ORR: Object-Relational Rapprochement*

James B. Crigler
Lockheed Martin Corporation
86 S. Cobb Dr.
Marietta, GA 30063
criglerj@acm.org

Ali Orooji†
School of Computer Science
University of Central Florida
Orlando, FL 32816
oorooji@cs.ucf.edu

Abstract

The current fashion in database technology is the eschewing of relational databases in favor of object-oriented databases. C. J. Date and Hugh Darwen have issued a challenge for the industry to turn away from today’s relational database products and to begin to develop and use databases conforming to the formal Relational Model. To meet the storage requirements of modern data, they propose to extend the notion of domains, and they claim that this expanded notion, when combined with the formal relational model, would be more than sufficient for the vast majority of the applications now being moved to object-oriented databases.

This paper makes a brief survey of the relational model, object orientation in programming, and some of the present attempts to combine the two. It then presents ORR, a prototype implementation of a database system that incorporates a version of the expanded notion of domains into a near relational database.

Keywords: Relational databases, Object-oriented systems, Data engineering.

1. Introduction

Modern databases are divided largely into two camps, relational and object oriented. While there is a tremendous amount of data in relational databases, object-oriented databases increase almost daily in their application, with proponents claiming their object-oriented champions are the knights errant to amend the shortcomings of the relational model.

There is, however, a dispute over whether the relational model as originally described has ever actually been implemented. Furthermore, it is claimed that with the imple-mentation of the formal model and expansion of one central idea, the domain, much of today’s fashionable abandonment of relational databases would be obviated.

This paper explores the current state of databases and objects and their respective detractors, and then reports on ORR, an attempt to realize the addition of generalized object-oriented domain capabilities to a relational system. Section 2 briefly describes the relational model and discusses some of the current trends in object-oriented programming and the theory, and in some cases, the lack of theory, behind it. This section also reviews the current state of affairs with respect to object-oriented databases. Section 3 excerpts and comments on the paper on which the ideas implemented here are based, and it summarizes what might be requirements and restrictions for implementing the relational model using a generalized notion of domains that includes objects and inheritance. Section 4 first gives an overview of the database and object systems on which the present work is based. It then describes ORR, a model database that implements some of the notions presented in Section 3. Finally, Section 5 presents some conclusions and gives a few ideas that might be pursued for future research and directions.

2. Relational vs. Object Oriented

2.1. Relational Databases

Codd gave us in the Relational Model (RM) a framework that allows us to reason about data. He proposed five characteristics of relations that should be noted:

1. Each row represents an n-tuple of the relation.
2. The ordering of rows is immaterial.
3. All rows are distinct. In practice, the key of each record is sufficient to identify it uniquely. The key may be one column or a set of columns.
4. The ordering of columns is significant. However, users should be unaware of the actual column ordering used.
for storage, but rather the user should deal with "relationships," i.e., the user deals with columns by name rather than by position.

5. The significance of each column is partially conveyed by labeling it with the name of its corresponding domain.

A foreign key is defined as a column of one table that must have as its value a key present in another table. This leads to the desirable quality called normalization, which reduces redundancy and increases information density.

The relational model also has operations which may be performed on relations:

- **Permutation:** rearranging the columns of a relation.
- **Projection:** gathering the values for particular columns from all records in a relation. The resulting relation may have fewer rows.
- **Join:** concatenating rows of two (not necessarily distinct) tables based on some relationship between them, and eliminating one of any redundant columns.
- **Composition:** concatenating two rows as in join but eliminating all redundant columns.
- **Restriction:** selecting rows. Sometimes called selection, though the SQL SELECT operator is far more than simple restriction. [6, p. 67]

Almost any reasonable reading of Codd’s original paper shows that he preferred domains to be atomic data types in the sense of being simple and non-decomposable, e.g., integer, character string, etc. More exactly, Codd acknowledged the possibility of non-simple domains, but stated his belief that they were undesirable. Date, however, treats a domain as a user-defined data type, opaque to the database system except through the operations defined on the domain. Camps [4] gives a history of Codd and Date’s respective uses of domain, leading up to the rather inclusive (with respect to the original) usage that Date promotes today.

2.2. Object-Oriented Programming Languages

Booch [3] defines object-oriented programming as follows:

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Another suitable definition might be that object-oriented programming languages are those that support both encapsulation and inheritance, and allow the programmer the facility of these features. Derived from inheritance are the common characteristics of substitutability and overloading.

One way to classify object-oriented programming languages is as class-based vs. object-based. Object-based languages do not use classes as their means of organizing objects: An object just is, and it responds to methods that it knows about. Objects in such languages can inherit from other objects and can invoke the methods of other objects. These interrelationships can be made via delegation to external objects or by subsumption, i.e., by containing other objects.

Multiple inheritance is simply the idea that an object may have, in its most natural classification, multiple parents. Classification is a thorny problem in itself — as Abelson and Sussman note [1, p. 200, footnote 52]:

Developing a useful, general framework for expressing relations among different types of entities (what philosophers call “ontology”) seems intractably difficult. The main difference between the confusion that existed ten years ago and the confusion that exists now is that now a variety of inadequate ontological theories have been embodied in a plethora of correspondingly inadequate programming languages. For example, much of the complexity of object-oriented programming languages — and the subtle and confusing differences among contemporary object-oriented languages — centers on the treatment of generic operations on interrelated types.

2.3. Objects and Databases

Three trends are visible in modern databases:

- **Object databases** were originally conceived as a way to provide persistent storage for object-oriented programming languages such as C++. Interestingly, marketing material from Object Design, the maker of ObjectStore, indicated fairly recently that "full join" was being added to ObjectStore's supported features.

- **Object-relational databases** attempt to merge the notions of objects and relations, generally providing support for a dialect of SQL, but treating attributes, tuples and even relations as objects.

- **Object-oriented access to relational databases** is an effort to provide object-oriented programming facility to the treatment of data still housed in relational databases.

3. A Summary of "The Third Manifesto"

The primary focus of "The Third Manifesto" [7] by Darwen and Date is to codify what must and must not be in databases of both relational and object-oriented ilk, and to provide additional very strong suggestions for both. In this
section, some of the requirements (which they call “prescriptions”) and prohibitions (which they call “proscriptions”) are listed. We have also included, with most items in all four categories of requirements and prohibitions, comments about how the design/implementation of ORR follows or does not follow the guideline and comments about how the item fits with Codd’s database theory or with current practice. (Inclusion of comments about ORR represents a classic chicken-and-egg problem, since ORR has not yet been described; please reference back to this section after reading later sections to clarify anything left murky on first reading.)

While Darwen and Date use the term “Other Orthogonal” rather than “Object-Oriented,” it is clear that object orientation is the target on which they have set their sights.

The requirements and prohibitions are stated in Darwen and Date’s paper in terms of a database language which they call D (D is not meant to be any particular programming language). Many of the features of D, notably those intended to support transaction processing and security policy, are not applicable to the data type modeling at the heart of ORR, and these are noted.

The requirements and prohibitions listed in this section use the same numbering as in [7] for easier cross referencing. Please note that not all requirements/prohibitions are included in this paper.

3.1. Relational Requirements

1. D must support opaque data types of arbitrary complexity, manipulable only via operators defined on the type. These data types are termed domains. Values that are members of a domain may be referred to as scalar values. A method must be provided for constructing values in a domain.

This is a major generalization of the original relational model, in which all domains were primitive, e.g., numeric, string, etc. As far as it is stated by this requirement, this prescription is fulfilled by many existing commercial and non-commercial products, e.g., PostgreSQL. ORR fulfills this requirement in its selection of the object model.

2. The equality operator must be defined for every domain. Equality for ORR primitives is provided by Scheme — in fact, Scheme provides four distinct equality operators. Since this requirement carries through to object-oriented databases, it is appropriate to note that the meaning of equality in an object-oriented context is a fundamental problem (this problem is described in Appendix A). Equality for ORR objects must be provided by the objects, since it is generally non-trivial.

3. A tuple is a set of ordered triples \((A, V, v)\), where \(A\) is the name of an attribute, \(V\) is the domain of the attribute and \(v\) is the value of the attribute. The set of ordered pairs \((A, V)\) is the heading of the tuple. All the attribute names must be distinct.

This is a departure from the original relational model: Codd used the domain name as the attribute name, though he sometimes attached a role prefix or suffix to avoid ambiguity. In ORR, a variable which contains a list that might be a tuple does not have a header associated with it directly, but any attempt to put such a list as a tuple into a table will be tested by up to four different integrity rules.

10. A relation consists of a heading (identical to a tuple heading) and a body. The body is a set of tuples all having the relation’s heading as their common heading. Every ORR table is described by another table, called its descriptor, which serves the purpose of the header, and imposes additional constraint checks.

11. Scalar variables in D must be typed; when they are created, they must be initialized, either with a given value or with a default. Since ORR variables are Scheme variables, typing is not present on variables, but typing is present on values.

17. D must provide operators to create and destroy domains, variables and integrity constraints; explicitly created domains, variables and constraints must be named; every base relation variable must have at least one candidate key, explicitly specified.

Domains in ORR are the domains of the underlying relational system, and domains may be added to and removed from the database’s domain table; variables are Scheme variables; integrity constraints apply to a value in a row in a base relation in three ways; primitive data types are named, and object types must be registered to be useful in ORR. ORR supports only explicitly specified keys.

18. The relational algebra must be expressible in D.

By “relational algebra,” the authors mean that D must support the following operators: restrict, project, product, union, intersection, difference, join, and divide. This is a superset of the operations that Codd originally defined. Generally, the operations of the relational algebra can be implemented in any algorithmic language.

23. Logical expressions may be used as integrity constraints on domains, attributes, relations and databases.

ORR integrity rules are logical-valued procedures. They are applied to domains, attributes and rows. The integrity rule for a row (called the user integrity rule) might be formulated to cover relations and perhaps even an entire database, though it is applied when a row is inserted or updated in a table.

25. Every database variable must have a self-describing catalog, which is a set of relation variables, to which it is possible to assign relation variables.
3.2. Relational Prohibitions

2. D must not contain a way to refer to tuples based on any ordering.
   This is, access to individual records must be by key, not
dependent on the ordering of records in the table, indexing, etc.

3. Duplicate rows in a relation are forbidden.
   Since rows in a relation must be distinguishable by key
   anyway, this prohibition is automatically fulfilled.

4. Every attribute of every row must have a value from the
   attribute's declared domain.
   Of course, some attributes may have their respective do-
   main's default value, but this item prohibits missing values
   (i.e., nulls). ORR fulfills this by the enforcement of
   integrity rules for the domain and attribute.

5. D must be completely separated from any notion of how
   the data is stored. Any storage specification must not be
   expressible in D.
   This seems to imply that there must be a separate data
   specification language for the sake of preventing access
   to the underlying structure of a database, table or row.

6. There shall be no tuple-at-a-time operations in D.
   The remarks that accompanied this prohibition seemed
   contradictory: Statements in D that perform, e.g., in-
   sertion, must operate insert relations, i.e., perform a rel-
   ational union operation. No mechanism is specified,
   however, to put a tuple into a relation that is to be in-
   serted into another.

10. D must not be called SQL.

3.3. Object-Oriented Requirements

2. D shall use some generally accepted model of single in-
   heritance.
   Section 4.1.2 describes in detail how inheritance is han-
   dled in ORR.

3. D may permit multiple inheritance.

4. D must be computationally complete.
   The underlying language for ORR, Scheme, is computa-
   tionally complete.

3.4. Object-Oriented Prohibitions

1. Relation variables are not domains.
   This is the opposite of PostgreSQL, in which object do-
   mains are relation variables. These are distinct from
   user-defined data types which may be the value of an at-
   tribute. PostgreSQL defines inheritance specifically on
   relations.

2. No scalar value shall possess an identity that is distinct
   from its values.

   This means, specifically, there must be no object IDs.
   This distinguishes Date and Darwen's ideas from Ob-
   jectStore (which assigns an “ID” which is a virtual ma-
   chine address) [10], and it flies in the face of “The
   Object-Oriented Database System Manifesto” (in which
   an object must have an identity that is distinct from its
   value) [2].

4. ORR

4.1. Scheme and Slib Components for Databases
   and Objects

4.1.1. Scheme. The implementation language of ORR is
   Scheme, a Lisp dialect of small footprint and clear seman-
   tics. Slib [8], the portable Scheme library made by many
   individuals and distributed freely on the Internet, provides
   much of the needed infrastructure for ORR. This includes
   components for database file manipulation, database han-
   dling at the (conceptual) relational level, and several com-
   ponents for object-oriented programming. Since these ele-
   ments are available as source code, it is possible, and some-
   times easy, to adapt the components to any appropriate use.

   The particular version of the Slib components used in
   ORR originated in version 2.6 of Slib. Subsequent releases
   of Slib (e.g., version 2c5) have changed some of the com-
   ponents.

4.1.2. Objects with object.scm. The object compo-
   nent chosen for ORR is a classless object system. That is,
   to construct a new object, there is no class object which
   knows how to make one, there is simply a user-defined pro-
   cedure that creates a closure consisting of the appropriate
   data and methods. “Generic” methods are declared at the
   outermost user level, then they are instantiated within an
   object creation procedure. When created, generic methods
   may specify a default procedure which will be invoked if the
   generic is not found to be defined for its first argument —
   if a generic is defined multiple times, each time defaulting
   to the previous definition, the stack of previous definitions
   will be unwound until some appropriate procedure is found.
   This problem is further considered in Section 4.2.5.

   Methods are inherited by “child” objects from their “par-
   ents,” but a parent's data is not directly visible to its chil-
   dren. A parent's methods may be overridden by the child
   and/or called by the child as required. A child may also
   disown a parent's method(s) so that they are not visible to
   the child's clientele, but the child may still invoke the dis-
   owned procedure(s) via the “owned” parent to operate on
   the parent's data.

4.1.3. Relational Manipulations with zams.scm. The
   relational layer of the database, zams.scm, represents
a partial implementation of the relational model. To be able to save data to disk, \texttt{rcdm.scm} requires that a storage layer with a specific interface be provided so that actual data may be manipulated. Slib includes a lower-level layer which can be used for this purpose. At the \texttt{rcdm.scm} level, each record in a relation is a list of attributes. In the present implementation, the row/list must have the values of the attributes in storage order.

The data definition language requires the user to specify the column number to which an attribute will be assigned, and rows must be inserted with attributes in order, but subsequent references to the attribute by name are resolved by \texttt{rcdm.scm}. Each attribute must belong to a \texttt{domain}, which is a basic data type subject to additional constraints. In addition, individual attributes are subject to column integrity rules, and columns which are foreign keys are checked to be sure the foreign item exists at the time of insertion or update. Finally, each row is subject in its entirety to a "user" integrity rule.

With respect to accessing a database, \texttt{rcdm.scm} provides procedures for accumulating rows from a table, selecting a single row, iterating over rows for side effects, inserting rows singly or in groups, updating rows singly or in groups (using the key to determine which row), and deleting rows.

A \texttt{rcdm.scm} database has several distinguished tables:

- \texttt{catalog-data} contains the list of all the tables in the database, including itself.
- \texttt{domains-data} contains the list of domains known to the database, along with a description of how the domains are related to the basic types.
- \texttt{catalog-desc} gives the definitions of the catalog table's attributes.
- \texttt{domains-desc} gives the definitions of the attributes of the domains table.
- \texttt{columns} describes descriptor tables, including itself.

To add a table to the database, there must first be a descriptor of its columns. For each column in the desired table, the descriptor table must know its column number, whether the column participates in the key, the column's name, the column's integrity rule, and the name of the domain to which all members of the column must belong. When this descriptor is complete, the new data table may be created and operations may begin on it.

### 4.1.4. Database Storage with \texttt{alistab.scm}

The storage layer for ORR, \texttt{alistab.scm}, stores all the data for a database in a single file, using the textual representation of an association list of association lists. \texttt{\lambda}-expressions used as integrity rules are stored in the form in which they would be input as program text to the interpreter. For example, here is the standard \texttt{domains-data} row for unsigned integer:

\begin{verbatim}
(uint ()
  (lambda (x)
    (and (number? x)
      (integer? x)
      (not (negative? x))))
  ()
)
\end{verbatim}

When this S-expression is read and passed to \texttt{rcdm.scm}, the higher level database evaluates the \texttt{lambda} expression which is the third element of the list and the resulting procedure is ready to check the domain of any unsigned integer columns that come along.

### 4.2. ORR: The System

The purpose of implementing ORR was to discover whether the relational model could withstand the onslaught of objects if the notion of domain were suitably generalized. With regard to adaptation of the Slib components, the goal was to perform the smallest possible set of alterations to them.

The changes required to implement the addition of objects to a model relational database was, in fact, surprisingly small. All the implementation code from Slib that changed for ORR was in the storage module, \texttt{alistab.scm}. The changes there were fairly pervasive, since a large new functionality had to be added to it.

The steps required to implement ORR fell into four categories (listed below and described in the following subsections):

- object storage
- object recovery
- \texttt{alistab.scm} changes
- method name registry

The system loads \texttt{object.scm} and \texttt{rcdm.scm} unaltered from the Slib library directory, then loads the altered \texttt{alistab.scm} module.

#### 4.2.1. Object Storage

Since the object system creates closures rather than simple data structures, a way of storing values along with an indication of their type was required. The method chosen was to create a generic procedure, called \texttt{storable-form}, which when called for an object would return the data of the object in a form which could be stored in the database. The storable form itself is simply a Scheme vector of the data items in the object with the name of the object as the first member of the vector. (In Scheme, a vector need not be homogeneous.) In the presence of inheritance, it is the responsibility of the derived object to obtain (by calling \texttt{storable-form}) the values from any base objects from which it inherits behavior and to store them in the vector in some appropriate fashion. The derived object has full discretion over how to place the base
data in its vector, whether “in line” with other derived members or with the base object’s data vector as a single member of the storable form.

4.2.2. Object Recovery. For the sake of being able to reconstruct ORR values that are stored in the database, and to prevent the low level database from being unextendable in the sense of being unable to add new types to the system, a type registry was devised. The type registry is an object implementing a list of (name, re-creator) pairs. The re-creator is a function that knows how to transform a value returned from storable-form back into an object that is equal? to the original.

Lookup Function: The lookup utility takes as parameters the name to look for in the table and a procedure to apply to the result. Assoc is used for the search, since it is the most general association list lookup function (it uses equal? as the comparison predicate)—this allows the type constructor to use any form as the type identifier. Proc is applied to return something from the association.

Type Registration Function: Type insertion consists of just looking for name in the table, and adding the reconstruction procedure if the name is not found. #t or #f is returned if the type is added or not added, respectively.

Type Re-creation Function: Returning the recreation function just requires looking up the name and returning the function that was registered.

4.2.3. Type Name Query. This process (registering a type) involves determining whether or not the type has already been registered.

4.2.4. Changes to alistab.scm. Two fundamental changes were made to alistab.scm for ORR. As mentioned in Section 4.1.4, the key fields in a row of a table maintained by alistab.scm are kept in the initial columns of the row. What was not mentioned there, however, was that for ease of reference, the members of the key are kept in a Scheme vector. ORR changes this to a list to allow easier uniform reference to the members of the key with respect to transformation for an object member of the key between the in-memory object representation and its storable form.

The second change was to make appropriate lookup in the type registry table and recovery of objects from it. This had to be done for each record whenever the object value is required, e.g.,

• for comparison with another object during key lookup,
• for reconstituting a row to be passed back to the higher (relational) level of the database.

4.2.5. Method Registry. If a module needs to register a method, say resolve, the normal object syntax would be:

(def resolve
 (make-generic-method))

But if, say, both a symbolic algebra system and a catalog of carpet cleaning products wanted to define resolve, it might not be reasonable to expect them to know about each other. A database system, however, tends to be heterogeneous almost by definition, so it is easily conceived that these disparate definitions be found together, making multiple invocations of make-generic-method for a given name not only likely, but virtually certain.

To prevent multiple definitions of generic methods so that there is only a single level of search made for methods, rather than defaulting through layers of multiply-defined generic methods, all generic methods should be created through the interface provided by the generic-method-registry.

Registering a method or predicate simply requires checking to see whether it already appears in the list of registered symbols, and if it is not there, adding it to the list and evaluating the definition in the top-level environment. Since these procedures are registered via eval, they are passed into register! as symbols, which guarantees linear lookup in the table.

5. Conclusions and Future Directions

What is required for the incorporation of generalized objects into relational databases?
• Variable-sized storage for individual attributes.
• A storage layer that can be taught how to convert an object to and from its storable form.
• A relational layer that makes no assumptions about its data aside from primitive data.

How might ORR be modified to further model the goals of “The Third Manifesto?”
• Completion of the relational algebra.
• Addition of the notion of transactions. This might be modeled in ORR using Scheme’s continuations, rather than regular procedure calls, as entry points into the relational layer. Modeling transactions in a more real-world way would require an underlying Scheme system that was network-aware and multi-threadable.
• Implementation of some security model.
• Replacement of the textual S-expression storage layer, in which the entire database is kept in memory while it is open, with a storage layer that uses disk access, paging, etc.
What is beyond the reasonable capabilities of the ORR infrastructure? That is, when should ORR be abandoned for a next-level prototype implemented in a systems programming language?
- When it reaches a size that makes the interpreted implementation impractical.
- If it ever becomes desirable to turn ORR into a product for distribution other than research.

What new information has come from Date and Darwen on this proposal since 1995? Most of the material seems to have been in the nature of repetitions of their complaint of the relational model having never been implemented, of the condemnation of SQL, and of noting that the object-oriented databases have “finally” implemented domains (though the notion of just what a domain is only recently reached its final form).

The most recent development, however, is the publication of Foundation for Object/Relational Databases: The Third Manifesto [5] by Darwen and Date in June 1998. Hopefully, it has answered the questions that remained open since the original publication of the Manifesto.

Is there hope for the relational model? The relational model is alive and well. Relational databases still hold a market position that far exceeds all their competitors, even though, according to Date, they have never implemented the model. Perhaps if database vendors, of whatever ilk, take up the challenge that Date and Darwen have given us, we will find out.

A. Equality in Object Systems

One of the hairier design problems in object systems is how to handle equality. For instance, suppose you have objects consisting of 2- and 3-dimensional points. Now if the equality query goes to a 3D point, that object knows it is a 3D point and can ask its argument whether it is also a 3D point. Thus if a 2D point is passed as the equality’s second operand, the equality query is false.

But suppose the object implementing the equality operator that gets called is the 2D point. A 3D point is also a 2D point, so in the terminology shown, (point? a-3d-point) is true, and the 2D point will proceed to compare its x and y parts with those of its argument.

There are essentially two approaches to this problem, which might be called restrictive and permissive. In the restrictive approach the equality comparison is made both ways, and is false if either comparison is false. The permissive approach is, predictably, to return true if either comparison is true.

This problem cannot, in general, be implemented in the objects involved, but must be solved at the application level. The modified alistab.scm takes the permissive route.

References