) inadequately modeled on a conceptual level, and (3) stored in a relational, record-based format. The EWIRDB was successfully mapped from a relational to an object-oriented schema and converted to an OPERA diagram. The OPERA schema was shown to be semantically superior to the EWIRDB record. In particular, superclass/subclass relationships were illustrated which were hidden in the relational schema. Finally, the OPERA schema was mapped to an object GORDAS schema graph, and relevant queries were formulated against the EWIRDB.

The OPERA model has some limitations. For example, object versions are not addressed. This is a very important part of complex database modeling, and should be incorporated as an enhancement. Another area for future investigation is that of database constraints. OPERA only models constraints inherent to the EER model; any additional constraints must be incorporated as methods. For advanced constraints, such as those required to enforce the invariants of schema evolution, some standard way of representing these in OPERA is needed. Also, an investigation into alternate object-oriented query languages, such as those supporting functional data models, would supplement the work done with GORDAS in this thesis. Also, since the EWIRDB is basically a relational

database, OPERA should be used to describe an existing object-oriented database with complex modeling requirements.

LIST OF REFERENCES

Banerjee, J., and others, "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems*, v. 5, pp. 3-26, January 1987.

Banerjee, J., Kim, W., and Kim, K., "Queries in Object-Oriented Databases," *Proceedings of the Fourth International Conference on Data Engineering*, pp. 31-38, February 1-5, 1988.

Bertino, E., and others, "Object-Oriented Query Languages: The Notion and the Issues," *IEEE Transactions on Knowledge and Data Engineering*, v. 4, pp. 223-237, June 1992.

Bertino, E., and Martino, L., *Object-Oriented Database Systems*, pp. 12-80, Addison-Wesley, 1993.

Berzins, V., and Ketabchi, M., "Modeling and Managing CAD Databases," *Computer*, pp. 93-102, February 1987.

Chen, P., "The Entity-Relationship Model-Toward a Unified View of Data," *ACM Transactions on Database Systems*, v. 1, pp. 9-36, March 1976.

Dos Santos, C., Neuhold, E., and Furtado, A., "A Data Type Approach to the Entity-Relationship Model," *Proceedings of the First International Conference on Entity-Relationship Approach*, pp. 103-119, December 10-12, 1979.

Elmasri, R., and Wiederhold, G., "GORDAS: a Formal High-Level Query Language for the Entity-Relationship Model," *Proceedings of the Second International Conference on Entity-Relationship Approach*, pp. 49-70, October 12-14, 1981.

Elmasri, R., and Navathe, S., *Fundamentals of Database Systems*, pp. 23-36, pp. 409-452, Benjamin-Cummings, 1989.

Gorman, K., and Choobineh, J., "An Overview of the Object-Oriented Entity-Relationship Model (OOERM)," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, vol. 3, pp. 336-345, 1990.

Hughes, J., *Object-Oriented Databases*, pp. 79-119, Prentice-Hall, 1991.

Kappel, G., and Schrefl, M., "A Behavior Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases," *Proceedings of the Seventh International Conference on Entity-Relationship Approach*, pp. 311-328, November 16-18, 1988.

Kemper, A., and Moerkotte, G., *Object-Oriented Database Management: Applications in Engineering and Computer Science*, pp. 349-376, Prentice-Hall, 1994.

Kim, W., Bertino, E., and Garza, J., "Composite Objects Revisited," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 337-347, May/June 1989.

Lazimy, R., "E^2R Model and Object-Oriented Representation for Data Management, Process Modeling, and Decision Support," *Proceedings of the Eighth International Conference on Entity-Relationship Approach*, pp. 129-151, October 18-20, 1989.

National Air Intelligence Center, *Electronic Warfare Integrated Reprogramming Database (EWIRDB) Guide*, vol. 2, pp. 1.1-7.3, April 1994.

Navathe, S., and Pillalamarri, M., "OOER: Toward Making the E-R Approach Object-Oriented," *Proceedings of the Seventh International Conference on Entity-Relationship Approach*, pp. 185-206, November 16-18, 1988.

Rosen, K., *Discrete Mathematics and Its Applications*, pp. 340-407, McGraw-Hill, 1991.

Scheuermann, P., Schiffner, G., and Weber, H., "Abstraction Capabilities and Invariant Properties Modeling Within the Entity-Relationship Approach," *Proceedings of the First International Conference on Entity-Relationship Approach*, pp. 121-140, December 10-12, 1979.

Smith, J., and Smith, D., "Database Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems*, v. 2, pp. 105-133, June 1977.

Teory, T., Yang, D., and Fry, J., "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *Computing Surveys*, v. 18, pp. 197-222, June 1986.

INITIAL DISTRIBUTION LIST

	Number of Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 052 Naval Postgraduate School Monterey, California 93943-5101	2
3. Computer Technology, Code 32 Naval Postgraduate School Monterey, California 93943-5120	1
4. C. Thomas Wu, Code CS/Wq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5118	1
5. Craig W. Rasmussen, Code MA/Ra Department of Mathematics Naval Postgraduate School Monterey, California 93943-5216	1
6. Ted Lewis, Code CS/Lt Department of Computer Science Naval Postgraduate School Monterey, California 93943-5118	1
7. LT Gerald B. Barnes 25668 Capshaw Road Athens, Alabama 35611	2