# Extraction of Object-Oriented Structures from Existing Relational Databases

Shekar Ramanathan
Julia Hodges
Department of Computer Science
Mississippi State University

## Abstract

Due to the wide use of object-oriented technology in software development and the existence of many relational databases, reverse engineering of relational schemas to object-oriented schemas is gaining in interest. One of the major problems with existing approaches for this schema mapping is that they fail to take into consideration many modern relational database design alternatives (e.g., use of binary data to store multiple-valued attributes). This paper presents a schema mapping procedure that can be applied on existing relational databases without changing their schema. The procedure maps a relational schema that is at least in 2NF into an object-oriented schema by taking into consideration various types of relational database design optimizations.

## 1 Introduction

Object-oriented technology is being widely used in software development these days. The object-oriented data model is growing in popularity in the area of database development. Consequently, there is also a growing interest in re-engineering relational databases to object-oriented databases. Reverse engineering of relational databases is concerned with the task of extracting structures corresponding to a particular conceptual model such as the entity-relationship (ER) model, the extended entity-relationship (EER) model, the object-oriented (OO) model, and so on [4]. Due to current investments in relational technology, a common motivation for reverse engineering is for the purposes of superimposing an object model on a relational model.

In this paper, we present an approach for extracting the structural portion of an object-oriented schema from an existing relational schema. We distinguish our work from other work reported in the literature later on in this paper. The major contributions of this paper to the research in reverse engineering of relational schemas to object-oriented schemas are:

- a schema mapping procedure that does not require the existing relational database to be modified
- ability to cope with database optimizations involving the use of BLOBs (binary large objects), horizontal and vertical partitioning, and relations in second normal form (2NF)
- demonstrate the use of techniques developed for knowledge discovery from databases

The rest of this paper is organized as follows. The next section contains an outline of the major steps for carrying out schema mapping. The main schema mapping process is described in section 3. Section 4 contains a brief review of the literature in related areas. Finally, the conclusions of this paper are given in section 5.

## 2 The Overall Process

Formally, a *schema mapping* of a source schema ${S_1}^{M_1}$ defined on a data model $M_1$ to a target schema ${S_2}^{M_2}$ defined on a model $M_2$ may be defined as a transformation $T$ such that $T({S_1}^{M_1}) = {S_2}^{M_2}$. The transformations are usually a collection of rules that map concepts from the source data model to the target data model.

The first task of mapping from a relational to an object-oriented schema is to identify what object-oriented constructs we are interested in extracting from the relational schema. Once they are specified, then the actual procedure for carrying out the mapping can be specified to target only the constructs of interest. In keeping with the standard practice in the database reverse engineering literature, when we talk about extracting the object-oriented schema, we mean extracting the structural portion of the object-oriented model. The constructs of interest in the object-oriented model are:

1. *Object Classes.* Those relations that correspond to object-classes must be identified.

2. *Relationships.* There are mainly three types of relationships that can be represented in an object model. They are associations, generalizations-specializations, and aggregations [10].

   (a) *Associations.* Since the object model allows associations to be modelled as classes, we must either establish a simple association between two object classes or identify relationships where the associations are modelled as classes.

   (b) *Inheritance.* Inheritance structures capture the generalization-specialization relationships between object classes that have been identified so far.

   (c) *Aggregation.* The aggregation relationship models the composition of one object with other objects. Such complex objects must be identified. The difference between aggregation and association is that the former involves existence dependence of the sub-object on the whole object.

## 2.1 Assumptions

The procedure presented in this paper takes into consideration relations that are not in third normal form (3NF) but are at least in second normal form (2NF). That is, the relations in the input schema for the mapping may contain transitive functional dependencies (but not partial dependencies). The availability of functional dependencies for the 2NF relations is required for the mapping process.

In addition, any schema mapping procedure involving the relational schema and a conceptual schema requires the knowledge of primary keys and foreign keys or their equivalent (e.g., inclusion dependencies)[8]. Two attributes are considered the "same" (synonymous) if they have the same domain. For example, "SSN" and "Leader-SSN" are considered the same since they both have the same domain (namely, the set of all social security numbers). Such similarity of the attributes can be determined based on the primary keys to which the foreign keys refer. If this information is not available from the data dictionary, then the user has to provide this information. It is also assumed that there are no homonyms (attributes that have the same name but different domains) in the schema.

# 3 The Mapping Procedure

## 3.1 Identify Object Classes

We define a class-relation to be a relation that directly maps to an object class. We first extract object classes from 2NF relations. Consider a 2NF relation $\Re(A)$ where A is the set of attributes of the relation. Let the functional dependencies of the form $P \to Q, Q \to R$ hold where $Q$ is not a subset of any key of $\Re$. $P$ and $Q$ could be complex attributes. Create a class $C_1$ having attributes $A - R$ and class $C_2$ having attributes $Q \cup R$. Create two temporary class-relations $CR_1$ having attributes $A - R$ and $CR_2$ having attributes $Q \cup R$ where $Q$ acts as a foreign key in $CR_1$ and as primary key in $CR_2$. This process is very similar to the normalization process except that the original schema is not being changed here.

For example, consider a 2NF relation[1]:

$$R(\underline{A}, B, C, D, E);\ A \to B, B \to CD$$

This relation $R$ can be split into two *temporary* class-relations

$$R_1(\underline{A}, B^*, E)$$

$$R_2(\underline{B}, C, D)$$

All the 3NF relations qualify to be class-relations and hence we map each of them to an object class. The classes whose relations have the primary key entirely composed of foreign keys represent associations that are themselves classes. Such classes will be termed association-classes. Note that creation of more classes is likely during the subsequent steps.

As explained earlier, three kinds of relationships can be can be expressed in the object model: association, inheritance, and aggregation. The following subsections describe methods to identify each of those.

## 3.2 Identify Associations

Associations between object classes can be identified based on two cases. First, every relation whose primary key is entirely composed of foreign keys is a class that represents an association between all the

[1] In this and other table definitions in this paper, underlined attribute names indicate primary keys and an asterisk on an attribute name indicates foreign keys.

classes corresponding to the class-relations that the foreign keys refer to.

Second, for every class-relation that is not an association class, establish an association relationship between the corresponding class and the class corresponding to each non-key foreign key.

## 3.3 Identify Inheritance

Three forms of relational structures may indicate an inheritance relationship. The following subsections address each of these three cases.

### 3.3.1 Case 1:

In the first case, the relational schema involves one relation for the super class and one relation each for all the subclasses. Every pair of class-relations $(CR_1, CR_2)$ that have the same primary key may be involved in an inheritance relationship. The inheritance relationship $CR_1$ "is-a" $CR_2$ exists if the primary key of $CR_1$ is also a foreign key and that refers to the class-relation $CR_2$.

This is the simplest of the three cases and is illustrated with an example in Figure 1. In that example, the $PROJECT$ relation and the $HARDWARE\_PROJ$ relation have Proj# as the key and $HARDWARE\_PROJ.Proj\#$ refers to $PROJECT.Proj\#$.
Therefore, $HARDWARE\_PROJ$ is a subclass of $PROJECT$. Similarly, $SOFTWARE\_PROJ$ is a subclass of $PROJECT$.

### 3.3.2 Case 2:

Sometimes, inheritance relationship may be embedded in a single class relation. For example, consider the table:

$$PROJECTS(\underline{Proj\#}, ProjName, ProjType, Supplier\#^*, Language, LOC)$$

In the PROJECTS table, the attributes Supplier# is relevant only for hardware projects (i.e., ProjType = 'H') and the attributes Language and LOC are relevant only for software projects (i.e., ProjType = 'S'). The table contains null values for the Supplier# attribute for all software projects and null values for the Language and LOC attributes for all hardware projects. This is an example of instances of two subclasses being stored in the single relation (perhaps to avoid a join operation).

Knowledge discovery from databases is concerned with extracting non-trivial information from existing databases [9]. There are several knowledge discovery algorithms that can identify rules from relational databases [9]. By using one of those algorithms on the PROJECTS relation, for instance, we can identify rules such as

$$\begin{aligned} ProjType =' S' &\Rightarrow Supplier\# = NULL \\ ProjType =' H' &\Rightarrow (Language = NULL \bigwedge \\ &\qquad LOC = NULL) \end{aligned}$$

Such types of rules are typically called "strong rules" since they hold for all the instances of the relation.

In effect, knowledge discovery algorithms can be used to identify the discriminant attribute of the generalization by finding rules whose right hand side is of the form $A = NULL$ where A is some attribute name. Following is the generalized procedure for identifying sub-classes once the characteristic rules have been identified.

Consider a relation $\Re(A)$ where A is the set of attributes of the relation and PK is the primary key. The following steps can be used to determine embeded inheritance relationships.

1. Apply an algorithm to identify strong rules of the form:

   $Rule_i : (D = V_j) \Rightarrow X_i = NULL,$ where
   $X_i$ could be a combination of more than one attribute

   In this rule, each $X_i$ may be considered to be a set of attributes that have null values.

2. The goal in the second step is to partition $\bigcup_i X_i$ and create one subclass each for each set in the partition. Let $P$ be the set of sets of the form $X_i$.

```
$P' = \emptyset$
while $X_i \bigcap X_j \neq \emptyset \ \forall X_i, X_j \ \epsilon \ P, \ i \neq j$ do
        $P' = P' \bigcup (X_i \bigcap X_j) \ \forall X_i, X_j \ \epsilon \ P, \ i \neq j$
        $P = P'$
end while
```

3. Create a temporary class relation and a superclass having attributes

$$A - \bigcup_k X_k, X_k \epsilon P$$

4. For each $X_k \ \epsilon \ P$ do

```
For each $X_k \ \epsilon \ P$ do
        create a temporary class relation and
        a subclass having attributes $PK \bigcap X_k$
end for
```

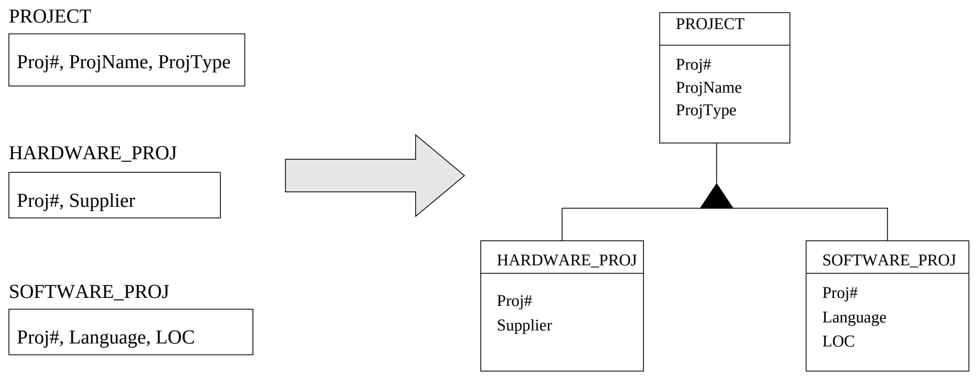

Figure 1: Example for case 1 inheritance

The result at the end of applying the above steps is the creation of one subclass for each unique value $V_i$ of the discriminant attribute[2] $D$.

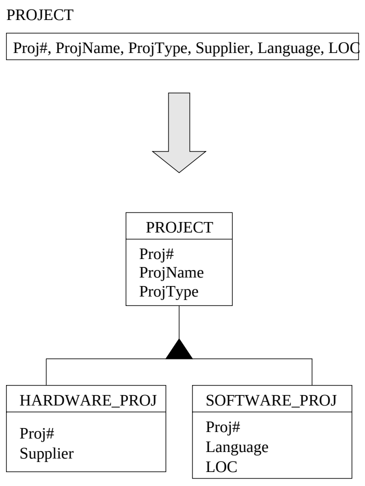

Figure 2: Example for case 2 inheritance

Applying this procedure on the $PROJECTS$ relation results in two temporary class-relations and two generalization relationships as shown in Figure 2. It is the user's responsibility to give meaningful names (e.g., HARDWARE_PROJ, SOFTWARE_PROJ) to the temporary class-relations. One restriction of the above procedure for this case is that it can identify inheritance hierarchies that are only one level deep.

### 3.3.3 Case 3:

In the third case, each subclass is represented by a single relation with no relation corresponding to the superclass. The attributes of the superclass are repeated in the schema of each of the relations corresponding to the subclasses. Superclasses and subclasses for this case can be identified using a three step algorithm as follows.

Consider a relational schema $\Re = R_1, R_2, ..., R_n$ having primary keys $PK(R_i)$ for all $i$. The result of applying the following steps is a set $G$ of ordered pairs $< C_1, C_2 >$ where $C_1$ is a subclass of $C_2$. Some new object classes may also result at the end of this process. The whole process may described in the following steps:

1. For every pair of relations $R_i$ and $R_j$ in $\Re$, that have the same key $PK$, if $PK \subset R_i \cap R_j$, then

   (a) create new temporary class-relation $T(R_i \cap R_j)$.

   (b) Add the ordered pairs $< R_i, T >$ and $< R_j, T >$ to $G$.

   Repeat the process until no new temporary class-relations is formed.

2. Create an object class corresponding to each temporary class-relation.

3. Eliminate transitive generalization relationships from $G$. That is, remove all $< C_1, C_2 >$ if there exists some $T$ in $G$ such that $< C_1, T >$ is in $G$ and $< T, C_2 >$ is in $G$.

To illustrate the three steps with an example, consider three hypothetical relations:

$$R_1(\underline{A}, B, C, D, E)$$
$$R_2(\underline{A}, B, C, F, G)$$
$$R_3(\underline{A}, B, H, I)$$

[2]This procedure may be extended to the cases where a relation has more than one discriminant attribute or a discriminant is composed of more than one attribute.

The result of applying steps 1 and 2 is

$$T_1(\underline{A}, B, C)$$
$$T_2(\underline{A}, B)$$
$$G = \{< R_1, T_1 >, < R_2, T_1 >, < R_1, T_2 >, < R_2, T_2 >, < R_3, T_2 >, < T_1, T_2 >\}$$

Finally, eliminate redundant generalization pairs (step 3) by removing $< R_1, T_2 >$, $< R_2, T_2 >$ from $G$. Thus, the final generalization hierarchy after applying this process is

$$G = \{< R_1, T_1 >, < R_2, T_1 >, < R_3, T_2 >, < T_1, T_2 >\}$$

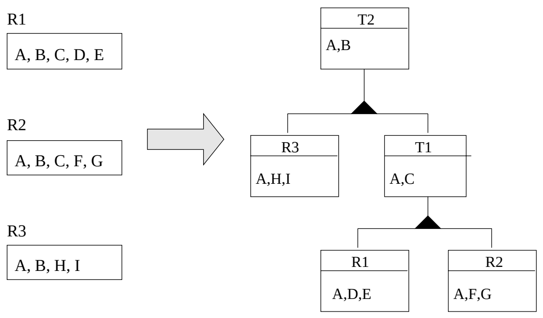

Figure 3: Example for case 3 inheritance

## 3.4 Identify Aggregation

Recall that an aggregation relationship exhibits existence dependency. That is, if A “is-part-of” B, then the existence of A depends on the existence of B. Aggregation relationships may be embedded in at least two forms of relational structures. The following subsections address each of these two cases.

### 3.4.1 Case 1:

Consider a class-relation $CR_1$ whose primary key has more than one attribute and at least one of them is not a foreign key. Let $CR_2$ be the class-relation corresponding to the largest subset of foreign keys in the primary key of $CR_1$. If such a class-relation exists, then $CR_1$ “is-part-of” $CR_2$. Notice that this definition recognizes aggregations of association classes, too.

The following two example relations illustrate this case:

$$EMPLOYEE(\underline{SSN}, Name, Age)$$
$$DEPENDENT(\underline{SSN^*, Dependent\#}, DependentName, DependentAge)$$

Based on the above procedure, the *Dependent* object “is-part-of” an *Employee* object. That is, a *Dependent* object cannot exist without a corresponding *Employee* object.

### 3.4.2 Case 2:

Many RDBMSs these days provide support for BLOB (binary large object) data type to facilitate the storage of multimedia objects such as images, audio data, and so on. Lack of support for arrays of values as part of a single tuple (i.e., multi-valued attributes) is a well-known limitation of the relational model. Many relational databases utilize the BLOB data type for storing multi-valued attributes. The applications take the responsibility of converting multiple values (arrays) to binary streams and vice versa. Since such arrays in scientific applications can contain thousands of elements, using BLOB data types is the most efficient way to store and retrieve the values.

It is impossible for the schema mapping procedure to determine whether a particular BLOB attribute represents a single value such as image or multi-values such as an array. Therefore, whenever the mapping procedure comes across BLOB data types, it has to consult the user. If the attribute corresponds to an array data type, then there is basically an aggregation relationship between the class-relation and the elements of the array. Note that the composition of each element of the array may be arbitrarily complex (e.g., a nested 'struct').

## 4 Related Work

We have thus described the steps to identify the major object-oriented constructs in a relational database. Chiang et al. provide a comprehensive method for extracting an EER schema from a 3NF relational schema[2]. Their method uses inclusion dependencies in the process. They also propose heuristics for automatic extraction of inclusion dependencies thus increasing the potential for automation. The main difference between their work and the procedure we described here is that we try to proceed with the mapping process even if the relational schema is not in 3NF. On the the other hand, they recommend converting the non-3NF schema to a 3NF schema. This is not a viable alternative for most existing relational database when the goal of the schema mapping is to be access the database using an object-oriented view. Fahrner and Vossen describe a method for transforming relational database schemas to object schemas that satisfy ODMG-93[3][3]. They, too, attempt to deal with non-3NF databases by incorporating normalization as one of the intermediate steps in the process. One other primary difference is that their motivation for schema mapping is directed more towards implementing in an ODBMS rather than using an object-oriented view.

There are at least three commercial systems that serve as middle-ware products between an object-oriented programming language and the relational database. The ONTOS Object Integration Server ($OIS^{TM}$) allows the developer to select from various mapping possibilities and carry out the mapping process interactively [5]. $Persistence^{TM}$ is an "object-to-relational mapping system which automates the mapping of object applications to relational databases"[6]. The system also provides the ability to read from existing relational data dictionaries to generate the object schema. More details on the reverse mapping are not publicly available at the time of this writing. A third product called $CGEN^{TM}$ does a straight table-to-class mapping without providing any interpretation of foreign key information[7].

One common observation among these different methods is that the user's intervention is required at some point and that few methods pay particular attention to unusual constructs in the relational database schema.

## 5 Conclusions

In this paper, we presented a procedure for mapping a relational schema to an object-oriented schema. Relational database developers employ many optimization techniques especially to avoid expensive operations like join. Second normal form relations and BLOB data are examples of these optimizations. We presented approaches that extracted object-oriented structures from such relational databases without the need to modify the existing relational database schema. We also demonstrated the use of knowledge discovery techniques for extracting inheritance structures embedded in a single relation. User's involvement is mainly required for specifying functional dependencies for 2NF relations and for specifying the composition of BLOBS in case they represent complex aggregates.

Our current work involves creating a schema mapping repository for an existing relational database and using the repository for dynamic creation of C++ objects from relational data. This task will lead to our overall objective of providing object-oriented views of existing relational databases.

[3]ODMG-93 is the result of an on-going standardization of object-oriented databases [1].

## References

[1] Rick Catell, editor. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan-Kaufmann, 1995.

[2] Roger H.L. Chiang, Terence M. Barron, and Veda C. Storey. Reverse engineering of relational databases: Extraction of an EER model from a relational database. *Data & Knowledge Engineering*, (12):107–142, 1994.

[3] Christian Fahrner and Gottfied Vossen. Transforming relational database schemas into object oriented schemas according to ODMG-93. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, Singapore, 1995. National University of Singapore. to appear.

[4] Christian Fahrner and Gottfried Vossen. A survey of database design transformations based on the entity-relationship model. *Data & Knowledge Engineering*, (15):213–250, 1995.

[5] Ontos Inc. Ontos object integration server. World-Wide-Web Homepage, 1996. http://www.ontos.com/.

[6] Persistence Software Inc. Persistence software: Technical overview. World-Wide-Web Home Page, 1996. http://www.persistence.com.

[7] Subtle Software Inc. Subtleware C++ class generator. World-Wide-Web Home Page, 1996. http://world.std.com/ subtle/.

[8] Paul Johannesson and Katalin Kalman. A method for translating relational schemas into conceptual schemas. In Frederick H. Lochovsky, editor, *Proceedings of the Eighth International Conference on Entity Relationship Approach*, pages 271–285, Toronto, Canada, October 1989. North-Holland.

[9] Gregory Piatetsky-Shapiro and William Frawley, editors. *Knowledge Discovery in Databases.* AAAI Press/The MIT Press, California, 1991.

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modelling and Design.* Prentice-Hall, Inc., Eaglewood Cliffs, New Jersey, 1991.