A RELATIONAL OBJECT-ORIENTED MANAGEMENT SYSTEM

Michael L. Nelson, Major, USAF
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

J. Michael Moshell and Ali Orooji
Department of Computer Science
University of Central Florida
Orlando, Florida 32816

ABSTRACT

Much research and development is currently taking place in object-oriented database systems. However, conventional data models (such as the relational model) are virtually ignored (other than for comparison purposes) in the development of these systems.

ROOMS (Relational Object-Oriented Management System) is intended to show that we do not have to abandon existing data models in organizing object-oriented data, and to allow more "conventional" applications to take advantage of the additional capabilities of object-oriented programming. By adding ROOMS to a commercially available object-oriented database system, the resulting system should be able to handle all of the data needs of a user (company), thereby eliminating the need for multiple database systems.

1. INTRODUCTION

Most research in object-oriented programming (OOP), including object-oriented database management systems (OODBMS), is concerned with supporting users which are not served well by more conventional technology. Our research has been from a different point of view - our primary motivating factor is to show how existing applications can be enhanced using object-oriented technology.

Like many new ideas, object-oriented programming does not yet have a universally accepted definition [1, 2]. Ideas on the subject do, however, seem to be converging - the "best" definition that we have seen to date is "object-oriented = objects + classes + inheritance" [3]. OOP can also be defined as an extension of the idea of abstract data type (ADT) [4].

A database can be defined as a collection of data stored "permanently" in a computer [5]. The data to be stored permanently in a database is said to be persistent in that it survives beyond a single programming session [6, 7]. A database management system (DBMS) or database system is the software that allows users to use and maintain the data in the database [5].

Object-oriented languages generally lack support for persistent objects, but conventional database systems often lack various features that are needed to store object-oriented data [6, 7]. Perhaps the biggest limitation of conventional database systems is that they offer only a fixed set of data types (such as integers and character strings) while lacking the capability to define even simple new types and operations, much less complex types and an inheritance scheme [6, 7].

It is interesting to note that we actually use abstract data types in conventional languages and database systems without realizing it [8]. The only truly non-abstract type is the bit string. All other types have an internal representation in the computer (a string of bits) and an external representation (a string of characters or digits). The problem is in the limited number of types that are available in conventional systems and the difficulties encountered in attempting to add user-defined abstract data types and operations.

Most commercially available object-oriented database systems organize objects into some type of collection [6, 7]. They make no use of traditional database organization schemes, such as the relational data model, other than for comparison purposes [9, 10]. However, we believe that the relational model [11, 5] is a viable approach to organizing persistent objects in an object-oriented database.

The Relational Object-Oriented Management System (ROOMS) is an experimental system in which objects are organized into records and relations from the more traditional relational model. It has been designed in such a way that it could be used as a stand alone database system or added to any commercially available OODBMS that allows user-defined collections of data.

It can be argued that most companies are not interested in having a different database system for each type of data and user that they have [12]. ROOMS is primarily intended to show that we do not have to abandon existing data models (specifically, the relational model) in organizing object-oriented data, and to allow conventional...
applications to take advantage of the additional capabilities of object-oriented programming. By including ROOMS in an object-oriented database system, a single database system can now handle all of the data and all of the applications that a user (company) might have. Separate systems for conventional applications and object-oriented applications are no longer needed.*

Conventional applications developed in ROOMS can easily be extended to include more complex data. For example, a digitized photograph could be included in an otherwise conventional personnel record. By including different methods for displaying and accessing the data, different applications (users) can easily use and manipulate the same data from completely different points of view.**

We first describe the different approaches for supporting the various types of data and users that a company may have. Next, a system overview of ROOMS is presented. We then discuss the methods that must be added to user-defined classes in order to use them in ROOMS. The options available for the record structure and the five basic relational algebra operations are covered next. Finally, we give a brief introduction to an Encapsulated Object-Oriented Programming System (EOOPS) [15]. EOOPS helps to solve the implementation problems that we encountered in developing ROOMS, and should also be helpful in developing any large object-oriented application.

2. SUPPORTING THE VARIOUS TYPES OF DATA AND USERS

There are several possibilities for supporting the various types of data and users that a company may have. Figure 1 shows "typical" object-oriented and relational systems supporting their respective users. This approach may handle both object-oriented and relational users and data, but does not allow for a "relational-object-oriented" use of the data (i.e., a relational application that includes more complex data).

In Figure 2, a relational interface has been added to an OODBMS. Although this does allow for a relational-object-oriented use of the data, it requires that two individual DBMSs and two physical databases be maintained.

Figure 3 shows the implementation of ROOMS in an object-oriented language. This approach also allows for a relational-object-oriented use of the data, but it does not support non-relational users (although a separate OODBMS could be maintained).

An OODBMS which includes ROOMS is shown in Figure 4. We believe this is the best approach in that it supports all three types of users (relational, object oriented,

---

* Apparently others are beginning to share in the belief that both conventional data and object-oriented data can be used and stored together. Since we began our research effort, GemStone [7, 13] has added the capability to access conventional database systems that have an SQL interface, and Vbase [7, 14] has added an Object-SQL interface.

** This is essentially a way of providing views or subschemes, an abstract model of a part of the data [5].

---

Figure 1: The "Typical" Object-Oriented and Relational Systems

Figure 2: The Relational Interface Approach

Figure 3: The Stand-alone ROOMS Approach
and relational object oriented) while requiring that only a single DBMS and physical database be maintained.

3. THE RELATIONAL OBJECT-ORIENTED MANAGEMENT SYSTEM

3.1. System Overview

The basic structure of ROOMS is actually quite simple. A relation is simply an object that is a collection of records. A record is an object that is a collection of fields. A field is an object that is an instantiation of a user-defined class. No distinction is made between simple objects (such as conventional data) and more complex objects (such as digitized pictures), so all of the user data can be stored together in the same database. In the current implementation of ROOMS, all of the records in a relation must be identical in format, but we are experimenting with defining the class relation as a subclass of a more general class that does not have this restriction.

The five basic relational algebra operations (union, set difference, Cartesian product, projection, and selection), from which all other operations can be constructed [5], have been implemented as methods that are defined for the class relation. Methods for displaying a relation, and for adding and deleting records have also been implemented.

Surprisingly few methods for the class record are needed. The two most important are display yourself and meet select criteria?. Display yourself displays the individual objects making up a record. Meet select criteria? is used by the select method of relations to determine which of the records in a relation meet the specified select criteria.

3.2. Methods Required for User-Defined Classes

Any user-defined class can be used in ROOMS, but certain methods are required of these classes. We have attempted to minimize the number of methods that are required for user-defined classes, but the display and select methods make certain assumptions about the objects that make up the records.

For instance, the message 'relation display yourself' concatenates the result of the message 'record display yourself' for each record in the relation. The message 'record display yourself' concatenates the result of 'user-defined-object display yourself' for each object in the record. As such, each user-defined object must have a display yourself method. Note that this display yourself method can return whatever value is appropriate for the object - it may include the value of all of the variables making up the object, some of them, or none of them. It can even display the object differently depending upon which user is making the request.

Equality methods are also required for each user-defined class. Specifically, greater-than?, less-than?, and equal-to? methods are needed. The equal-to? method must be able to determine if two objects are equal. Similarly, the greater than? and less than? methods must determine if an object is greater than or less than another object, but these two methods may simply return FALSE if it is inappropriate to say that one object is greater than or less than another object. Note that this somewhat restricts our normal use of logic, as it is possible for two objects to be NOT greater than, NOT equal to, and NOT less than one another (this means that greater-than-or-equal-to is not the same as not-less-than, etc.).

In an attempt to minimize what the user must do to add a new class description to the system, we have provided a class which contains the "generic" routines (methods). This class provides display and equality routines when included as an ancestor for a user-defined class. For these routines to work, the user-defined class must have two additional class variables defined for it - display list and equality list. The names of any variables to be displayed, in the order in which they are to be displayed, make up the display list. Similarly, the names of any instance variables to be compared, in the order in which they are to be compared, make up the equality list. *

3.3. The Record Structure

Since a record in ROOMS consists of other objects, either multiple inheritance (MI) or some form of composite object can be used to build the record. Using MI, the record structure is simply defined as having the appropriate user-defined classes as subclasses.

Unfortunately, some OOP languages do not support

* This is similar to the idea of subclass responsibility as used in Smalltalk applications. However, there is no requirement that user-defined classes be a subclass of any other class. Also, when using this class of "generic" methods, user-defined classes are required to include specific variables rather than specific methods. This could easily be changed to the Smalltalk approach, but variables are easier to modify than methods, and in most languages it is faster to directly access a variable rather than send an additional message.
multiple inheritance. And even those that do suffer from potential problems due to name conflicts - if two classes have identical variable names or method names, then one of the variables/methods may be lost if a subclass attempts to inherit from both of these classes.

To avoid this problem, some form of composite object must be used. A composite object is an object which consists of other objects. That is, the variables (or some of the variables) of a composite object are defined as being objects themselves (called dependent objects) [16]. We have also experimented with composite objects consisting of subobjects, where the variables (or some of the variables) of the composite object consist of pointers to other objects.

Composite objects can be viewed as an alternative to multiple inheritance, choosing one or the other depending on what the relationship between the classes should be - composite objects are said to be appropriate when we have the sum of the parts while multiple inheritance is appropriate when we have the sum of the behaviors.

In the current implementation, both dependent objects and subobjects are supported. Subobjects are in some ways the more powerful of the two, but also present more problems. With subobjects, the user can define an object and then attach it to a record by name. The object can then be manipulated by name or as part of the record. It is also possible to share subobjects between records/relations, but this complicates the storage and retrieval of the data, so only temporary relations/records (i.e., relations/records that will not be written to secondary storage) are allowed to share subobjects at this time. Dependent objects can also be defined by the user, but must be attached to the record by value rather than by name. Modifying the value of the original object has no effect on the dependent object and vice versa.

We would prefer to use multiple inheritance, however, as it seems more appropriate to send a message to the record (such as 'record change zip to new-zip') rather than to a specific part of the record (such as 'addr-of-record change_zip_to new-zip'). This is contrary to the arguments presented in [17] on this subject, as a record can be considered to be the sum of its parts (fields) rather than the sum of the behaviors of its parts. However, we believe that a database user should only be concerned with sending messages to relations and to records, not with how the record is constructed or to which part of the record a message must be sent.

3.4. The Relational Algebra Operations

The five basic relational algebra operations have been implemented as methods for the class relation. The union and set difference operations were relatively easy to implement. Both use the equal-to? methods that are defined objects. That is, if OBJ1 is the name of a subobject or dependent object field in a record, then selections can only be made based on equality with the value of OBJ1; selection is not allowed on equality with only a part of the value of OBJ1.

Project and Cartesian product both presented special problems. Since individual record structures are defined as separate classes, the record structure for the result of project and Cartesian product is different from that of the original relation(s). This may produce a new record structure that needs to be defined, or it may produce one equivalent to a previously defined structure. Currently, a new record structure is built unless the user specifies otherwise.

3.5. A Real-World Example

Consider, for example, a real estate database. Virtually any conventional DBMS could be used for a database such as this. Appropriate fields could be used to store various information about each piece of real estate (such as cost, lot size, number of bedrooms, number of bathrooms, square footage, etc.). A query system could then select entries based on any number of constraints on the information.

Unfortunately, non-conventional information (such as pictures and blueprints of a house and maps of the area) could not be added to most conventional systems. In ROOMS, however, any object can be added to the database. Therefore, digitized pictures of a house and drawings representing its blueprints could simply be added to each record. The record could then be displayed differently depending upon the current view of the data (e.g., these pictures and blueprints may not appear when looking at a list of records/houses while they would be present when looking at an individual record/house). A class of maps (with methods such as area_measure and layering) could also be simply defined and used.

Adding a map showing how to get to a house is another problem, even with ROOMS if it has been implemented using the stand-alone approach of Figure 3. A simple map could be added to each record, but it would not be easy to show the "big picture". However, if ROOMS has been added to an OODBMS (as in Figure 4), then a detailed map of the area could be developed outside of ROOMS. The records in ROOMS could then provide some form of coordinates that would indicate where in this detailed map a particular house is located - the OODBMS could then be used to see the general location of the house, details of the immediate neighborhood, or anything in between.

4. EOODS - A BETTER WAY

We have made every attempt to keep the design of ROOMS general enough so that it could be easily implemented in any OOP language or added to any OODBMS. While we have found ways around all of the problems that we have encountered, we hesitate to say that these "ways around" are true solutions. Therefore, we have designed an
Encapsulated Object-Oriented Programming System (EOOPS) [15] to help solve these problems. EOOPS is designed as an add-on package that could be added to any conventional language.

EOOPS implements an encapsulated form of inheritance that avoids the name conflicts associated with inheritance in "conventional" OOP languages. Thus, multiple inheritance can be used to build the record structure. EOOPS also includes generic methods so that routines such as display_youself can be defined and then easily instantiated for user-defined classes. A proper metaclass is included to facilitate the dynamic building of new classes, for routines such as project and Cartesian product.

EOOPS is not intended solely for the implementation of ROOMS. The problems associated with name conflicts in inheritance exist in any OOP application. Generic routines could also be beneficial to nearly any application. The additional metaclass capabilities are useful in applications such as testing and implementing new variations on class definitions.

5. CONCLUSIONS

The development of ROOMS has shown that it is feasible to organize data from object-oriented applications using a relational database scheme. Since relations can be implemented as a type of collection, any commercially available object-oriented database system that allows objects to be organized into user-specified collections can now be expanded to include all the capabilities of the relational data model. And depending on the I/O capabilities of the language used, ROOMS can also be implemented in any object-oriented language even if a commercial object-oriented database system is not available.

Even though the relational data model is a viable way to organize objects, it is not necessarily the best way for all applications. By including ROOMS as part of an object-oriented database system there is no reason why a single database system could not serve all of our data needs.

REFERENCES