An Object-Oriented Approach to Database System Implementation

A. JAMES BARODY, JR.
Xerox Webster Research Center
and
DAVID J. DeWITT
University of Wisconsin-Madison

This paper examines object-oriented programming as an implementation technique for database systems. The object-oriented approach encapsulates the representations of database entities and relationships with the procedures that manipulate them. To achieve this, we first define abstractions of the modeling constructs of the data model that describe their common properties and behavior. Then we represent the entity types and relationship types in the conceptual schema and the internal schema by objects that are instances of these abstractions. The generic procedures (data manipulation routines) that comprise the user interface can now be implemented as calls to the procedures associated with these objects.

A generic procedure model of database implementation techniques is presented and discussed. Several current database system implementation techniques are illustrated as examples of this model, followed by a critical analysis of our implementation technique based on the use of objects. We demonstrate that the object-oriented approach has advantages of data independence, run-time efficiency due to eliminating access to system descriptors, and support for low-level views.

Key Words and Phrases: database systems, data manipulation routines, procedural binding, object-oriented programming, high-level languages, data independence
CR Categories: 4.22, 4.33, 4.34

1. INTRODUCTION

In database systems the principles of abstraction support two fundamental properties: data independence and a generic-procedure user interface. Data independence requires that user programs be isolated from the details concerning the underlying physical and logical structures used to implement the database. Data independence is achieved by presenting the user an abstraction of the database in which details of its actual implementation are hidden.

This research was supported in part by the National Science Foundation under Grant MCS78-01721 and in part by the U.S. Army under Contract DAAG 29-75-C-0024.
Authors' addresses: A. J. Baroody, Jr., Xerox Webster Research Center, Xerox Square-128, Rochester, NY 14534; D. J. DeWitt, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706.

In current approaches for database systems (relational [10], hierarchical [16], and network [8, 9]) the only access path to the entities and relationships contained in the database is through a user interface that consists of a set of generic procedures. These procedures operate on entities or relationships of any "type." For example, the relational operator "restrict" operates on any relation type (e.g., suppliers, parts).

An abstraction technique supported in modern programming languages (SIMULA67, MODULA, MESA, ADA, etc.) is object-oriented programming. This programming technique provides a strong isolation of users of a software system from its implementation. We believe that it is important to examine whether the concepts of modern programming languages are appropriate for supporting the generic procedure interface. This paper examines object-oriented programming as an implementation technique for database systems. Our interest in applying the object-oriented approach is based on the nature of the conceptual schema and the internal schema. These two schemata describe the attributes of database entities and the procedures that access and manipulate these representations. The conceptual schema defines the entities and the relationships between them, while the internal schema describes the internal data representation of entities, the physical and logical structure used to represent relationships, and the access paths that support database access.

The object-oriented approach encapsulates the representations of entities and relationships with the procedures that manipulate them. To achieve this, we first define abstractions of the modeling constructs of the data model that describe their common properties and behavior. Then we represent the entity types and relationship types in the conceptual schema by objects that are instances of these abstractions.

The generic procedures that comprise the user interface can now be implemented as calls to the procedures associated with these objects. We demonstrate that this approach can guarantee that the only access path to the database is through the generic procedures. In addition, since all internal representations are encapsulated by these objects, these representations are hidden from the generic procedures and therefore from the user. This satisfies the goal of data independence.

Several authors have reported on related research efforts. Stemple examined the automatic generation of data manipulation routines [26]. Yeh and Minsky have examined multilevel implementation techniques [17, 32]. The possible use of abstract data types (and the related concepts of modules) as a technique for the well-structured implementation of multilevel database system organizations (such as the ANSI/SPARC three-level organization) has also been suggested by Weber [30]. This proposal is sketchy and does not address such issues as elimination of run-time schema interpretation, data independence, or the need for multiple implementations of a single abstraction. Another application of abstract data types and modules in database systems is their use in either the development of user interfaces that are tailored to a specific application [6, 15, 17] or as structuring tools in languages designed to facilitate database access [13, 21, 23, 29, 30].

In Section 2, a generic procedure model of database implementation techniques is presented and discussed. Several current database system implementation techniques are illustrated as examples of this model. Then, in Section 3, our implementation technique based on the use of objects is described.

Section 4 presents a critical analysis of our approach and compares the object-based approach with other implementation techniques. This analysis is done with respect to requirements for the database system to manage meta-information, support of low-level and high-level views, performance of the resulting system, frequency of binding, and concurrency control. We demonstrate that the object-oriented approach has advantages of data independence, run-time efficiency due to eliminating access to system descriptors, and support for low-level views.

2. A GENERIC PROCEDURE MODEL OF DATABASE SYSTEM ARCHITECTURES

The programming language concepts of types and generic procedures are directly related to database system features. The concept of *type* in programming languages is well understood. A type implies a set of values visible to the user, an underlying representation that is not visible to the user, and the manner in which primitive operations on the type are to be interpreted in terms of the type's representation.

As an example of the use of types within database systems, the entity types and relationship types that are visible to users are defined in the conceptual schema. Each entity declaration in the conceptual schema defines a new type in terms of a set of underlying constituent types. For example, the definition of a PART relation describes the attributes of parts in terms of underlying data types. We regard tuples within this relation as instances of type PART.

Work in programming languages has extended procedural abstractions to define a *generic procedure* as a procedure that performs the same basic operation on actual parameters of more than one type. The implementation of the operation will generally vary depending on the type of the actual parameter. The plus (+) operation in ALGOL 60 or PASCAL represents such a procedure since it accepts operands of types integer, real, complex, and so forth. Thus a generic procedure must know the primitive operations that are supported for each valid parameter type.

In all current approaches for database systems the only access path to the entities and relationships contained in the database is through a user interface that consists of a set of generic procedures, which we refer to as *data manipulation routines.* These procedures operate on entities or relationships of any "type" defined in the conceptual schema. For example, the relational operator "restrict" operates on any relation type. And the network data manipulation language verb "findnext" accepts parameters of any record and set types.

Each data manipulation routine uses descriptions of the entity types and relationship types from the schemata to determine the correct computation to be performed. The conceptual and internal schemata associate with an entity not only the specifications of data items, but also the specification of an access strategy. This access strategy specifies those procedures to be used to store and to retrieve entity instances from the database. For a given actual parameter (e.g.,

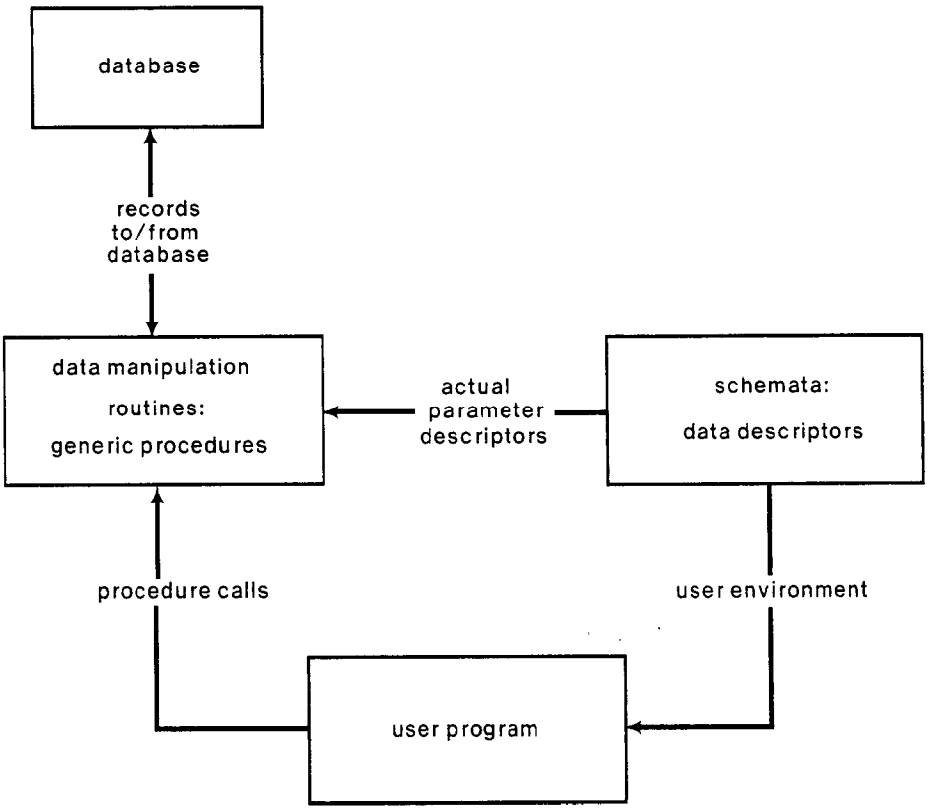


Fig. 1. A generic procedure model of database systems.

relation type, set type), a data manipulation routine utilizes these schema descriptors to determine which access methods are to be invoked to perform the desired action.

By representing the data manipulation routines as generic procedures, we can model a database system as shown in Figure 1. This model highlights the two basic functions performed by a database system. The first function is to provide run-time access to the database. However, not only does the database system perform run-time access to the database, but secondly, it controls access to the *meta-information.* This meta-information consists of the schemata that describe both the representation of information and the primitive operations that may be performed on this information (e.g., the access methods that are available). This meta-information is, in effect, a private database for the database system software.

Management of this meta-information involves two functions. The first is a mechanism that enables the data manipulation routines to associate a given user program with a particular schema and with a particular database. This mechanism makes available the data descriptors required by the data manipulation routines to perform run-time access to the database. The second function is related to the first and defines for user programs the attributes of database objects that the user program references. The ability of the database system to

control and to restrict access to the data descriptors in the schema has a significant impact on data independence.

Since the time of binding also has a major impact on data independence, techniques for managing the meta-information differ in terms of three major characteristics:

(1) mechanisms employed to share the data descriptors maintained by the database system among all user processes accessing the database through a particular schema,
(2) controlling access to attributes within the data descriptor to enforce or guarantee data independence,
(3) time of binding the data descriptors to the attribute names and attribute types referenced in the user's data manipulation routine call.

The implementation approaches used for database systems are direct analogs of the approaches used within programming languages. In order to implement all of the various computations required for the legal parameter types to a generic procedure, a programming language compiler must provide access to a description of the characteristics of all valid actual parameter types. Thus the types of the operands can be regarded as implicit parameters to the procedure. In programming languages two approaches are used to bind the descriptor of the actual parameter characteristics to the generic procedure. The first approach is the *compilation approach*, which binds this information at compile time. By contrast, the binding may be postponed until run time by using the *interpretive approach*.

In the compilation approach the descriptor is the type information from the compiler symbol table. One implementation method for this approach is to supply an instance of the procedure implementation for each type of actual parameter. The procedure body instances are expanded at compile time as macros [14].

As already noted, run-time binding involves interpretation. EL1 is an example of a language that supports the interpretative approach [31]. EL1 represents type information as a MODE (similar to the tag of a variant record) which is associated with each variable and which is testable at run time. A procedure may perform operations on its arguments depending upon the run-time value of the argument's MODE.

For a database system, the determination of when the data manipulation routines bind the data descriptor from the conceptual schema and internal schema to their actual parameters is a critical decision. In general, the longer binding can be delayed, the easier the goal of data independence is to support. Thus the most frequently used implementation technique is the interpretive approach. Variations of the interpretive approch are used in IMS [16], DMS1100 [24, 25], INGRES [27], and so forth. The conceptual schema and internal schema are encoded into an internal form, referred to as the object schema. Using the types referenced as actual parameters in a data manipulation routine call, the object schemata are accessed to retrieve the appropriate descriptors. These descriptors are then interpreted as one of the first steps in performing the data manipulation language command.

In DMS1100, which we believe is a typical example of this approach, the source schema is translated into a form suitable for interpretation at run time. A

secondary output of this translation is a set of specifications for a result delivery area that serves as a communications buffer between the user program and the database. The user program is then compiled using the object schema for type checking. The result delivery area specifications generated in the previous phase are used as a template to generate a user work area within the user's object module.

Final binding occurs at run time. Each user call to a data manipulation routine specifies one or more actual parameters. For each actual parameter the corresponding type descriptor is located in the encoded schema. This descriptor is interpreted to determine the function to be performed by the data manipulation routine, for example, the access methods to be employed and the like.

The interpretive approach has some significant disadvantages. Interpretation will require increased processing time to interpret the object schema and the object subschema. If the object schema is stored on secondary storage to facilitate sharing by all programs accessing the database, then increased I/O costs will be incurred in addition to the increased processing costs.

Database system implementations have traditionally avoided the compilation approach because it tends to sacrifice the important objective of data independence for increased performance. This loss of data independence occurs with compile-time binding because once the schema descriptors are bound to a user's program, a change to the physical storage structure of the database or to its access mechanisms will invalidate all user programs that are dependent on that particular mechanism. These programs will require recompilation.

System R is an example of a database system that supports the compilation approach and yet avoids any loss of data independence [2, 7]. The fundamental idea of the System R approach is that when a user's program is compiled, each embedded data manipulation command is replaced with a call to a compiled version of the command. All compiled commands from the program (along with each original source command) are then combined into a module that is placed in a library of such modules associated with the database. When the user's program is executed, the various entry points of the module are invoked to operate on the database.

Whenever a storage structure or access mechanism is altered, System R examines each module in the library and marks those sections of each module that have been invalidated by the changes. The next time an invalidated module section is invoked, the original command is automatically recompiled (using the new access paths, for example) and then executed. In this way both the goals of data independence and fast query execution for the majority of queries can be achieved.

In the next section we will present a database system implementation technique that also achieves both of these objectives through the use of object-oriented programming and load-time binding of schema descriptors.

3. AN OBJECT-ORIENTED APPROACH

In Section 2 we introduced a generic procedure model of database system implementation techniques that we demonstrated is used to support a generic procedure interface to users. The implementations that we described do not

exploit recent advances in programming languages and methodologies. We believe that it is important to examine whether the concepts in modern programming languages are appropriate for supporting the generic procedure interface that is found in database systems. In this section we demonstrate that object-oriented programming, which is supported by a number of programming languages, provides structures and facilities that exactly match the implementation needs of database systems.

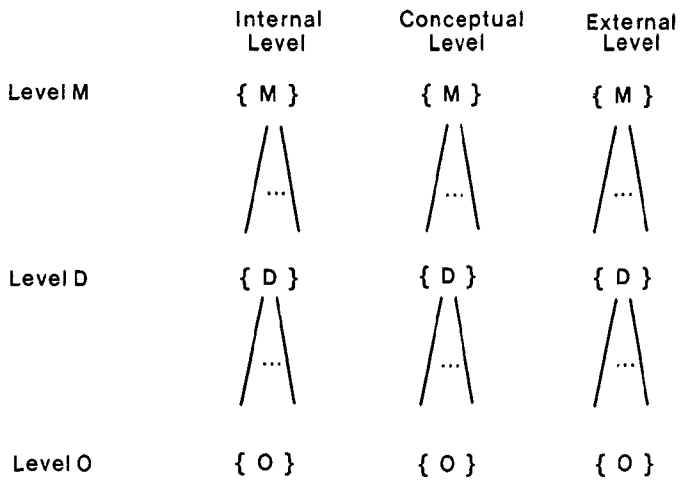
As our earlier discussion implied, the external, conceptual, and internal schemata define multiple levels of data representations and define the data structures and computations that are used to perform efficient access to/from the database and to perform mappings between the various levels. Our background in programming languages leads us to examine the object-oriented approach because we believe that it is natural to regard a database system as supporting not only levels of data representations but also levels of procedures that operate on these data representations. The object-oriented approach places each procedural level with the data which it manipulates, thus assuring that the procedure manipulates data at the proper level of abstraction. In this way primitive data manipulation functions are associated with each schema level. If the procedures at a given level are used as primitives to implement the outer procedural levels, then we can guarantee that the only access to the database is via the generic procedure interface.

The object-oriented approach also achieves both of the objectives of data independence and run-time efficiency that are achieved by System R. However, in contrast to System R, which compiles a separate query packet for each data manipulation routine call, we compile the procedures associated with each object. Then at run time the data manipulation routines select the appropriate objects and invoke the procedures associated with them.

In Sections 3.1 and 3.2 we present an overview of an object-oriented approach to database system implementation (for more implementation details we refer the reader to examine [3–5]). In Section 3.3.1 we describe a network data model system that was implemented using the object-oriented approach. In Section 3.3.2 we discuss a proposal for applying this approach to a relational database system.

3.1 Basic Assumptions

The basic building block we use in our approach is the *class* construct. A class is the encapsulation of a data structure and the procedures that manipulate the internal representation of the data structure. A key feature of a class is that the internal representation of the data structure is hidden from users. Users of a class instance are *only* able to manipulate the data structure by calling one of the procedures associated with the class or by accessing attributes that are declared as externally visible. It is important to distinguish between a class and instances of the class. This difference can be understood in terms of the difference between a type and instances of a type. For example, INTEGER variables are instances of the type integer. By analogy, a class is the declaration of an abstract data type and *objects* are instances of the class.



Note:

Level M: Classes for modeling constructs of the data model

Level D: Classes correspond to entity and relationship types of the database. Each D class is a subclass of a Level M class

Level O: Objects corresponding to the physical database and to the record delivery area. Each object is an instance of a D subclass

Fig. 2. The class hierarchy.

A class defines a group of objects with similar properties. A class can also be partitioned into secondary groups based on common properties, or into *subclasses*. In the examples presented below we base our definitions of class and subclass on SIMULA67 and we use the SIMULA67 syntax [11, 18, 19].

The foundation of our approach for database system implementation is a set of classes M. M contains one class for each fundamental modeling construct provided by the data model to be supported. The type and number of classes in M will vary from data model to data model. All entity types, relationship types, entity occurrences, and relationship occurrences are instances of one of these classes, and can be organized into the subclass hierarchy shown in Figure 2.

There are three levels in this hierarchy. The first level is the M. The number of members in this set, $|M|$, is the number of classes required to represent the abstract properties of the data model and is dependent on the number of levels of representation used in the data model. For the ANSI/SPARC organization, the classes of M are partitioned into three disjoint subsets M_e, M_c, and M_i corresponding to the external, conceptual, and internal levels [1]. In this section we do not explicitly consider the external level. However, in Section 4 we discuss how the object-based approach also supports the external level.

The second level in the hierarchy shown in Figure 2 is the set of classes D, which is partitioned into two disjoint subsets D_c and D_i corresponding, respectively, to the entity types and relationship types defined by the conceptual and

internal schemata for a specific database. The number of members in D is determined by the number of entity types and relationship types defined by the conceptual and internal schemata. Each member of D_c (D_i) is a subclass of some class in M_c (M_i) and shares the external interface it defines. Defining each member of D to be a subclass of a member of M allows it to support the same external interface, but to internally implement this interface in a unique way, if necessary. In addition, each member of D concatenates to the attributes defined at level M the attributes defined in the conceptual or internal schema that are specific to an entity type or relationship type in the database.

The third level is the set O and is composed of those objects that define the occurrences of entities and relationships within the database. The set O also defines the user's record delivery area. Each member of O is an instance of a subclass defined at level D. For example, assume that level D contains a subclass specifying a SUPPLIER record type. Then level O will contain objects corresponding to SUPPLIER record occurrences in the database. (The subclasses defined at level D may be regarded as templates that are used to generate the objects at level O.) Level O also contains an object implementing a SUPPLIER record delivery area. Thus O is partitioned into two disjoint subsets: O_c, the set of objects which implement the record delivery area and which are instances of D_c, and O_i, the set of objects which correspond to database record occurrences and which are instances of D_i.

3.2 Overview of the Approach

The view presented by the object-oriented approach to a user's program consists of four components. The first component is O_c, which is the user's logical view of the database and consists of the entities and relationships that are defined by the set D_c. The second component is the database, which contains objects belonging to the set O_i, that is, records in their internal schema representation. The third component supported by the object-oriented approach is the set of data manipulation routines. These routines are implemented in terms of the attributes of the classes in level M. These routines provide the only access path to the database for the user's program and are responsible for mapping a user's request in terms of the conceptual schema into the access methods and entity types supported by the internal schema. The final component is a result delivery area which resides within the address space of the user program and which is implemented by the conceptual schema objects O_c. Entities in an internal schema representation returned by the data manipulation routines are placed in the record delivery area after conversion to their conceptual level representation.

Three major phases or steps are used to generate the set of objects O, which comprise a database instance. The first phase involves definition of the set of classes M, which are required to support a specific data model. On the basis of the definition of the external interfaces of these classes, the data manipulation routines are implemented by using the procedural attributes of the classes in the set M as primitive operators. At this point the user's view of the *logical properties* of the data model is complete.

The next phase is the translation of the conceptual and internal schemata into the declarations of subclasses at level D. On the basis of the data and procedural

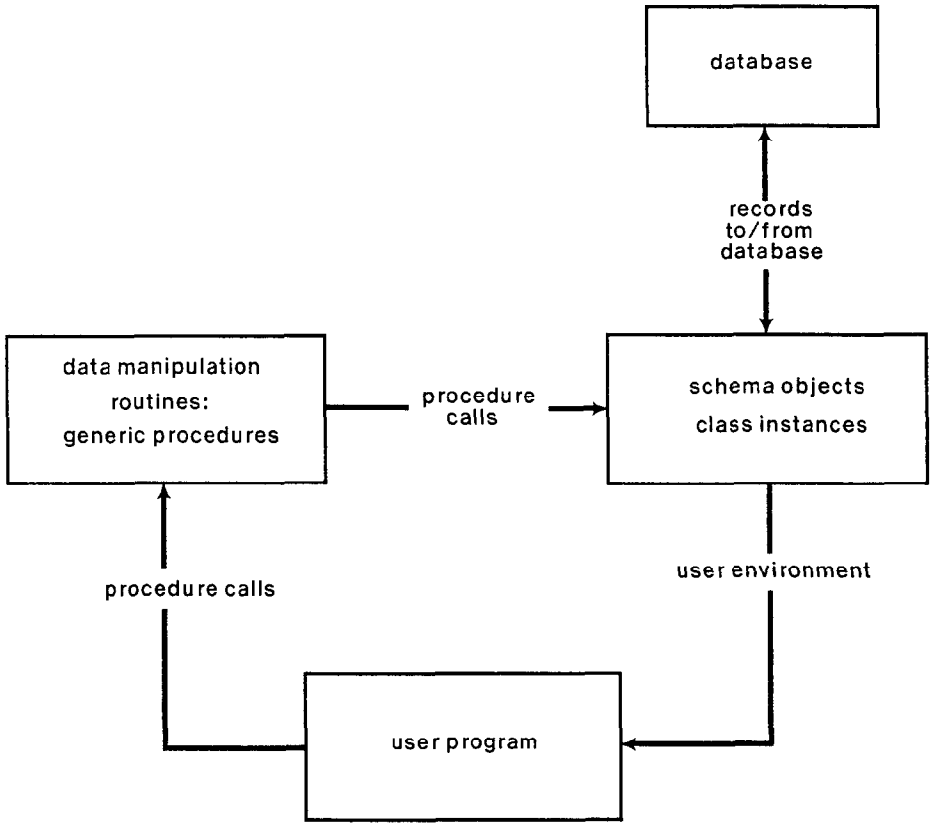


Fig. 3. An object-oriented model of database systems.

definitions in the schemata, these subclasses implement the data and procedural attributes of the data model classes. Associated with each subclass definition are the data attributes of a database entity type, or relationship type, and implementations of a set of procedures that access these data. These procedure implementations are the primitive operations that are used to perform run-time access to a database instance.

The third, and final, step is the generation of the objects which comprise the user's record delivery area and which comprise the physical database. The record delivery area is a set of objects that are subclasses of D_c; the objects that comprise the database are instances of D_i.

For each component of the conceptual view of the database, that is, for each member of D_c, there is one object in the user's record delivery area. These objects are automatically generated and bound to each user program at run time. The objects that comprise the database are generated at run time in response to calls to the data manipulation routines by using the subclasses in D_i as templates.

By implementing each schema level as a collection of objects, we have converted the schemata from their passive role in other implementation approaches to an active role, as shown in Figure 3. The underlying theory of this implemen-

tation approach is quite simple. In other approaches the data manipulation routines can be viewed as first-order functions, and the actual parameters are used to determine what computation is performed (either at compile time or at run time). By contrast, in our approach the data manipulation routines are second-order functions. The actual parameters to the data manipulation routines are references to objects and to perform the requested computation, the data manipulation routines invoke, at run time, the procedures that were bound to these objects at compile time.

Specifically, in the interpretive approach a user's program that calls a data manipulation routine will pass conceptual schema entities and relationships as actual parameters to the routine. The data manipulation routine will use these parameters to examine an encoded form of the conceptual schema to determine what operations should be performed in order to carry out the user's request. These operations generally involve invoking the access methods associated with the internal schema in order to retrieve (store) an entity occurrence from (into) the database.

In the object-oriented approach, when a data manipulation routine is called, the user program passes as an actual parameter a pointer to an object in the conceptual level, that is, a member of O_c. The data manipulation routine then uses the procedure and data attributes associated with this object to perform the desired function. A procedure attribute of the object in O_c will generally, in turn, access data attributes and perhaps procedure attributes of internal schema level objects (i.e., members of O_i) during its execution. The procedures associated with the classes at each representation level are also generic procedures and are used as primitives to implement the procedures at an outer level.

3.3 Implementation Examples

The conceptual and the internal schemata perform two functions. The first is to provide descriptions of the actual parameters to the data manipulation routines. The second is the definition of the environment for user programs. Corresponding to these two functions the database system must support two different representations of the database, the internal representation and the conceptual representation. In the following discussion we describe the classes and objects that represent these viewpoints.

The set M_c contains one class for each basic modeling construct that is provided by the conceptual data model being implemented. Two classes are required for the network data model: one for the conceptual schema record type (CREC) construct and one for the conceptual schema set type (CSET) construct. Instances of the CREC class and CSET class correspond, respectively, to the record and set types in the conceptual schema. The data and procedure attributes of these classes will be described below.

Two classes are also required for the entity–relationship data model. One class will represent the entity-set construct and one will represent the relationship construct.

Since the relation construct is used to represent both entities and relationships in the relational data model, only one class for the conceptual schema relation

construct (CREL) is required. Instances of the CREL class correspond to the relations defined by the conceptual schema.

The number of classes representing internal schema entity types and relationship types in the set M_i depends on the implementation strategy defined by the internal schema. For example, if network data model sets are implemented using linked lists, then only one class is needed in M_i [30]. Instances of this class correspond to physical record occurrences in the database, and set membership information is embedded within these objects as pointers to other objects in the same set occurrence. On the other hand, if pointer arrays are used to implement sets, two classes are required. One class represents physical record occurrences and the other represents physical set occurrences.

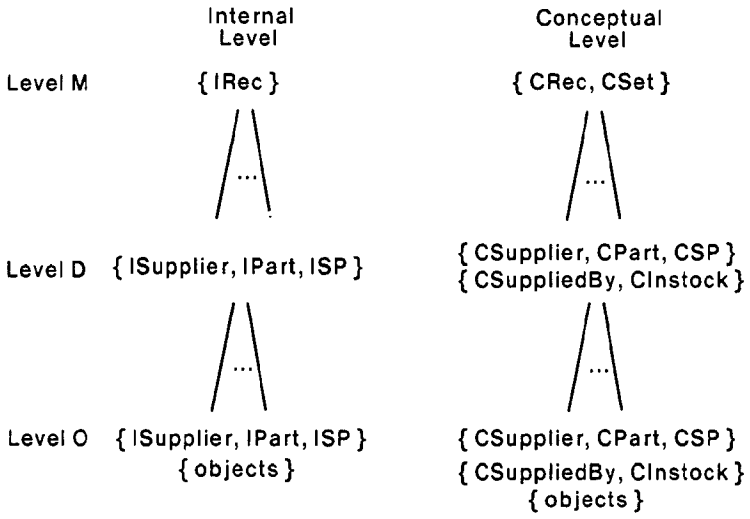
The number of classes required to implement the internal schema for the relational model also depends on the implementation strategy chosen. If relations are stored as heaps with secondary indices, then two classes are required: one for tuple occurrences in the heap and one for secondary indices. If B-trees are used instead, one class is sufficient since a B-tree can be used as both the primary storage for the relation as well as a mechanism for secondary indices.

3.3.1 *A Network Database System Implementation.* In this section we examine a CODASYL-DBTG system that was implemented using our approach. The hierarchy of classes and objects representing the Supplier–Parts database is shown in Figure 4.

The first representation we consider is the physical database defined by the internal schema. The only member of M_i is the IREC class, which is an abstraction of internal schema record types. Level D subclasses of IREC are generated for each record type in the schema and are used as templates to generate all record instances in the database. Each physical record occurrence (internal schema record occurrence) is an instance of an IREC subclass and is thus an object. SIMULA67 class concatenation is used to concatenate the internal schema data items defined in each specific subclass D_i with the IREC object. Therefore each entity in the database is a concatenated object containing the common properties of all record objects (as defined by the IREC class) and those of a specific record type (as defined by the D_i subclass of which it is an instance). The subclass ISupplier, IPart, and ISP are defined as subclasses of the IREC class as shown in Figure 4.

The IREC class and the ISupplier subclass are shown in Figure 5. The VIRTUAL attributes of IREC implement set membership and ownership as specified in the schemata. The implementation of VIRTUAL attributes First and Last in ISupplier uses pointer chains to implement ownership of the SuppliedBy set type.

The second representation of the database is the conceptual representation and is defined by the CREC and CSET classes in M_c. The CREC class is an abstraction of the network data model record construct. Associated with each instance of the CREC class (i.e., a record type in the conceptual schema) are data attributes that describe the set types in which record occurrences may participate and procedures that implement the access methods (e.g., location mode clause). Additional procedures are associated with the CREC class that map the concep-



Note:

Level M: Irec, Crec, and Cset classes are modeling constructs of network data model

Level D: Classes correspond to the record types (Supplier, Part, SP) and set types (SuppliedBy, Instock) of the Supplier-Parts database

Level O: ISupplier, IPart, ISP implement record occurrences
CSupplier, ... implement user view of record types
CSuppliedBy, ... implement user view of set types

Fig. 4. The network model class hierarchy.

tual representation to/from its corresponding representation as an IREC in the database.

The CSET class, which defines the properties common to all set types, is the second member of M_c. The CSET class embodies relationships as an entity. Associated with the CSET class are attributes that describe the record types, that is, CREC instances, which participate in the set as owner and member. The attributes of the CSET also include procedures that implement the procedural aspects of the set declaration, such as the set occurrence selection criterion.

Each CSET instance actually defines two relationships. The first is between those IREC instances that participate in a set occurrence. This relationship is implicitly represented by the procedures associated with the CSET. This relationship is not explicitly visible to users except through the use of a data manipulation routine. The second relationship is the relationship between CREC instances that represent the set owner and set member. This relationship models the relationship defined in the conceptual schema and is explicitly represented by an owner CREC and a member CREC pointer in the CSET object. The procedures associated with the CSET class map between these two relationships.

```
CLASS IRec;
      VIRTUAL:   REF (IRec) ARRAY First, Last;
                 REF (IRec) ARRAY Succ, Pred, SetOwner;
      COMMENT First & Last implement set ownership
      COMMENT Succ, Pred & SetOwner implement set membership
BEGIN
      TEXT RecId;
      INTEGER RCode;
      REF (CRec) RecPtr;
      INNER;
END;


IRec CLASS ISupplier;
BEGIN
      TEXT SName, SCity;
      INTEGER SNumber, SStatus;
      REF (IRec) ARRAY First, Last (1:1);
      COMMENT Initialize IRec data ;
      RecId : = "SUPPLIER";
      RCode : = 1;
      NUMSETSOWNED : = 1;
      RecPtr :· Supplier;
END;
```

Fig. 5. The IRec class and the ISupplier subclass.

For each record type and set type in the conceptual schema, subclasses of CREC and CSET are generated in level D. There is one CREC (CSET) subclass in D_c for each record (set) type in the conceptual schema. Each member of D_c uses class concatenation to combine the common CREC (CSET) properties with those of a specific record (set) type declared in the conceptual schema. The CREC classes, CSupplier, CPart, and CSP, and the CSET classes, CSuppliedBy and CInstock, are shown in Figure 4. Abbreviated examples of CREC and CSET are shown in Figures 6 and 7 along with examples of subclasses generated from the conceptual and internal schemata.

A feature of object-oriented programming that is used in this approach is the separation of the definition of an abstraction from its implementation; for example, the separation of cluster declarations in CLU from their implementation in the CLU library. VIRTUAL class attributes are a mechanism defined in SIMULA67 for performing this separation. Using the VIRTUAL facility, attributes are defined to be part of a class, but implementation may be postponed until the class is used as a prefix, in which case the prefixed class may supply an implementation.

VIRTUAL attributes achieve an effect that is similar to that of FORWARD procedures in PASCAL: the properties of a class attribute, for example, procedure A in class B, can be declared before the attribute is implemented. In the example of class B, the attributes of VIRTUAL procedure A include its type and its parameters. This enables code in a strongly typed language that references class B objects to be compiled before procedure A has been implemented. When instances I1 and I2 of class B are created, a procedure body for A may be supplied

```
CLASS CRec;
     VIRTUAL:  REF (CSet) ARRAY OwnerOf, MemberOf:
               PROCEDURE Locate, Allocate, Store, Load;
BEGIN
     REF (IRec) Current;
     INTEGER NumSetsOwner;
     INTEGER NumSetsMember;
     TEXT Id;
     INTEGER Code;
     INNER;
END;


CRec CLASS CSupplier;
BEGIN
     HIDDEN Allocate, Locate, Store, Load, OwnerOf:
     TEXT SName, SCity;
     INTEGER SNumber, SStatus;
     REF (CSet) ARRAY OwnerOf(1:1);
     PROCEDURE Allocate;
          COMMENT Code here allocates a new ISupplier using a hashing
          algorithm with SNumber as the key and make it current;
     PROCEDURE Locate;
          COMMENT Code here retrieves an ISupplier by hashing with
          SNumber as a key and make it current;
     PROCEDURE Store;
          COMMENT Code here performs encoding as required and copies
          local data items from CSupplier into current ISupplier database
          object;
     PROCEDURE Load;
          COMMENT Code here performs decoding as required and copies
          data items from current ISupplier database object into local data
          items of CSupplier;
     COMMENT Initialize CRec data ;
     Id := "SUPPLIER";
     Code := 1;
     NumSetsOwner := 1;
     COMMENT SuppliedBy is a reference to CSuppliedBy;
     OwnerOf(1) :- SuppliedBy;
     NumSetsMember := 0;
END;
```

Fig. 6. The CRec class and the CSupplier subclass.

for each class instance. These implementations of A are bound to a class instance, that is, I1 and I2, and are referenced as attributes of the class B.

The procedure attributes of the CREC and CSET classes are VIRTUAL. When a CREC (or CSET) object is created in level O_c to form, for example, a CSupplier object corresponding to the SUPPLIER record type, a customized procedure for each VIRTUAL procedure will be generated on the basis of the schema definition of that record (or set) type. Furthermore, each CREC (or CSET) object will have an independent implementation of the VIRTUAL attributes.

As an example, the LOCATE procedure of the CREC class implements retrieval of IREC instances using the strategy specified in the location mode clause of the schema. Each CREC instance possesses an independent implementation of LOCATE that is based on the location mode clause of the schema record type declarations in the conceptual schema. In our Supplier-Parts schema example, CSupplier is an instance of CREC and possesses as an attribute a procedure

```
CLASS CSet;
      VIRTUAL: PROCEDURE Remove, Scan, Insert;
               REF (IRec) PROCEDURE Locate;
               REF (CRec) OwnerRecord, MemberRecord;
BEGIN
      REF (IRec) Current;
      TEXT Id;
      INTEGER Code;
      REF (CREC) OwnerRecord, MemberRecord;
      INTEGER OwnerOffset;
      INTEGER MemberOffset;
      COMMENT Remove: remove specified IRec from a set occurrence
              Scan: traverse set using pointers and OwnerOffset &
                    MemberOffset;
              Insert: add a new IRec instance into set occurrence;
              Locate: implement set occurrence selection;
      INNER;
END;


CSet CLASS CSuppliedBy;
BEGIN
      PROCEDURE Locate;
            COMMENT code here implements set occurrence selection
            through location mode of owner by invoking OwnerRecord.Locate.
            Current becomes the owner occurrence;
      PROCEDURE Remove;
            COMMENT code here deletes the indicated IRec instance from a
            set occurrence. Use OwnerOffset and MemberOffset to reference
            the current pointers;
      PROCEDURE Scan;
            COMMENT code here traverses a set occurrence. Use
            OwnerOffset and MemberOffset to reference the current pointers;
      PROCEDURE Insert;
            COMMENT code here inserts new IRec instance into a set
            occurrence. Maintain set order as sorted in ascending order using
            SNumber;
      COMMENT initialize CRec data;
      Code : = 5;
      Id : = "SuppliedBy";
      OwnerOffset : = 1;
      OwnerRecord :- Supplier;
      MemberOffset : = 1;
      MemberRecord :- SP;
END;
```

Fig. 7. The CSet class and the CSuppliedBy subclass.

LOCATE that implements hashing on its LASTNAME attribute. CSP is also a CREC instance but possesses an implementation of LOCATE that performs retrieval via the pointers of the INSTOCK set type.

As discussed earlier, the data manipulation routines are implemented solely in terms of the data attributes and procedure attributes of the CREC and CSET classes. Figure 8 contains an implementation of the data manipulation routine FETCH, which has one parameter, RECPTR, which is a pointer to a conceptual schema record. RECPTR is used to call at run time the implementation of the LOCATE procedure associated with the proper CREC instance on level O_c. In this way run-time interpretation of the schema can be eliminated. For more detail, the reader is encouraged to examine [3].

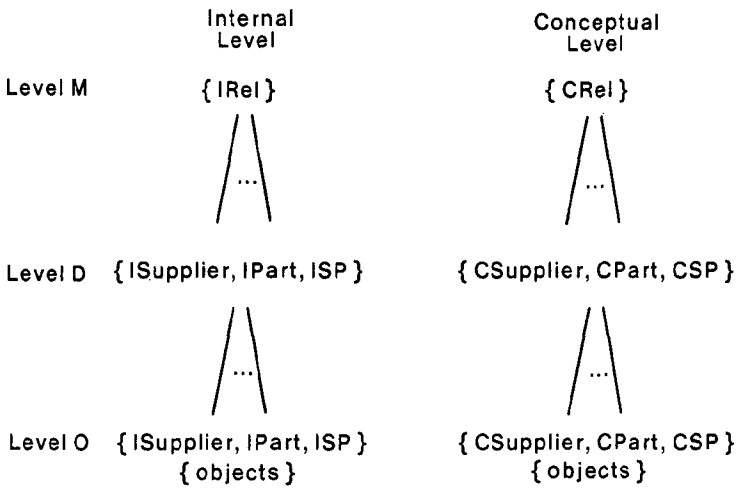
```
PROCEDURE FETCH(RECPTR);
REF (CRec) RECPTR;
BEGIN
        INTEGER I;

        COMMENT use the schema location mode to find the Oi instance
                (IRec) and then copy Oi instance fields into Oc instance
                (CRec) field. Also make the Oi instance the current of record
                type;

        CurrentOfRun :- RECPTR.Current :- RECPTR.Locate;

        COMMENT Load all data items into the Record Delivery Area. Then
                make the record occurrence the current of set in all sets in
                which it participates ;

        RECPTR.Load (RECPTR.Current);

        FOR   I:=1   STEP   1   UNTIL   RECPTR.NumSetsOwned   DO
                RECPTR.Owner(I).Current :- RECPTR.Current;

        FOR   I:=1   STEP   1   UNTIL   RECPTR.NumSetsMember   DO
                RECPTR.Member(I).Current :- RECPTR.Current;

END;
```

Fig. 8. The data manipulation routine FETCH.

The examples that we have shown implement record-at-a-time access to the database. The object-oriented approach could also be used to implement a set-at-a-time interface. Two approaches are possible. In the first, set-oriented data manipulation routines could be implemented using the functions associated with CREC objects as primitives. In the second approach, set-oriented operations could be defined as attributes of the CSET objects.

3.3.2 *A Relational Database System Implementation.* The same implementation strategy can also be used to implement relational database systems. As an illustration we present an example of such an implementation in this section. We have assumed that all relations in the database are stored as heaps and that no secondary indices are available to enhance system performance. As shown in Figure 9, the IREL class, an abstraction of all internal schema relation formats, is the only member of M_i required to model internal schema objects. As with the network example, the set D_i contains a subclass of IREL for each relation in the conceptual schema. The members of D_i are used as templates to generate tuple instances in the database.

As described previously, only one class, CREL, is defined in M_c to represent the conceptual schema for the relational data model. The structure of the CREL class is shown in Figure 10. Using the same approach described in Section 3.2.1 for the network data model, the set D_c contains an instance of CREL for each relation type defined in the conceptual schema. Associated with a CREL instance are procedure attributes that are used both as access methods (to map the conceptual representation of a tuple to/from its internal representation) and as primitives for the relational algebra operators. The data attributes of an object in



Note:

Level M: IRel and CRel classes are modeling constructs of of the relational data model

Level D: Classes correspond to the relation types (ISupplier IPart, ISP) and to user view of relations (CSupplier CPart, CSP) of the Supplier-Parts database

Level O: ISupplier, IPart, ISP implement tuple instances CSupplier, Cpart, CSP implement user view of relations

Fig. 9. The relational model class hierarchy.

D_c correspond to the attributes of the relation type. Abbreviated examples of IREL and CREL are shown in Figures 10 and 11, along with subclasses generated from the conceptual and internal schemata.

To illustrate how these procedure attributes are used to implement relational algebra operations, consider the following restrict operator:

RESTRICT(SRELPTR, COND, TRELPTR)

This operator extracts from the source relation (SRELPTR) those tuples that satisfy condition COND and places the qualifying tuples in the target relation (TRELPTR). The form of the selection condition argument COND might be

$$\text{attribute}_j \text{ op value} \qquad \text{where} \quad \text{op is } =, \#, >, <, <, >$$

(Our approach can easily be extended to a more complex selection criterion.) The code for the RESTRICT operator is shown in Figure 12.

The function of the ScanId is to provide a place marker on the relation being examined so that the NEXT procedure attribute can return the next tuple relative to the current position of the ScanId and also update the ScanId. This feature is especially useful in processing some relational queries such as joining a relational with itself.

```
CLASS IREL;
BEGIN
      INTEGER TId;
      REF (CRel) RelPtr;
      BOOLEAN DELETEFLAG;
      COMMENT  Since relations are maintained as heaps, this flag
               indicates whether the tuple instance has been deleted;
      INNER;
END;


IRel CLASS ISupplier;
BEGIN
      TEXT SName, SCity;
      INTEGER SNumber, SStatus;
      RelPtr :- Supplier;
END;
```

Fig. 10. The IRel class and the ISupplier subclass.

```
CLASS CRel;
     VIRTUAL PROCEDURE  APPLY, CLOSE, DELETE, INSERT, NEXT,
        OPEN;
BEGIN
     REF (IRel) ScanId;
END;


CRec CLASS CSupplier;
     HIDDEN APPLY, CLOSE, DELETE, INSERT, NEXT, OPEN;
BEGIN
      TEXT SName, SCity;
      INTEGER SNumber, SStatus;
      REF (CSet) ARRAY OwnerOf(1:1);
      PROCEDURE APPLY;
           COMMENT apply qualification to a tuple instance;
      PROCEDURE CLOSE;
           COMMENT a virtual procedure to close a scan on a relation;
      PROCEDURE DELETE;
           COMMENT a virtual procedure to delete a tuple (i.e. member of
           Oi) from a relation by setting DELETEFLAG;
      PROCEDURE INSERT;
           COMMENT This virtual procedure will attempt to insert the new
           tuple(i.e. a new member of Oi) in the space currently occupied by
           a deleted tuple. Otherwise new space is allocated in the heap;
      PROCEDURE OPEN;
           COMMENT a virtual procedure to open a scan on a relation.
           Returns a scan pointer to first tuple in the relation;;
      PROCEDURE NEXT;
           COMMENT a virtual procedure to return the next tuple in a
           relation;
END;
```

Fig. 11. The CRel class and the CSupplier subclass.

```
PROCEDURE RESTRICT (SRELPTR, COND, TRELPTR);
BEGIN
    REF (IRel) ScanId;
    COMMENT Open scan on source relation. Then apply selection
        criteria and insert qualifying tuple into target relation;
    ScanId :- SRELPTR.OPEN;
    WHILE ScanId =/= NONE DO
    BEGIN
        IF (SRELPTR.APPLY(COND))
        THEN TRELPTR.INSERT (ScanId);
        ScanId :- SRELPTR.NEXT (ScanId);
    END;
    TRELPTR.ScanId :- SRELPTR.OPEN;
    SRELPTR.CLOSE;
END;
```

Fig. 12. The data manipulation routine RESTRICT.

If our example had been based on a more sophisticated storage structure, then the CREL class would probably have had additional procedure attributes so that the next tuple with a particular value could be returned directly.

4. AN EVALUATION OF THE OBJECT-ORIENTED APPROACH

In the previous sections we have presented an object-oriented approach to database system implementation. Traditionally, a database system has been required to support data independence and multiple concurrent users while providing acceptable performance. In this section we present a critical evaluation of our approach in terms of these requirements and contrast its performance with other techniques. We begin with a discussion of data independence that is extended to examine management of meta-information and its relationship to query optimization and the frequency of binding. Following this discussion, we then examine several concepts related to the support of multiple users: concurrency control, support of low-level views, and finally support of high-level views.

Date defines data independence as the immunity of the application to changes in the storage structures and access strategies of the database [12]. The data manipulation routines support an interface through which the user interacts with the database with no knowledge of the database's underlying representation.

The object-oriented approach not only supports data independence, but also guarantees that it is not violated. Objects encapsulate data structures with the procedures that manipulate the data. The user environment consists of the data manipulation routines and the names and data attributes of the objects representing the conceptual schema; the procedural attributes of these objects are not visible. Since programming languages easily prevent users from accessing information not defined in their environment, the user is required to invoke a data manipulation routine in order to access the database.

Table I. Phases of Binding in the Object-Oriented Approach

Phase of binding	Binding in the object-oriented approach
(I) Binding data model types	(a) Declaration of conceptual schema and internal schema classes (b) Implementation of data manipulation routines in terms of conceptual and internal schemata class attributes
(II) Binding database entity types to data model types	(a) Translation of data definition language syntax into corresponding subclasses (b) Compilation of these subclasses
(III) Binding user program to database entity types	(a) Compilation of user program: binding user references to attributes of subclasses (b) Linking user program to subclasses

This call to a data manipulation routine is a transition to a separate environment in which not only the data attributes are visible, but also the procedure attributes. These procedures are used as primitives to implement the data manipulation routines and are the only location of knowledge of the database's structure and access methods. Thus the user is isolated from the database's structure by at least two levels of procedural abstraction.

Associating primitive operations with objects allows the database system to be simplified by eliminating the need to manage meta-information. The stages of binding that achieve this are shown in Table I. The first stage defines the classes that represent the basic constructs of the data model and implements the data manipulation routines in terms of these classes.

In the second stage the schemata are converted from their data definition language representation into object declarations, which are subclasses of the data model classes defined in Phase I. At the completion of this step, all entity and relationship types within a database are defined and the procedures that may be invoked to access entities within the database are implemented. These procedures and the data manipulation routines comprise the database system.

After Phase II, these objects are managed by the user's programming language, not by the database system. The programming language's environment, or name scope, control mechanisms are used to control which attributes of the objects defined in Phase II are accessible to the user program. As we show later, these environment control mechanisms are also used to support views.

The binding mechanisms within a database management system have an impact upon system performance. To understand this impact, we compare the frequency and overhead of binding in our approach with that in the interpretive and the compiled approaches that we introduced earlier. The object-oriented approach is the only approach in which Phase I binding is performed explicitly; in the interpretive and the compiled approaches it is peformed implicitly.

Performing Phase I explicitly allows the management of meta-information to be integrated into the programming language's type system, instead of construct-

ing special functions within the database system. All three approaches perform Phase II binding: the interpretive and compiled approaches translate the data definition language forms of the schemata into internal forms that are managed by the database system; in our approach these schemata are translated into subclass declarations that are managed by the programming language compiler. The three approaches differ strongly in their approach to Phase III binding. The interpretive approach performs this binding during every data manipulation routine call. This provides a high degree of data independence, but results in a significant run-time overhead [4].

In both the System R implementation of the compiled approach and our approach, Phase III binding occurs only after a change in the schemata, for example, after a change in the access methods. In System R, queries are compiled into query packets that are stored in a library associated with the database. Rebinding involves the overhead of completely recompiling those queries that use the invalidated access methods upon their next invocation. In our approach modified subclasses must be recomplied and linking to the user program performed again.

Our approach does introduce new forms of run-time overhead that are not present in the other approaches. The first, and not significant, overhead is the use of indirection within the data manipulation routines to invoke the correct procedural attribute of an object. The second overhead, which may be significant, is the context switch required by each invocation of a procedure associated with an object. Related to the context switching is the additional overhead of code segment management by the operating system.

Given the number of procedure calls executed by each data manipulation routine in our approach, the overhead of procedure entry and exit may be significant. The studies by Scheifler demonstrate that in-line substitution may be an effective way of implementing the object-oriented style of programming [22]. However, in-line substitution requires recompilation of all application programs following a change to the schemata.

To improve database system performance, two general optimization approaches are used to reduce execution time. The first approach, low-level optimization, involves access path selection at run time. Low-level optimization requires that the procedure attributes of objects perform selection of an optimal access path. To make this selection, information concerning database size, such as the size of each relation, must be stored in the database and must be accessible to these procedures. This information can be used to determine which of several access methods will provide minimum execution time. Since this selection is encapsulated within an object, only "local" optimization is possible in our approach. Thus "joint" optimization of access paths for both relations in a join operation is not possible.

The second optimization approach, high-level optimization, involves reordering the user's sequence of data manipulation routine calls to reduce their execution time. High-level optimization requires that the programming languages compiler be enhanced to recognize database queries. The techniques that are commonly used to reorder operators within a query must then be added to the compiler.

4.1 Multiple User Support

A general-purpose multiuser database system must be capable of correctly controlling simultaneous access to the database by two or more users and must be able to provide different external views of the database to different classes of users. In this section we examine how a database system that is constructed using the object-oriented approach can satisfy these requirements.

4.1.1 *Concurrency Control.* One concurrency control mechanism that may be employed to coordinate the operations of processes accessing a common database is based on the approach in System R. In this approach a shared read/write table contains the lock information. This table can reside in either a shared virtual memory segment or a special file, or can be distributed among the headers of all files holding data in the database. Before the user's program (or an access mechanism invoked by a data manipulation routine) attempts to access a database entity (relation, tuple, etc.), this shared table is examined and perhaps modified as an indivisible operation. If requested access is permitted, the transaction proceeds; otherwise the transaction waits. Since this approch may lead to a deadlock condition, some additional mechanisms must be employed to either prevent or resolve deadlock situations.

The same concurrency control scheme can be used for a database system that is implemented using the object-oriented approach. When a user program calls a data manipulation routine (e.g., find next, restrict), the data manipulation routines will invoke those procedure attributes associated with the class instances that were passed as parameter types in order to perform the requested operation. Each procedure attribute, when called, will, when necessary, access the shared lock table to determine whether it can proceed with the action invoked by the data manipulation routine. When the desired action cannot be immediately performed (e.g., another procedure has set an exclusive lock on an entity), the procedure will wait until the monitors controlling access to the database objects grant it permission to proceed. When deadlock is detected, the procedure will be instructed to invoke a rollback/recovery process as a means of safely terminating the transaction on behalf of the user.

If a more active concurrency control mechanism is desired (such as a centralized wound–wait scheme [20], the procedure attribute invoked would send a message requesting permission to perform the desired action to a process that is responsible for controlling concurrent access to the database. This process would respond with a proceed, wait, or die message to the waiting procedure.

Since either of the above approaches is viable for a database system implemented using the object-oriented approach, successful concurrency control is possible.

4.1.2 *Multiple View Support.* The external schema of the ANSI/SPARC data model is intended to provide a mechanism for supporting alternative views of the database from that provided by the conceptual schema. We classify such views into two gross categories: low-level views and high-level views. In this section we discuss the support of both categories of views in a database system using our approach.

Low-level views (what in CODASYL terminology are termed subschemata) support such operations as hiding entities and relationships declared in the conceptual schema, renaming and/or reformatting attributes, and redefining procedural information declared in the conceptual schema. We demonstrate how the object-oriented approach provides an excellent mechanism for supporting such views.

Since the user environment for the object-oriented approach consists of a prefix chain of classes—user program, external view, conceptual schema, data manipulation routines, and the set of data model classes—entities and relationships defined in the conceptual schema can be hidden from the user of an external schema by declaring them to be HIDDEN in the external view. This permits the database administrator to define exactly which components of the conceptual schema are visible to the user.

Renaming of attributes, hiding of attributes, and reformatting of attribute types can be accomplished in our approach by redefining a class instance (corresponding to an entity or relationship) in the view. This permits one to hide certain attributes from a class of users (for example, an employee's salary should not be visible to all users of an employee database) or to rename an attribute so that different groups of users can refer to the same attribute with different names. If this mechanism is used to change the type of an attribute in the view, then each time a user accesses the attribute, a coercion in the view will be invoked to convert the attribute between the type assumed by the user and the type actually stored in the database.

Redefining a class instance in a view can also be used to redefine procedural attributes associated with the instance of the class. This is possible because within the prefix chain multiple implementations of the same virtual attribute may exist. The one chosen by a data manipulation routine at run time is defined by lexical scoping to be the one closest (logically) to the user program. This permits the view to define, for example, a new set-occurrence selection clause for a CODASYL set type or a new order clause for the tuples in a relation. Since each of these procedure attributes associated with a class was defined to be VIRTUAL, redefining them in a view is straightforward.

High-level views are used in an attempt to add more semantics to the database and the operations performed on it. As an example, consider a data model that permits types such as employees with associated operations such as add-new-employee, increase-salary, and delete-employee to be defined. While the object-oriented approach does not enhance the ability to define high-level views, we feel that it provides a convenient and natural framework for their implementation.

5. CONCLUSIONS

This paper has examined object-oriented programming as an implementation technique for database systems. As we have demonstrated, the conceptual schema and the internal schema describe the attributes of database entities and the procedures that access and manipulate these representations. The object-oriented approach encapsulates the representations of entities and relationships with the procedures that manipulate them. To achieve this, we defined abstractions of the modeling constructs of the data model that describe their common properties

and behavior. Then we represented the entity types and relationship types in the conceptual schema by objects that are instances of these abstractions. On the basis of this approach, the generic procedures that comprise the user interface to the database were implemented as calls to the procedures associated with these objects.

This approach guaranteed that the only access path to the database is through the generic procedures, which provides data independence. In addition, we demonstrated that the object-based approach reduces the requirements for the database system to manage meta-information. We also demonstrated that the object-oriented approach has advantages of data independence, run-time efficiency due to eliminating access to system descriptors, and support for low-level views. It was further shown that the object-oriented approach makes query optimization more difficult and increases context switching overhead.

ACKNOWLEDGMENTS

The authors wish to express their appreciation to Dan Murray, Larry Rowe, and the referees for their valuable suggestions during the preparation of this paper.

REFERENCES

1. ANSI/X3/SPARC. Interim Rep. 75-02-08. *FDT Bulletin ACM-SIGMOD 7*, 2 (Feb. 1975).
2. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
3. BAROODY, A.J. The evaluation of abstract data types as an implementation tool for database management systems. Ph.D. Dissertation, Univ. Wisconsin-Madison, Madison, 1978.
4. BAROODY, A.J., AND DEWITT, D.J. The design and implementation of a database management system using abstract data types. Tech. Summary Rep. 1970, Mathematics Research Center, Univ. Wisconsin-Madison, June 1979.
5. BAROODY, A.J., AND DEWITT, D.J. The impact of run-time schema interpretation in a network data model DBMS. Submitted for publication, Jan. 1980.
6. BRODIE, M., AND SCHMIDT, H. What is the use of abstract data types in data bases? In *Proc. 4th Int. Conf. Very Large Data Bases*, 1978, pp. 140–141.
7. CHAMBERLIN, D.D., ET AL. Support for repetitive transactions and ad-hoc query in System R. IBM Research Rep. RJ2551, May 1979.
8. CODASYL. Data base task group report. ACM, New York, 1971.
9. CODASYL. Data description language. *J. Dev.* Document C13.6/2:13, U.S. Government Printing Office, Washington, D.C., 1973.
10. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
11. DAHL, O.J., MYHRHARG, B., AND NYGAARD, K. *The Simula 67 Common Base Language.* Publ. S-22, Norwegian Computing Center, Oslo, Norway, 1970.
12. DATE, C.J. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 1975.
13. FURTADO, A.L. A view construct for the specification of external schemas. In *Series: Monografias em Ciencia da Computacao*, M. Challis, Ed., 1978.
14. GRIES, D., AND GEHANI, H. Some ideas on data types in high-level languages. *Commun. ACM 20*, 6 (June 1977), 414–420.
15. HAMMER, M. Data abstractions for databases. In *Proc. Conf. Data: Abstractions, Definition, and Structure, SIGPLAN Notices 11* (Special Issue), (1976), 58–59.
16. MCGEE, W.C. The information management system IMS/VS. *IBM Syst. J. 16*, 2 (1977), 84–168.
17. MINSKY, N. Files with semantics. In *Proc. ACM-SIGMOD Int. Conf. Management of Data*, June 2–4, 1976, pp. 65–74.
18. MYHRE, O. Protecting attributes of a local class. *SIMULA Newsletter 5*, 4 (Nov. 1977), 14–15. Norwegian Computing Center, Oslo, Norway.

19. PALME, J. New feature for module protection in SIMULA. *SIGPLAN Notices 11*, 5 (May 1976), 59–62.
20. ROSENKRANTZ, D.J., STEARNS, R.E., AND LEWIS, P.M., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
21. ROWE, L.A., AND SHOENS, K.A. Data abstractions, views and updates in RIGEL. *Proc. ACM-SIGMOD 1979 Int. Conf. Management of Data*, May 30–June 1, 1979, pp. 71–81.
22. SCHEIFLER, R.W. An analysis of inline substitution for a structured programming language. *Commun. ACM 20*, 9 (Sept. 1977), 647–654.
23. SCHMIDT, J. Type concepts for database definition. In *Proc. Int. Conf. Data Bases*, Haifa, Israel, Aug. 1978.
24. SPERRY UNIVAC. *1100 Series Data Management System (DMS 1100) Schema Definition Data Administrator Reference Manual*, UP-7907, Rev. 2, Sperry Univac, 1975.
25. SPERRY UNIVAC. *1100 Series Data Management System (DMS 1100) System Support Functions Data Administrator Reference Manual*, UP-7909, Rev. 3, Sperry Univac, 1975.
26. STEMPLE, D.W. A database management facility for automatic generation of database managers. *ACM Trans. Database Syst. 1*, 1 (March 1976), 79–94.
27. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
28. TSICHRITZIS, D.C., AND LOCHOVSKY, F.H. *Database Management Systems*. Academic Press, New York, 1977.
29. WASSERMAN, A.I. The data management facilities of PLAIN. *Proc. ACM-SIGMOD 1979 Int. Conf. Management of Data*, May 30–June 1, 1979, pp. 60–70.
30. WEBER, H. A software engineering view of data base systems. In *Proc. 4th Int. Conf. Very Large Data Bases*, 1978, pp. 36–51.
31. WEGBREIT, B. The treatment of data types in EL1. *Commun. ACM 17*, 5 (May 1974), 251–264.
32. YEH, R.T., AND BAKER, J.W. Toward a design methodology for DBMS: A software engineering approach. In *Proc. Int. Conf. Very Large Data Bases*, Tokyo, Japan, October 6–8, 1977, pp. 16–27.

Received May 1979; revised November 1980; accepted December 1980