Comparative Analysis of RDBMS and OODBMS: A Case Study

M.A. Ketabchi

Department of Electrical Engineering and Computer Science
Santa Clara University
Santa Clara, CA 95053

S. Mathur

Data Management Systems Division
Hewlett-Packard Company
19447 Pruneridge Ave. Bldg.47L
Cupertino, CA 95014

T. Risch

Database Technology Department/Stanford Science Center
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304

J. Chen

Corporate Product Generation Process Department
Hewlett-Packard Company
3155 Porter Drive, Bldg. 28AD
Palo Alto, CA 94303

Abstract

The underlying hypothesis of the case study described in this article is that the development, implementation, operations, and maintenance of large complex data intensive applications such as computer integrated manufacturing can be simplified through the use of object-oriented DBMS. The objective of the case study is to verify this hypothesis.

The approach of the case study is to prototype selected representative components of a computer integrated manufacturing system, which have been developed on top of a relational DBMS, using an object-oriented DBMS.

The results of the study illustrate that the object-oriented prototype has a superior schema, is capable of providing convenient access to information, and is easier to extend and maintain.

1. Introduction

Database Management Systems (DBMS) are software systems which provide facilities to model, manipulate, and maintain large amount of interrelated data which are shared by many users over a long period of time. A DBMS is an implementation of a data model, and a set of facilities to help maintain the database. The data model [TSIC82] defines the set of structures which are utilized to model the information, a set of operations which are utilized to manipulate these structures, and a set of constraints which define the consistent states of the data. As an example, the relational model [CODD70, DATE85] supports relations which are sets of tuples with fixed number of primitive data elements. The set of operations in the relational model includes operations on the sets of tuples, and two operations of *select* and *project*. Select operation allows those tuples which satisfy given conditions to be selected. Project operation allows desired elements of all the tuples in a set to be selected. Select cuts a relation horizontally and project cuts it vertically. An important constraint enforced by most relational DBMS is that each tuple in a relation must be uniquely identifiable.

The facilities provided by DBMS include:

1. Persistency - DBMS users can create data items which reside in persistent storage, rather than volatile memory, without creating files which contain the data and are referenced by their names.
2. Concurrency - Multiple users can access the same database simultaneously without creating inconsistent results due to the interactions of concurrent operations.
3. Transaction management - DBMS users can request that a sequence of database operations be treated as a single indivisible unit. The unit will either be executed completely or will not execute at all. The database will be consistent before the operation begins and after it is completed.
4. Recovery - Data will not become inconsistent or will not get lost as a result of system failures. If failure occurs in the middle of a transaction then the changes made by the transaction will be undone and the database will return to its state before the transaction begin.
5. Query language - DBMS provide query languages which are high-level, easy to use interfaces for accessing information.
6. Performance - DBMS provide efficient access to large amount of persistent multi-user data.
7. Security - Facilities are provided to protect the information against unauthorized accesses.

The set of structures, operations, and constraints in relational data model is limited and fixed. Consequently, any structure and operation needed by the application should be mapped by the application into this set. As the applications become complex this mapping becomes complex and involves large amount of application code. Moreover, to retrieve an application specific object, several DBMS operations must be performed. Large application code makes their development and maintenance difficult and expensive. The need to perform multiple database operations to retrieve an application specific entity makes applications slow. OODBMS were developed to solve these problems.

An OODBMS is a DBMS and as such provides the facilities listed above. It is also an object-oriented system [GOLD83] and as such supports the following capabilities:

1. Object identity - System defines and maintains unique identifiers for objects which represent entities. This allows equal objects (objects which have the same attributes and equal attribute values) to coexist. It frees users from the need to define unique keys for entity instances.
2. Active data - Database may contain information which are defined procedurally.
3. Classification - Similar entity instances are classified into types or classes. A type defines the properties and operations which are available to manipulate its instances. The relationships between entity types and instances are known to the system and can be utilized to formulate queries which span data and schema.

4. Generalization - Similar entity types can be generalized into supertypes which capture their similarities. Existing types can be refined to create subtypes which inherit the properties and operations of their supertypes and may have their own specific properties and operations.
5. Encapsulation - Objects are manipulated by operations which are provided by their types. The implementation of these operations may change without invalidating their use.
6. Composition - Objects may be assemblies of other object. In other words, objects are not limited to primitive domains.
7. Extensibility - The set of operations, structures, and constraints available to applications is not limited and fixed. Applications can define new operations and structures which are used the same way as the built-in structures and operations.

Relational DBMS are well into commercial production environment while OODBMS are in their infancy. However, there exists a significant amount of interest in OODBMS which were developed in response to the deficiencies of RDBMS and the requirements of new applications such as CIM (Computer Integrated Manufacturing), CASE (Computer Aided Software Engineering), and GIS (Geographic Information Systems). To understand the differences between RDBMS and OODBMS, and to gain experience in using OODBMS, we began a project for prototyping an existing CIM (Computer Integrated Manufacturing) application developed using a RDBMS. Iris [FISH88], an OODBMS developed in Database Technology Department of Hewlett-Packard was used to implement the prototype.

This paper gives an overview of the project and its results. Section 2 gives an overview of Iris and the application system. Section 3 is a discussion of information modeling using RDBMS and OODBMS. Section 4 compares the user interfaces of the RDBMS and OODBMS. Section 5 describes the application of a new functionality supported in Iris. Section 6 provides concluding remarks.

2.1 Overview of Iris Object-Oriented Database Management System

The fundamental elements of the data model of Iris are objects, types, and functions [FISH88, SHIP81]. An Iris object is represented by a unique identifier. Each object is associated with at least one type. This association supports classification. An object is said to be the instance of the types with which it has classification associations.

Types participate in the definition of functions, as their input and output specifications. A function can be applied to the instances of its input types. Application of a function results in objects which are the instances of its output type. An Iris schema consists of the definitions of types and functions. Definition of a function consists of its specification and its implementation. The specification of a function provides name of the function, types of inputs to function, types of the outputs of the function. Implementation of a function may follow directly after its specification or may be given later. If a function is implemented by a stored table of inputs/outputs tuples, it is called a stored function. If a function is implemented by a query written in OSQL (Object SQL which is the high-level data definition and manipulation language of Iris), it is called a derived function. If a function is implemented by a link to an executable code which may be written in a programming language such as C, it is called a foreign function. In the following example the first and the second functions are implemented by a stored table, the third function is implemented by a query, and the fourth function is implemented by a link to a routine called "avg_r".

```
create function name (Person p) -> Charstring n;

create function enrolled (Student s,  Course c);

create function grandParent (Person p) -> Person gp as
    select gp for each person gp
    where gp = parent(parent(p));

create function average (bag of Real) -> Real as
    link 'arg_r';
```

Iris functions model attributes, relationships, and operations. For instance, function *name* defined above models the *name* attribute of type *Person*, function *enrolled* models a relationship between types *Student* and *Course*, and function *average* models an operation on *bag of Reals*.

2.2 Overview of the Application System

The application system is a Computer Integrated Manufacturing application system which emphasizes the link between design and manufacturing. It attempts to link the design and development phases to manufacturing by providing an information centered integration of applications which are used to carry out different tasks. We prototyped two parts of the application system using Iris: Component Information Center (CIC) and Electronic File Cabinet (EFC). CIC provides functions for the acquisition, review and classification of purchased and fabricated parts. It couples component search and selection with computer aided enginering design tools. Its objective is to allow automatic optimal parts selection based upon the characteristics of design, and the preferred parts classification determined by the manufacturing site.

The functionality of the CIC includes:

1. Enabling designers to perform online component search and selection based upon the attributes of a component
2. Allowing users to enter or edit attributes of components for either corporate or divisional information.
3. Providing authorized users with ability to create and execute rules for classifying preferred parts based on the technology, reliability, and vendor performance.
4. Ensuring the availability of the data which are required in various manufacturing processes.
5. Providing form-oriented user interfaces for viewing and entering information.

The current CIC which has been implemented using a commercial relational DBMS has approximately 550 MB of component related data. 120 MB of these data are independent of the category of component. This portion includes information such as *contracts*, *suppliers*, and *drawings*. The remaining portion of the data depends on the category of components. This portion includes information such as *logic-size* for *Digital* and *density* for *Memory*.

EFC application attempts to improve the communication of design and manufacturing data by providing a means of electronic storage and on-line access to manufacturing process and product data and their relationships. It provides an integration framework for programs and translators, which are normally used as stand-alone tools for communicating data among different processes.

The functionality of the EFC includes:

1. Managing internal documents (documents which are stored in EFC), and the relationships among the internal and external documents (documents which are known to systems but are stored off-line).
2. Managing extended material list and the description of hierarchical structures of products.
3. Providing facilities for revision management for products, parts, and assemblies.
4. Supporting data or demand driven start of transfer and/or translation processes.
5. Supporting event-triggered distribution of information by mail to predefined distribution lists.

The current prototype of EFC has been developed on top of a commercial relational DBMS.

In the following sections we describe and compare the schema and interface of the relational and Iris implementation of the application system.

3. Information Modeling

This section describes and compares the relational and Iris schemas of the CIC and EFC. It describes the observations we made in the course of the project, and identifies the features of OODBMS which we found important and useful.

3.1 Component Information Center

The Component Information Center consists of two kinds of data. One kind of data describes information which is applicable to all parts independent of their categories. Examples of this kind of data, referred to as non-parametric data, are *contract*, *usage*, and *failure*. The second kind of data describes category dependent properties of components. Some of these data, such as *part_number* and *Manufacturer* are applicable to the most general category of components called *Part*. Because all components are *Parts* they have *Part_number* and *Manufacturer*. Subcategories of *Part* have information which are specific to them. Examples of data which are subcategory specific are *technology* for *Digital* and *Memory*, *logic_size* for *Digital*, *access_time* for *Memory*, and *temp_coef* for *Fixed Resistors*. We refer to this kind of data as parametric data.

3.1.1 Relational Schema for Component Information Center

The CIC in it's relational implementation consists of approximately 28 relations which contain non-parametric information, and 160 relations which contain parametric information. These relations form an unstructured set, because there exists no apparent relationship among them in the schema.

There are 32 categories of parts with 4 to 8 subcategories within each category. There is a relation for each subcategory. Each subcategory relation has approximately 16 attributes, therefore, there are more than 2,500 parametric attributes in the schema. 40% of the attributes among the tables within each subcategory are common. Intuitively, this is because these relations describe parts which belong to the same category. Examples of tables representing subcategories are shown in figure 1. The first two tales, *Digital* and *Memory* belong to category *IC*, and the second two tables, *Fixed* and *Var_sgl* belong to category *Resistor*. As can be seen, the first 6 attributes in the first two relations, and the first 6 attributes in the second two relations are the same. Moreover, the first three attributes among all four relations are the same.

Another cause of repetition in the relational schema are compound keys. Compound keys in certain parts of schema consists of several attributes and appear in many relations. For instance, there are compound keys with up to 7 attributes which appear in more than 10 different relations. These keys create redundancy and increase the complexity of the database, its use, and maintenance.

3.1.2 Iris Schema for Component Information Center

Iris schema for CIC is hierarchical. The root of the hierarchy is a system defined type called *UserTypeObjects*. The nodes of the hierarchy are Iris types which define the entity types. Attributes of these entity types are defined as Iris functions. The edges in the hierarchy represent subtype/supertype (specialization/generalization) relationships which support inheritance.

Generalization and the related inheritance allowed us to impose structure onto the set of entity types in the database, capture the semantics of categorization of components, and at the same time eliminate multiple occurrences of the same attributes, and therefore reduce the size of the schema considerably.

The Iris schema for the portion of the database in figure 1, is shown in figure 2. Connections among nodes of the hierarchy in figure 2 represent generalization relationship. *Part*, the root of the hierarchy is the most general entity type because all components are *Parts*. The nodes at the second level of the hierarchy are categories of parts, and the nodes at the third level are subcategories. Subcategories are the most specific types of parts. Because all components are parts, they inherit the attributes defined in type *Part*. An Instance of a subcategory (for example, an instance of *Memory*) is an instance of is category (for example, an instance of *IC*) as well, therefore, *Memory* inherits the attributes defined in type *IC*. In the relational schema there is no *IC* and *Resistor* and the fact that an instance of *Memory* is an *IC* is not stored in the database. Because of the structure and additional semantics, the schema in figure 2 contains more information than the schema in figure 1, and it is easier to understand.

Inheritance along the generalization relationship allowed us to reduce the size of the schema, which deals with the parametric data, by more than 35%. We factorized the common attributes among subcategories and defined them once in the categories. This reduced the number of attributes defined at subcategory level by almost 50%. We then factorized the common attributes of the categories and defined them as the attributes of the *Part*. Note that this reduction does not affect the size of the database. It reduces the size of the schema only.

We reduced the size of the schema further by replacing compound keys with object identifiers which are defined and maintained by Iris for each entity instance. As we mentioned, in the relational schema, there are compound keys which have up to 7 attributes and appear in more than 10 relations. By replacing these compound keys with object identifiers the size of the schema was reduced. Moreover, the readability and the semantic contents of the schema were enhanced. For instance, there is a compound key *part_number* + *part_revision* + *manufacturer_name* + *manufacturer_address* which appears in any relation which references a part. In the relational schema all the elements of the compound key appears for each of these references without any syntactic indication that they, together, reference a part. In the Iris schema they are replaced by a single attribute whose type is declared to be *Part*.

Replacing multiple occurrences of compound keys reduces the size of the database as well, but this reduction may be offset by increased space utilization due to system-defined object identifiers.

The notion of user defined type enhances the semantic contents of the schema because they are used to specify the types of the attributes which are references to other objects. For instance, assume parts have an attribute whose values are the organizations which are responsible for their designs. We can then define the *design_org* attribute of parts to be of type *Organization*. In the relational version, the type of the attribute *design_org* will be *Char(30)*, or a similar primitive type (RDBMS which support referential integrity may simplify this problem). The fact that *design_org* of *Part* are *Organizations* is embedded in the values of *Char(30)*.

The ability to define functions which return a list of values (an aggregate value) further enhanced the semantic contents of the schema. Assume we need to define height, length, and width of the packages of parts. In the relational schema these attributes will be defined independently or they will be defined as the attributes of another relation. In the first case the fact that they all belong together is lost, and in the second case a new user-defined key attribute or a compound key which consists of four elements has to be defined. In Iris we defined package_dimension function shown below, which takes a *Part* and returns a list of four values.

```
create function package_dimension (Part p) ->
        <Real height, Real length, Real width, Charstring unit>;
```

To retrieve dimensions of the package of a part P1 one needs to execute the expression: *package_dimension (P1)*, which returns a list of four elements.

3.1.3 Comparison of Relational and Iris Schemas

We believe Iris CIC schema is superior to relational CIC schema:

1. It contains more semantics because it captures generalization, supports user-defined types, and supports functions (which model attributes) which return aggregate values.
2. It is smaller because of inheritance and object identifiers.
3. It is easier to understand because it is better structured, and has a simple mapping to the logical view of data.

3.2 Electronic File Cabinet

Electronic File Cabinet (EFC) is a databased application which manages R&D and Manufacturing related documents. It addresses the need of getting the right revision of the right data to the right place at the right time. EFC was a part of a set of CIM based applications, we set out to prototype using Iris.

The original application was designed for a relational database system. The corresponding Entity Relationship (ER) schema is shown in the figure 3. The schema designers had run into problems often associated with traditional ER based schema design and had come up with Object-Oriented solutions to get around some of these problems. In this section we describe some of the schema design issues that came up during the redesign of EFC schema using Iris.

To simplify the comparison between the original schema and the Iris solution, we decided to retain the design of the original schema as much as possible. If we had started designing the Iris schema from the scratch, the final Iris schema would have possibly appeared much different from the current version.

3.2.1 EFC Schema.

The figure 3 shows part of the EFC schema which we implemented in Iris. The main entities in the schema are - *User*, *Process*, *Assembly*, *Part*, *Operation* and *Document*. Since most of these main entities participate in similar relationships with other entities, a new entity called *EFC Object* was created. All of these main entities are related to *EFC Object* via the relationship *object_ref*, and the *EFC Object* entity can play the role of any of the specific main entities. This way all similar relationships on the main entities could be defined on the entity *EFC Object* rather than defining separate relationships on each of the main entities. For example most of the main entities have a relationship with the entity *Document* called *object_desc*. Rather than defining it repeatedly for the entities *User*, *Process*, *Assembly*, *Part* and *Document*, this relationship needs to be defined once for the entity *EFC Object*. If a particular *Part* p1 has the relationship *object_desc* with a *Document* d1, we first find the *EFC Object* eo1 which acts as a surrogate for p1 via the relationship *object_ref* and then relate eo1 to the *Document* d1.

Let us look at some of the main entities mentioned above. The entity *User* refers to the users of the documents managed by the EFC. *Part* refers to the actual physical parts (like ICs and resistors) which are used in the manufacturing processes. *Document* refers to any on-line documentation maintained by the system. *Assembly* refers to aggregate structures which could be composed of individual documents, parts or other sub-assemblies. *Process* refers to the processes understood by EFC. Processes take assemblies as inputs and create (output) other assemblies. Some of the processes (like text formatting, compilation) could actually be invoked from within EFC, and information is maintained for their execution. For other processes, which cannot be invoked via EFC, only auxiliary information is maintained. EFC also maintains a list of operations which are applicable on assemblies, and keeps track of the users who could invoke these operations.

Certain users who behave in a similar way, can be grouped in user groups called *User Group*. Similarly, processes, assemblies, parts and documents which behave in the same way, are also grouped together in groups called *Process Group*, *Assembly Group*, *Part Group* and *Document Group* respectively. Together we refer to these entities as *EFC Group*. The relationship *group_object* keeps track of which *EFC Objects* belong to which *EFC Groups*. Using groups we can describe information about more than one *EFC Objects* at a time. So if a particular document d1 describes a number of parts, we can group all these parts in one group and then use the relationship *group_desc* to relate the document d1 to this group. This will relate the document d1 to all of the parts in the part group.

Users can put locks on other assemblies to prevent certain operations on these objects. EFC also stores information regarding the projects users work on, and the *EFC Objects* which are relevant for a particular project. EFC also does primitive revision management. It keeps track as to which assemblies are actually revisions of older assemblies, and who is responsible for maintaining these revisions.

3.2.2 Iris Schema for EFC

As mentioned earlier, we had chosen a path in which we wanted to map the original EFC ER schema into Iris schema with minimum possible transformation. Therefore, most of the entities in the ER schema are mapped directly to Iris types and most of the relations were mapped into Iris functions. The Iris schema is shown in figure 4.

The entity *EFC Object* was mapped into an Iris type called *EFC_Object*. *EFC Objects* - *User*, *Process*, *Assembly*, *Operation* and *Document* are subtypes of *EFC_Object*. All relationships common to all *EFC Objects* are defined as functions on the type *EFC_Object*. Since Iris provides inheritance, all of these functions defined on *EFC_Object* are also inherited by its subtypes and the semantics of the original EFC schema are maintained.

Since all documents and all parts are also considered to be assemblies (a part could be considered to be a assembly composed of that part and no other substructure), the type *Document* and *Part* are subtypes of type *Assembly*. A hierarchy parallel to that under *EFC_Object* is formed under the type *EFC_Group*, which keeps track of all EFC groups maintained by the system. The only difference in the hierarchies is that there is no *Operation_group* because the operations known to EFC are not grouped together. For each subtype of *EFC_Object* there is a function called *belongs_to* which returns the corresponding subtype of *EFC_Group* and which relates an *EFC_Object* to the group it belongs to.

Processes generate assemblies from other assemblies. The input assemblies might be simple assemblies like parts or individual documents, or could be complex assemblies themselves. To store this information, we define two functions - *input* and *output* on the type *Process*. The function *input* relates a *Process* to all the assemblies it takes as inputs and the function *output* relates a process to all assemblies the process generates. Since this is useful information for assemblies themselves, we also define a derived Iris function called *generated_from* on assemblies. The function *generated_from* takes an assembly as input and returns all the assemblies it is generated from. This information is not actually stored in this function but is derived from the *input* and *output* functions of *Processes*. This is possible in Iris because any query (in this case it would be a query which determines the process which generates the assembly and then reports all the input assemblies to this process) could be modeled as a derived function and so duplicate information need not be stored.

We created a type called *Project* which maintains information about projects known to EFC. The functions *works_on* relates users to the projects they work on, the function *responsible_for* relates the users to the projects they are responsible for. The function *relevant_for* keeps track of all the EFC objects relevant for a specific project. Similarly the functions *ownership* and *responsibility* relate documents to the users who own and are responsible for the document respectively. The type *Lock* keeps track of the locks placed by users on the assemblies to prevent invocation of certain operations.

What we have described above are some of the main types and functions in the Iris schema. Besides these, a few types and a number of other functions were defined which keep auxiliary information about the main entities. In Iris, we model attributes as functions and each of the main types have a number of functions defined on them which play the role of attributes. Since they do not illustrate any major point, we have not mentioned them here and neither do we show them in the figure 4.

3.2.3 Design Issues in EFC ER and Iris schema.

Let us look at some of the differences in the ER and Iris schemas for EFC which illustrate some of the design issues involved. As you can observe from figures 3 and 4, there is no relationship *object_ref* in the Iris schema. This relationship is used in the ER schema to relate main entities (*User*, *Process*, *Assembly*, *Document*, *Part* and *Operation*) to the generic entity *EFC Object*. For each main entity instance, there is a corresponding *EFC Object* instance related via the relationship *object_ref*. This *EFC Object* instance acts as a surrogate for the corresponding main entity instance so that the relationships which were to be defined on the main entity instances, are now defined over the surrogates. Besides this problem of indirection, the ER schema suffers from the problem of maintaining a unique surrogate for each corresponding main entity instance. To keep track of this correspondence, the ER schema had similar attributes defined over all of the main entities, and these attributes acted as unique keys. The actual maintenance of this correspondence itself, was the job of the applications written on top of the ER schema.

In Iris both of these problems are solved simply. The problem of indirection is solved because Iris provides the mechanism of inheritance, and all relationships (functions) on the type *EFC_Object* are automatically inherited by it's subtypes. These functions are defined directly on objects of the subtypes like *User*, and we do not have to go through the intermediate step of first accessing an intermediate entity instance, as we have to do in the ER schema.

The second problem of maintaining regular, unique keys for the *EFC Objects* does not arise in Iris because the system automatically generates a unique identifier for all objects known to Iris. This unique identifier can be used to refer to an object in a regular fashion.

Another point to note is that the relationship *revision_of* in the ER schema has been modeled as a type in Iris. This relationship relates an old assembly to it's revision. The date on which this change was made and the information about the engineer who made this change is kept in attributes defined over the relationship. In Iris you cannot have user defined attributes over the functions. We had a choice of either modeling this relationship as a multi-argument function or as a type (this choice is made clear in section 3.3). We chose to define revision as a new type, which has an attribute *old_a* linking this revision to the old assembly; an attribute *new_a* linking this revision to the new assembly; and two other attributes - *date* and *project_engr* returning the date when this link was established and the name of the engineer responsible for this link.

Another advantage of using Iris is the ability to define derived information in Iris schema. Any query in Iris could be thought of as a function taking certain arguments and returning certain results. In Iris, one can define derived functions which take certain arguments, invoke the query which defines their function body based upon the value of the input arguments, and return the results of this query. No information is actually stored in these functions, but the information is obtained by accessing other pre-defined stored or derived functions. We see an example of this when we define the relationship *generated_from* on assemblies. The function *generated_from* uses the information stored in functions *input* and *output* to obtain the required information. In the corresponding ER schema, either this information would have to be stored redundantly or the designer would have to encode this derivation in the application.

A final point to note is that we have been comparing the ER schema of EFC with the Iris Schema. Actually the information is not stored in the ER schema, but a transformation to the relational tables has to be carried out. This transformation is usually a non-trivial task. Since the relational database designers have to store all entities and relationships as flat tables, a part of the information is lost in the process of transformation. The designers have to maintain the correct semantics (logical view) at the application level. They also have to take care of issues like normalization and referential integrity. None of these issues come up as a major problem in designing Iris schema. This makes designing Iris schema a much simpler task.

3.3 Mapping ER Schema into Iris schema.

Our experiences with the CIC and EFC projects suggest that it is very simple to map an ER schema into an Iris schema. However, the reverse transformation is nontrivial because Iris provides much more semantics than ER schema. Some of the constructs modeled in Iris need complex transformation before they can be modeled in ER schema.

The two basic constructs in ER model are - entities and relationships. Some people consider attributes to be different from relationships, but we will consider them together in the following discussion.

Any ER entity could be modeled as an Iris type. The entity instances would then correspond to Iris objects. However, some entities could also be modeled as Iris functions. Most of these would fall into the category of entities which are created to avoid many-to-many relationships. Even though ER model allows many-to-many relationships, designers sometimes introduce an artificial entity corresponding to the many-to-many relationship because the underlying physical database may have difficulty in managing many-to-many relationships. A typical example would be the entity *Enrollment*, which might be created to avoid the many-to-many relationship between *Courses* and *Students* (figure 5). Such entities could easily be modeled by Iris functions which can take multiple arguments and return multiple results. The Iris model allows and manages such many-to-many relationships without any difficulty. The advantage of such an Iris schema is that it models the real world semantics of a relationship, rather than introducing artificial entities.

The relationships in ER model can easily be modeled as functions in Iris. One can restrict participation of Iris function argument and result parameters to obtain one-to-one, one-to-many, many-to-one or many-to-many relationships.

ER model relationships can also be modeled as Iris types. A good heuristic to determine whether a relationship should be modeled as an Iris function or an Iris type is as following. If the relationship is interesting in itself, ie. we are interested in manipulating information about the relationship itself, we should map this relationship into an Iris type. However, if the relationship is only interesting because it relates various entities, we should model such a relationship by an Iris function.

Let us again consider the example of enrollment of students to illustrate this point. If we only use the relationship *enrollment* to establish the connection between students and classes, then we should model this relationship as an Iris function. This function may take a student and a class as arguments and return true if and only if the student is enrolled in the class. However, if we also want to manipulate information about the *enrollment* itself - like counting the total number of *enrollments* - we should model *enrollment* as an Iris type. This type would have two attributes - *student* and *class* - which will refer to the student and to the class in which the student is enrolled.

Another reason for modeling *enrollment* as an Iris type is - we might be interested in adding more information about the relationship at a later time. For example, if later on we decide to associate a grade received by the student with each enrollment, it will be trivial to add a new attribute *grade* to the Iris type *Enrollment* and assign it's value for each enrollment. However, if we had mapped *enrollment* to an Iris function,

we would have to delete this Iris function, and in the process destroy all the relationships it kept track of. We would then have to create a new function which might take three arguments - students, classes and grades - and reestablish the link between students, classes and grades. As you can observe, the second alternative (*enrollment* as an Iris function) leads to more work in this particular case.

Another example where we model an ER relationship into an Iris type is when we map the relationship *revision_of* in EFC ER schema to a type *Revision* in Iris schema. We made this decision because we felt that revision was an interesting relationship by itself and we would like to keep information about the date on which this link between an old assembly and a new assembly was established, and about the engineer responsible for maintaining this link.

In general it is very easy to transform ER schemas into Iris schema. Unfortunately the reverse transformation is difficult. Iris has additional semantics - like inheritance and ability to define derived functions - which are difficult to model using ER schema.

4. Interface

The relational DBMS used in the implementation of the application has SQL interface and allows end-users to interact with the system through a form-oriented interface generated through 4GL facilities of the system.

Iris has OSQL (Object SQL) interface which is SQL-like, but takes advantage of the object-oriented and functional features of Iris to simplify queries. Iris also has a graphic browser/editor which allows users to view, browse, and edit Iris databases (both schema and data) in X-Window environment.

In the following sections we comment on the advantages of the OSQL vs. SQL, and Iris's Graphic Browser/Editor. vs. the form-oriented interface of the relational system.

4.1 OSQL Vs. SQL

We found two facilities of OSQL very useful: function composition and derived functions. Function composition simplifies queries by eliminating joins in most queries. The following example illustrates the substitution of join with function composition.

Assume different *Engineering_Divisions* are responsible for different *Parts*. Also assume *Parts* have *part_number*, and *Engineering_Divisions* have *name*. An SQL query to retrieve *names* of the *Engineering_Divisions* who are responsible for a given part with *part_number* myPart may be formulated as follows:

```
select name
from Part, Engineering_Divisions
where (Part.part_number = Engineering_Divisions.part_number)
      and (part_number = myPart)
```

The equivalent OSQL query, assuming that myPart is the object identifier of the desired part can be formulated as follows:

```
select name(Engineering_Divisions(myPart);
```

Derived functions provide an abstraction of frequently occurring queries. Complex queries can be simplified by defining frequent subqueries as derived functions. The following example illustrates this point:

Assume we need to find the *names* or *addresses* or *phone_numbers* of the *Manufacturers* of *IC* 's whose *failure_rate* are less that x. We can define the selection of the manufacturer as a derived function and then use it to retrieve different attributes of the manufacturer. The definition of the function is shown below:

```
create function better_manufacturers (Integer x) -> Manufacturer as
    select m
    for each IC i , Manufacturer m
    where manufacturer (i) = m
        and failure_rate (i) < x ;
```

Having defined this function it can be used in other queries such as the following, which retrieve the *names* and *addresses* of the manufacturers of *ICs* with failure rate less than 5:

```
select name ( better_manufacturer ( 5 ));

select address ( better_manufacturer ( 5 ));
```

4.2 Object-Oriented Graphic Browser Vs. Form-Oriented Interface

The end-user interface of the current implementation is based on a relational 4GL. It consists of a set of forms which are displayed one at a time. The typical sequence of interactions with the system is as follows: a form is displayed, the user makes a selection, another form is displayed and the previous form disappears. After a few steps a form is on the screen, but it is not readily obvious how we got there. In other words the context of information which is on display is not available. Another problem is that the interaction with the system always starts from a fixed initial step and it usually takes several steps to get to the desired information.

The graphic browser of Iris displays the type hierarchy of the schema. Users can select any type they want with the first selection by mouse click. Depending on what object type has been selected a menu of meaningful operations appears which allows users to select an operation which is guaranteed to be meaningful when applied to that object. Users can browse both schema and data by selecting an object and then selecting an operation from the menu of the valid operations on that object. At any point during interaction, the user has the context of the data that is being consulted. Normally it takes fewer interactions to get to the desired information because hierarchy of all the types in the schema is available to user throughout the interaction with the system.

5. Monitors

The application system keeps track of manufactured products and parts and their producing departments. The database contains data about failures of parts and departments which are responsible for servicing those parts. When parts fail, engineers update the current failure rate of failing parts.

It would be desirable to to be able to monitor the failure rates of parts. For example, a service department may want to know if a failure rate of some part for which the department is responsible gets too high. We call such a program a *failure rate monitor* .

The failure rate monitor is an example of a new kind of database applications where the user is informed when critical situations occur over the global state of a database. Iris has been extended with the capability to handle such *database monitors*. We demonstrated the use of database monitors in the context of the our CIM application system.

Note that database monitors actively notify the user when interesting situations arise. To achieve the same result in a conventional DBMS the user would have to regularly query the database for every situation that is of interest.

A short description of the database schema, failure rate monitor from end-user point of view, and the implementation of the failure rate monitor follow.

5.1 Database Schema

The database schema used in the implementation of the failure rate monitor is shown below. As before, to simplify the discussion we use a small subset of the application system schema.

```
/* Entities as Iris types */
create type Department(
    Number          Charstring,
    Name            Charstring);
create type Product(
    Number          Integer,
    Name            Charstring);
create type Failure(
    FailureRate     Real);
create type Part(
    Number          Charstring unique,
    Name            Charstring);

/* Relationships */
create function FailureProduct(Failure) -> Product;
create function ResponsibleDepartment(Part) -> Department;
create function PartFailure(Part) -> Failure;
```

In the failure rate monitor we display the failing parts, products, and failure rate for a given responsible department whose failure rates are larger than a given threshold. This information can be retrieved by the following derived Iris function:

```
/* Significant failed parts and products for responsible department */
create function FailureReport(Department d, Integer th) ->
    <Charstring prn, Charstring pan, Integer ta> as
     select prn, pan, ta
       for each Failure f, Part pa, Product pr,
         Charstring prn, Charstring pan, Integer ta where
       ResponsibleEntity(pa) = d and
       PartFailure(pa) = f and
       FailureRate(f) = ta and
       th < ta and
       FailureProduct(f) = pr and
       prn = Name(pr) and
       pan = Name(pa);
```

We developed an application program to track critical failures relevant to a given department. The appearance of the program in an interactive window-based environment is illustrated in figure 6.

The failure rate monitor can inform the user about the parts which fail more than a certain threshold. The program allows users to enter department name, and parts which are serviced by those departments. The user also enters the critical failure rate threshold. Different users can enter different thresholds, depending on their responsibilities. The failure rate monitoring starts when the user clicks on 'Monitor ON'. The user can turn off monitoring at any time by clicking on 'Monitor OFF'.

5.2 Database Monitors

The database monitor feature used to implement our failure rate monitor was first described in [RISC89] The main idea behind this feature is as follows.

Assume that we have a multi-user Object Oriented DBMS, such as Iris, where we also have a declarative query language, such as OSQL. The database is continuously and concurrently updated by transactions.

At any point in time the database has a global state, *Si*. Transactions are functions, *T(Si)* -> *Sj*, that transform the database state from one state, *Si* into another state, *Sj* . A monitoring program, such as the failure rate monitor, is interested in tracking when the database gets into interesting states, i.e. when the database satisfies certain conditions. When the database moves into such a state, the database monitoring program is informed about the state changes.

In our database monitor feature, interesting database states are described by the result of any monitored database query. For example, in the failure rate monitor we are interested in monitoring the result of a query that retrieves the failing parts of a given department and a failure rate threshold, as defined by the derived function *FailureReport*. The failure rate monitor should get notified whenever the value of *FailureReport* is changed. Notice that *FailureReport* returns the empty set when there is no parts which fail beyond the threshold. In general one can always define the monitored query so that it returns some particular value.

We also need some mechanism to inform the application program that a state change has occurred for a monitored query. Therefore, the programmer does not only specify which query to monitor, but also a tracking procedure. A tracking procedure is an application program procedure that is called by the DBMS when a state change has occurred in the monitored query. Thus the tracking procedure is part of the application program and it is written in the programming language of the application program (e.g. C). The DBMS does not send any data to the tracking procedure when a state change has occurred. However, the tracking procedure can freely access the database to retrieve the current value of the monitored query.

Database monitors need to be dynamic, i.e. they are active only while a particular application program requires them to. The system provides primitives for fast activation and deactivation of database monitors.

5.3 Implementation of Application

With the database monitor feature of Iris the implementation of failure rate monitor becomes very simple. Basically we assign program variables to the input fields *Service Department* and *Failure Rate Threshold* The database monitor tracks the *FailureReport* query for the current service department and failure rate threshold. The tracking procedure retrieves the failure rate query and displays its result in the window. Finally, the 'Monitor ON' button activates the monitor and then changes its label to 'Monitor OFF'; another click deactivates the monitor.

The program was implemented using the X window system, version 11. The code is approximately 400 lines of C, of which about 80 lines are interface code to Iris and the rest is X user interface code. The failure rate reporting program has 164 lines of code.

6. Summary and Concluding Remarks

Based on the results and observations made in this project, it is clear that object-oriented DBMS such as Iris provides better information modeling facilities than relational DBMS do. We illustrated the following advantages of OODBMS Iris with respect to information modeling and in section 3 and with respect to interface in section 4:

1. The schema is easier to understand because it is structured, contains more of the semantic of the data, and is intuitive.

   Generalization imposes structure, user-defined types and operations capture more semantics, and simple mapping from application entities to user-defined types make it more intuitive.

2. The schema is smaller because of inheritance and object identifier.

   Inheritance allows common attributes to be factorized and stated once. System defined object identifiers allow compound keys, which can potentially become large, to be replaced by a fixed size, smaller, and uniform system defined object identifiers.

3. The object-oriented graphics interface is easier to use than form-oriented character-based interface. The object-oriented graphics interface of Iris allows users to get to the desired information with fewer actions (mouse clicks and menu selections). Moreover, it preserves the context of information on the display.

Another very important observation that we made in the course of our project is that an object-oriented DBMS such as Iris enhances logical independence (independence of applications from database implementation) of applications. This is mostly because of the encapsulation of data, and the abstraction of implementation from use through functions.

One of the problems that we encountered was the lack of methodologies and tools to support application development using OODBMS.

Two questions remain:

1. How does the size of the database of an application implemented by a relational DBMS compare with the size of the same application implemented using an object-oriented DBMS such as Iris?
2. How does the performance of the relational implementation compares to the performance of object-oriented implementation of the same application?

We are planning to address these questions by conducting a benchmark.

The plan is to unload the contents of some of the existing relations, reformat them, and then load them into Iris schema. Once the Iris schema is populated we will run queries which achieve the same task in both relational and object-oriented implementations. We will then compare and contrast the two performances.

We will also compare the sizes of the relational and object-oriented databases with the same information contents.

References:

[CHEN76] Chen, P.P., "The Entity-Relationship Model: Toward a Unified View of Data," ACM TODS, 1976.

[CODD70] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," CACM, June 1970.

[DATE85] Date, C.J., "An Introduction to Database Systems," Addison Wesley, 1985.

[FISH88] Fishman et. al., "Overview of The Iris DBMS,", Database Technology Department, Hewlett-Packard Laboratories, June 1, 1988.

[GOLD83] Goldberg A., Robson, D., "SmallTalk-80 The Language and its Implementation," Addison Wesley, 1983.

[RISC88] Risch, T., "Database Monitors," Proceedings of VLDB 1989.

[SHIP81] Shipman, D. "The Functional Data Model and the Data Language DAPLEX," ACM TODS, 1981.

[SMIT77] Smith, J.M. and Smith D.C.P., "Database Abstractions: Aggregation and Generalization," ACM TODS, June 1977.

[TSIC82] Tsichritzis, D.C., Lochovsky, F.H., "Data Models," Prentice-Hall, 1982.

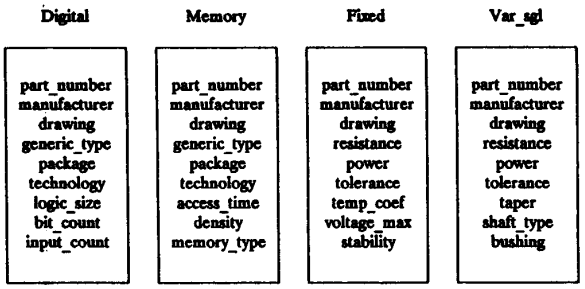

Figure 1
Subcategory Relations in the Relational Schema

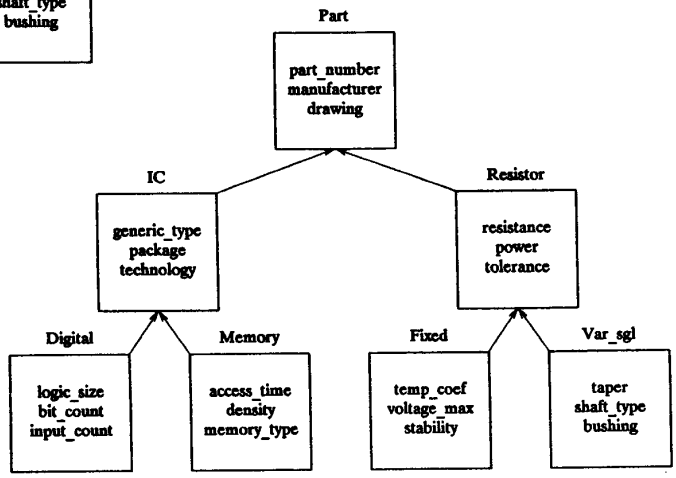

Figure 2
Part Category Hierarchy in the Iris Schema

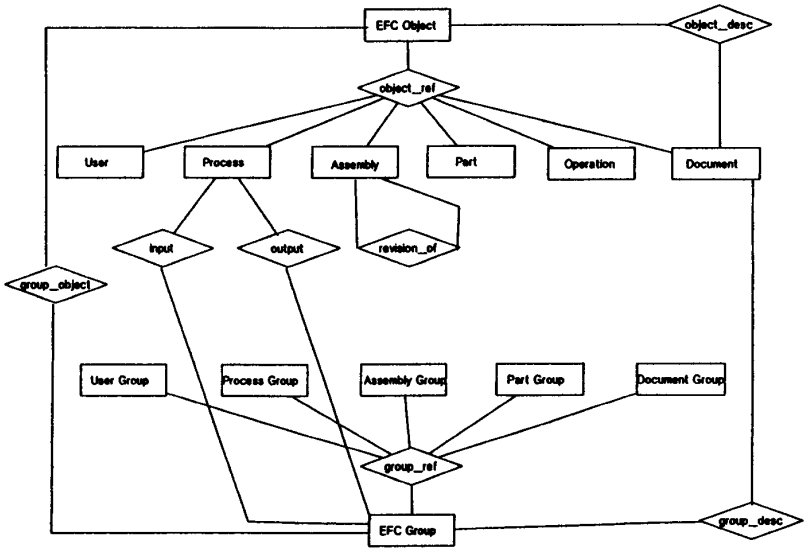

Figure 3
Partial Entity-Relationship Schema

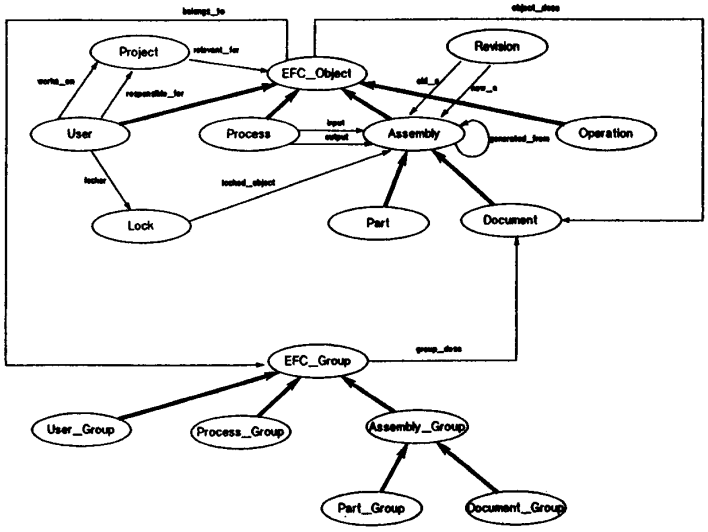

Figure 4
Partial Iris Schema

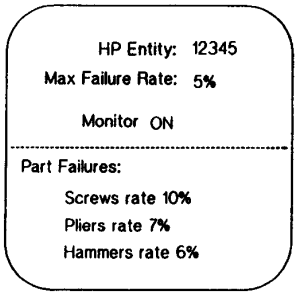

Figure 6
User Interface for Failure Rate Monitoring

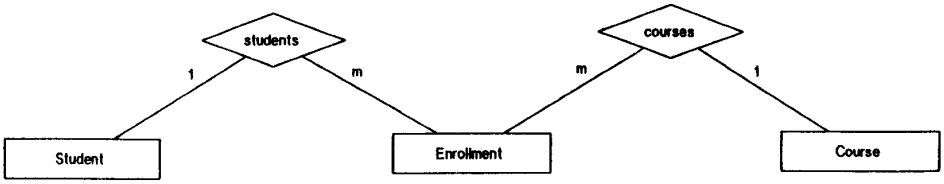

Figure 5
Converting Many-to_Many Relationships to One-to_Many Relationships