Producing Relational Database Schemata from an Object Oriented Design

P. Fitsilis V. Gerogiannis A. Kameas G. Pavlides

INTRASOFT SA
2 Adrianiou st.
Athens, Greece

Dept. of Mathematics
Univ. of Patras
Patras, Greece

Dept. of Computer Engineering
& Informatics, Univ. of Patras
Patras, Greece

Abstract

Although Object-Oriented Database Management Systems provide a direct mapping between problem domain and the way data are stored, they are not yet as popular as Relational Database Management Systems. Storing objects in a relational database requires fitting them into tables. In this paper, a straightforward methodology for producing a Relational Database schema from an Object-Oriented Design is presented. The proposed approach provides a solution for mapping all object oriented constructions to tables. The strongest features of the approach are that it yields normalized schemata and can be easily automated. Finally, a high-level algorithm and a very simple example to illustrate the approach and its benefits, are given.

1 Introduction

Applications implemented with Object-Oriented (OO) programming languages require persistent objects storage. Dealing with objects persistence consitutes one major decision among three differing approaches, each of which could be attractive under certain circumstances [5,10]:

- *flat file management*, which offers rudimentary file handling and storing capabilities
- *relational database management systems*, used to built relational databases
- *object-oriented database management systems*, which, based on the OO paradigm, provide a way for storing objects without any translation between storage data structures and program data structures

Its up to the designer to decide which approach fits in the problem domain and to the specific application. Flat file management is usually used in an ad hoc way. It can offer a quick and useful solution when the size and complexity of the application are rather small. Some programming languages are offering objects persistence by using flat file management schemata, an approach that has been proved useful and conceptually simple. However, as the size of the application becomes larger, the complexity of storing objects in flat files increases rapidly. For applications of this size it is almost impossible to use such an organization of data.

Relational Databases (RDBs) are a mature technology, having a very sound underlying theory. Very powerful products are in use and the developers are familiar with this technology and the every day problems.

The recently developed Object-Oriented Database Management Systems (OODBMS) seem to offer a very attractive solution for modelling both the problem and the solution domains, especially when they are combined with OO programming languages extended with facilities for persistent objects handling. The problem with OODBMS is that although this technology is emerging, these systems are not yet mature and hence not popular. In addition, there seems to be a lack of general consensus as to what constitutes an OODBMS [1,2]. This has led to the development of a number of different variations of the OO data modelling paradigm.

Therefore, the Relational Database Management Systems (RDBMS) seem to be the best solution available, since they are based both on a sound theory and on a mature technology. At the same time RDBMS vendors add to their products capabilities for storing objects. Until now, however, there exists no standard, not even consensus to this subject [4]. Consequently, techniques for RDB design based on OO analysis have evolved and experience has shown that they are good and superior to other techniques [12].

Even though RDBs are a commonly used implementation solution, there are some issues that come up when implementing an OO model; these involve database design, the use of triggers within the conceptual model, and the interface to the application code. To these problems, various authors suggest general approaches [6,9,11].

Furthermore, an approach for mapping an Object-Oriented Design (OOD) to an RD schema must consider the normalization principle: the concept that each piece of information should be stored in exactly one place [7] (the simplest yet most important goal in database design). This discipline eliminates redundancy, simplifies the process of updating the database, facilitates the maintenance of database integrity and reduces the storage requirements. Nevertheless, achieving this goal is not easy. The degree of elimination of data redundancy is defined in a range of normal forms:

- *First normal form (1NF):* Each attribute represents an atomic value
- *Second normal form (2NF):* Tables are in 1NF, and each attribute depends upon the key
- *Third normal form (3NF):* Tables are in 2NF, and no attribute represents a fact about another attribute
- *Fourth normal form (4NF):* Tables are in 3NF, and two or more non-key attributes do not always map to another non-key attribute
- *Fifth normal form (5NF):* Tables are in 4NF, and two or more non-key attributes do not always map to another non-key attribute following a join operation

The disadvantage of normalization is the large number of resulting tables, which usually leads to cumbersome coding and to performance degradation. In addition, the match between problem-domain constructs and tables is decreased.

In this paper, a practical approach is presented to the problem of implementing an RDB schema out of an OOD; this solution has been applied in ORIENT CASE Tool [8]. The advantages of the proposed solution are that a straightforward mapping from OOD to RDB implementation is provided, the proposed methodology can be fully automated (in fact, a high-level algorithm is also included) and its implementation yields a normalized RDB schema. It is expected that the resulting database design will normally end up in third normal form.

After a short introduction to OOD, the methodology that can be used for mapping an OOD to a RDB schema is described and a high-level algorithm for the development of the initial RDB schema is presented. The methodology is explained with the brief presentation of an example, and the paper concludes with the future research directions of the authors.

2 Overview of Object-Orienter Design

In Object-Oriented Design there exist four major components which have to be defined [5]:

- Problem domain component
- Human interaction component
- Task management component
- Data management component

For the definition of the problem domain component, the analysis results are initially used. The analysis model is composed of several layers, namely Subject layer, Class&Object layer (Class, Class&Object), Structure layer (Generalization-Specialization structure, Whole-Part structure), Attribute layer (Attributes, Instance connection) and Service layer (Services, Service connection). The analysis results are reviewed, challenged and extended in order to become more detailed and specific, and form the OOD problem domain component.

The strategy to design the human interaction component consists of the following steps: user classification, user's task scenarios description, command hierarchy design, detailed interaction design, prototyping and user interface classes design.

In order to define the task management component, the event/driven tasks, the clock driven tasks, the priority and critical tasks, together with a co-ordinator, to monitor each task.

The data management component is used to isolate any impact that the data management schema may have on the complete design model. The other OOD components are designed transparently based on the data management schema.

OODB development is an emerging technology. Consequently, its theory is not so well defined, products are not mature yet, while the developers are not fully aware of this technology. The main advantage is that the mapping between the problem domain and the way the data are stored is direct. Until now the object oriented approach has been tested for relatively small systems. Now it has to be proved useful and applicable in large scale system development.

3 Mapping the OOD model to an RDB design

In an RDB, information is stored in tables of primitive types. In the OO paradigm data resides within objects either as primitive types or as complex structures. This results to some problems when one is trying to store objects in an RDB. For this reason a transformation is needed in order to map the object information structure into a table-oriented structure. This problem is often referred to as the impendance problem [11]. The same problem occurs when a program has too rich a set of user-defined types, which have to be transformed to primitive data types of the DBMS.

The impendance problem yields to another problem: it creates a strong coupling between the application and the DBMS. The database schema should be independent from the application code and changes in the application should affect as little as possible the database. Another problem relates to the expression and storage of the inheritance relation in the database.

The following list summarizes the main problems of mapping an OOD to a relational implementation of a database:

- Mapping of the instance connections to tables
- Mapping generalization-specialization structures to tables
- Mapping whole-part structures to tables

3.1 Storing Classes&Objects into a relational database

The first thing to do is to decide which Classes&Objects and which variables of each class must be stored in the database. Each one of these classes will be represented by at least one database table. To store a Class&Object in the database, the following rules must be followed:

- Assign one table to each Class&Object
- Each primitive attribute will become one column in the table. If the attribute is complex an additional table for this attribute is added or it is split over several columns in the table of the class.
- The primary key column will be the unique instance identifier. This identifier should not affect the user and it will be machine generated. Therefore, each instance of a Class&Object will be represented by a row in this table
- Each instance relation with cardinality greater than 1 will become a new table. This new table will connect the tables representing the objects that are to be associated. The primary key of these tables can be used as the key of the new table.

3.2 Modelling the generalization-specialization structure

The objects that should be persistent will be mapped onto tables in the database. There exist two different approaches to solve this problem [5]:

1. The inherited attributes are copied to all tables that represent the descendant classes. No table will represent the abstract class.
2. The abstract class is in one table of its own, to which the tables of the descendant classes refer.

Both alternatives have advantages and disadvantages. The first approach introduces redundancy since the attributes of the superclass are inherited by the subclasses. The problem is magnified when the generalization-specialization structure has more than two levels which is far than unusual. Furthermore, the introduction of redundancy leads to integrity and consistency problems, since a change to a table has to be propagated to all related tables. However this approach is normally faster since no joining, or searching in several tables is necessary to get information about one object.

The second alternative does not result in integrity problems introduced by replication of data, but searching for object attributes is more time consuming, especially when the generalization-specialization structure is expanded in more than two levels.

In our approach, the second alternative is chosen, in order to prevent the logical decomposition of the subject matter.

3.3 Modelling the whole-part structure

A whole-part structure is one of the basic methods of organization that pervade all human thinking. This structure can be viewed as an aggregation relation. Each "whole" is composed of a number of "parts". Also, each whole-part relation is marked with an amount or range, indicating the number of parts that a whole may have and vice versa, at any given moment in time.

It is obvious that a whole-part relation resembles the instance connection. For this reason the whole part relation can be mapped as an instance relation. Therefore, each whole-part relation with cardinality greater than 1 will become a new table. This new table will connect the tables representing the objects that are to be associated. The set of the primary keys of these tables can be used as the key of the new table.

4 Creating the initial RDB Model

In order to create the relational database design from an OODB design, first Classes&Objects must be handled and the generalization-specialization structure must be processed (table 1) and subsequently the whole-part structure and the instance connections must be checked in order for relationships to be created (table 2).

However, database normalization introduces issues of low performance, because the way the database is used is not taken into consideration.

4.1 An example

In this subsection an example showing the way objects and relations will be stored in an RDB is presented. The RDB models a small part of a public domain organization which gives licences to vehicles. The tables produced for mapping of the Classes&Objects as well as the dependencies between them are shown in figure 2, while in Figure 1 the OOD of the model is depicted.

This part has four Classes&Objects (*Organization*, *ClerkPerson*, *OwnerPerson*, and *ClerkOwner*) and one Class (*Person*). The Class&Object *Organization* belongs to a Whole-Part structure with Class&Object *ClerkPerson*. A generalization-specialization structure exists between Class *Person* and Classes&Objects *ClerkPerson*, *OwnerPerson*, and *ClerkOwner*. All attributes are considered as primitive types except attribute *Address* which is considered as composite, with sub-attributes *StreetName*, *Number*, *PostCode*, and *City*.

The definition of each table includes the following attributes:

ORGANIZATION

ORG_OID	NAME
MANAGER	TELEPHONE

The address attribute is a composite attribute and for this reason it is not included in the above table. Instead, the following table is created:

```
/* Handle Classes&Objects */

For all persistent Classes&Objects
        Create a table with the same name as Class&Object Name;
        Create an attribute OID(Object ID);
        Mark this attribute as primary key;
        For all attributes of Class&Object
                If an attribute is primitive
                        Create a table column with the same name;
                If attribute is composite /* recursive - a composite attribute */
                        /* may contain other composite attributes */
                        Create a new table having as a name
                            the Class&Object_name+attribute_type;
                        Create an OID attribute same as parent's table;
                        Mark it as primary key;
                        Mark it as foreign key;
                        For all attributes of Class&Object
                            If an attribute is primitive
                                Create a table column with the same name;
                            If attribute is composite
                                Make the recursion on composite attributes;

/* Up to this point all tables corresponding to Class&Object */
/* and tables for composite attributes have been created */

/* Process generalization-specialization structure */

For all the generalization-specialization structures
        Find root of the structure;
        Traverse the structure depth first;
        For all children Class&Object
                Create a new column with the parent OID name;
                Mark this column as a foreign key;
```

Table 1: Part 1 of high-level algorithm

ORGANIZATION-address

ORG_OID	STREETNAME
NUMBER	POSTCODE
CITY	

The Class *Person* does not have instances and for this reason no new table is created, but the information of this Class is included in its descendent Classes&Objects. The table which corresponds to the Class&Object *ClerkPerson* is:

ClerkPerson

CLPR_OID	LEGALNAME
USERNAME	AUTHORIZATION
BEGINDATE	ENDDATE

The attribute *Address* is handled in the same way as attribute *Address* of Class&Object *Organization*.

OwnerPerson

OWPR_OID	LEGALNAME	TELEPHONE

The Class&Object *ClerkOwner* does not have attributes of its own, except those inherited from Classes&Objects *ClerkPerson* and *OwnerPerson*. The table corresponding to this object is shown below.

ClerkOwner

CLOW_OID	CLPR_OID	OWPR_OID

Finally, the whole-part relation between *Organization* and *ClerkPerson* is modelled with the following table:

WP-ORGANIZATION-CLERKPERSON

ORG_OID	CLPR_OID

The fields ORG_OID, CLPR_OID, OWPR_OID and CLOW_OID are automatically- generated object

```
/* Check the Whole Part structure for creating relationships */

For all Whole Part relationships
    If the Whole Part relation cardinality is many to many
        Create a new table with columns the primary keys of both tables;
    If the Whole Part relation cardinality is one to many
        Create a new column to the table with many cardinality,
            which corresponds to the OID of the table with the one cardinality;
    If the Whole Part relation cardinality is one to one
        Create a new column to one of the tables,
            which will contain the other table OID;

/* Check the instance connections for creating relationships */

For all instance connections
    If the instance connection cardinality is many to many
        Create a new table with columns the primary keys of both tables;
    If the instance connection cardinality is one to many
        Create a new column to the table with many cardinality,
            which corresponds to the OID of the table with the one cardinality;
    If the instance connection cardinality is one to one
        Create a new column to one of the tables,
            which will contain the other table OID;
```

Table 2: Part 2 of high-level algorithm

identifiers. The database schema produced using an OOD will normally end up in third normal form. A database design is in third normal form, if and only if a row consists of a unique object identifier together with a number of mutually independent attributes. Furthermore, since OO models are modelling reality, objects are identified uniquely and attributes are assigned where they naturally belong. Therefore, since reality is normalized as such, a good object model will also be normalized.

5 Conclusions

In this paper, the mapping from an OOD to an RDB schema has been presented. The OOD method used is the one introduced by Coad/Yourdon [5], which is one of the most popular OO methodologies. The mapping from this method to RDB design seems to be straightforward and the resulting RDB schema is in third normal form. Guidelines for handling the persistent object, storing the inheritance structure, the whole-part structure and the instance connection are presented, together with a high-level algorithm that derives the initial RDB schema. The presented approach can be automated and has been implemented in the ORIENT CASE tool [8].

We believe that RDBs will continue to be dominant, despite the evolutions in OO development, because of the impendance of changing existing database models (Hierarchical, Network). For homogeneity reasons, in the course of time the alternative approaches for objects persistence handling will converge rather than differentiate. Consequently, our future research direction will focus on the adjustment of RDBs to offer an OO interface, since the complete replacement of RDBs by OODBs seem higly unprobable. In the context of this research, we expect to have soon result on the efficiency of the presented methodology when applied to small and medium-size databases.

References

[1] M. Atkinson, F. Bancilhon and D. DeWitt, *The Object-oriented Database system manifesto.* Proceedings of ACM/SIGMOD International Conference on Management of Data, 1990.

[2] D. Beech, *Groundwork for an Object Database Model.* In *Reasearch Directions in Object-Oriented Programming* (B. Shriver and D. Wegner D - eds), MIT Press, 1987, pp 317-354.

[3] M. R. Blaha, W. J. Premerlani and J. E. Rumbaugh, *Relational database design using an object-oriented methodology.* Communications of the ACM, 31(4), 1988.

[4] T. Bloom and S. B. Zdonik, *Issues in the design of object-oriented database programming lanquages.* Proceeding of OOPSLA 87, Orlando, USA, 1987.

[5] P. Coad and E. Yourdon, *Object-Oriented Design, Second Edition.* Yourdon Press, 1991.

[6] M. K. Crowe, *Object systems over relational databases.* Information and Software Technology, Vol.35, No. 8, 1993.

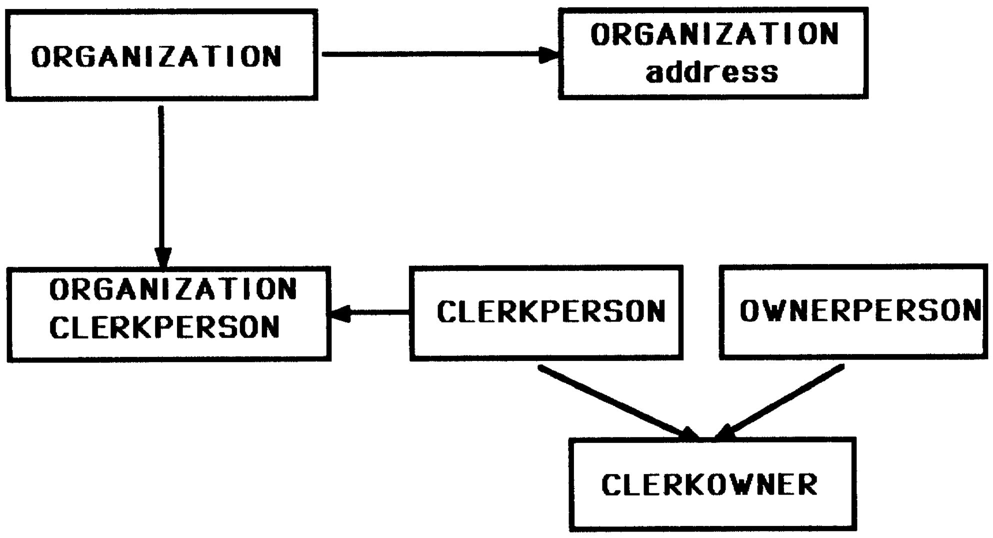

Figure 1: Dependencies among the RDB schema tables

[7] C. J. Date, *An Introduction to Database Systems, Vol. 1.* Addison-Wesley, 1986.

[8] P. Fitsilis et al, *ORIENT User's Guide.* INTRA-SOFT 1993.

[9] I. Jacobson, *Object-Oriented Software Engineering.* Addison- Wesley, 1992.

[10] M. Loomis, *Making objects persistent.* Journal of Object-Oriented Programming, October 1993.

[11] J. Martin and J. Odell, *Object-Oriented Analysis & Design.* Prentice-Hall, 1992.

[12] W. J. Premerlani, M. R. Blaha, J. E. Rumbaugh and T. A. Varwig, *An object oriented relational database.* Communications of the ACM, 33(11), 1990.

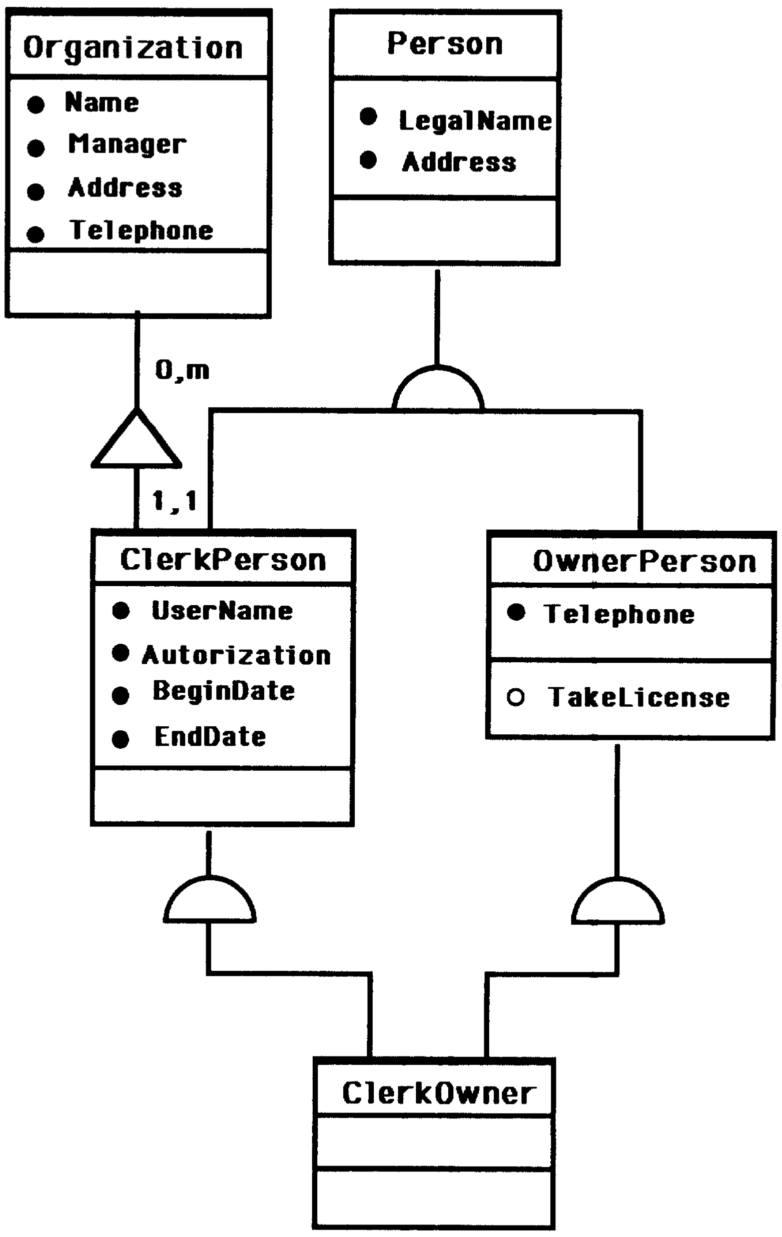

Figure 2: OOD of the model used in the example