ISSN: 0128-7680
Pertanika J. Sci. & Technol. 13(1): 1 - 21 (2005)

Developing Translation Rules for Converting Relational to Object Oriented Database Conceptual Schema

Hamidah Ibrahim, Soon Lay Ki, Ali Mamat & Zaiton Muda
Department of Computer Science
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
43400 UPM, Serdang, Selangor, Malaysia
E-mail: hamidah@fsktm.upm.edu.my

Received: 27 November 2001

ABSTRAK

Pangkalan data multi adalah satu persekutuan sistem pangkalan data teragih, heterogen dan berotonomi yang telah ujud. Kebiasaannya, proses integrasi adalah perlu dalam usaha membentuk satu sistem pangkalan data teragih yang heterogen. Proses ini secara amnya mengandungi dua fasa utama, iaitu fasa penterjemahan skema konseptual diikuti dengan fasa integrasi. Makalah ini mempersembahkan satu pendekatan penterjemahan untuk menukar skema pangkalan data hubungan kepada skema pangkalan data berorientasi objek. Pendekatan penterjemahan tersebut mengandungi satu set peraturan penterjemahan, yang berdasarkan kepada kebergantungan terangkum, atribut kunci dan jenis atribut. Satu prototaip alat penterjemahan skema pangkalan data, dipanggil RETOO dibangunkan berdasarkan kepada pendekatan penterjemahan yang dicadangkan. RETOO menerima skema pangkalan data hubungan sebagai data input dan menjana skema pangkalan data berorientasi objek sebagai output. Pendekatan penterjemahan bukan sahaja dapat memelihara semantik skema pangkalan data hubungan tersebut, tetapi juga meningkatkan semantik skema berorientasi objek yang diterjemahkan melalui konsep permodelan data berorientasi objek.

ABSTRACT

A multidatabase is a confederation of pre-existing distributed, heterogeneous, and autonomous database system. Obviously, the integration process is essential in the effort of forming a distributed, heterogeneous database system. This process generally consists of two main phases, which are conceptual schema translation phase followed by the integration phase. This paper presents a translation approach to convert relational database schema to object-oriented database schema. The translation approach consists of a set of translation rules, which is based on inclusion dependencies, key attributes and types of attributes. A database schema translation tool prototype, called RETOO (RElational-To-Object-Oriented) is then developed based on the proposed translation approach. RETOO receives a relational database schema as input data and generates an object-oriented database schema as the output. The translation approach is not only able to maintain the semantics of the relational database schema, but also enhance the semantics of the translated object-oriented schema via object-oriented data modeling concepts.

Keywords: Relational schema, object-oriented schema

INTRODUCTION

In today's information age, databases and database technology are having a major impact on the growing use of computers. The government, education, medicine, engineering, business and other areas have computerized all or part of their daily functions. Undoubtedly, these computerization processes often include database systems to model and store the information of the real-world entities involved in these functions. The computing environment in most of these contemporary organizations contains distributed, heterogeneous, and autonomous hardware and software systems. Therefore, there is an increasing need to support the co-operations of the services provided by these different software and hardware.

The existence of multiple, heterogeneous and autonomous databases within an organization means the globally important information exists in separate local database management systems (DBMSs), thus making the existing data inaccessible to remote users. One solution is to integrate these databases to form a single cohesive definition of a multi-database. Most of the integration is made possible with the support of database translation, which is the task of translation from one database conceptual schema into another.

Most works on schema translation deal with conversion from the entity-relationship (ER) model to the relational model or some extension of it (Castellanos *et al.* 1994; Castellanos and Saltor 1991). There are many works on translation from ER model into relational model or vice versa (Huang *et al.* 1997; Lukovic and Mogin 1996; Seol 1997). Besides, works on general frameworks for schema translation were also carried out (McBrien and Poulovassilis 1998).

Nevertheless, only a few works have been done on translating relational schema into object-oriented (OO) schema (Castellanos *et al.* 1994; Castellanos and Saltor 1991; Fong 1997; Soon *et al.* 2001; Stanisic 1999). Stanisic (1999) focused his work not only on schema translation, but query translation as well. While Castellanos *et al.* (1994) proposed a methodology to translate the relational model into Barcelona Object-Oriented Model, namely BLOOM model. However, these works have their limitations respectively, especially in terms of translated OO model representation. The limitations in the BLOOM OO model include (i) the syntax of resulted BLOOM OO model is not easy to understand, such as the keywords *s_aggreg_of* and *compl_generaliz_of;* (ii) the model tends to create extra classes, which are sometimes not necessary; and (iii) the data types of attributes in BLOOM OO model are not specified.

In our work, a set of translation rules is proposed to translate relational database conceptual schema into OO database conceptual schema. This set of translation rules is applied in a database schema translation tool prototype, called RETOO (RElational-To-Object-Oriented), with the assumption that OO conceptual schema is used as the canonical conceptual schema (CCS). This canonical conceptual schema will then be integrated into the global conceptual schema (GCS) of the distributed, heterogeneous database system. *Fig. 1* briefly illustrates the system. InS_1 ... InS_n shown in *Fig. 1* are intermediate schemas or known as canonical conceptual schemas.

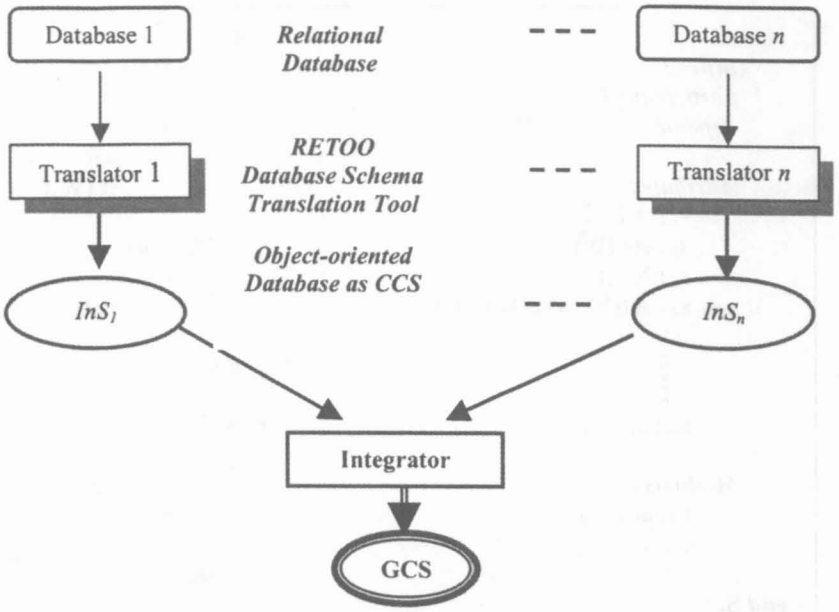

Fig. 1: Relational-to-object-oriented database schema translation tool

PRELIMINARIES

In our work, the relational conceptual schema and object-oriented conceptual schema are represented in the format as shown in *Fig. 2* and *Fig.* 3, respectively.

S, T and *U* are the names of the relations while s_1 to s_n, t_1 to t_3 and u_1 to u_3 are the attributes of relations *S, T* and *U* respectively. Ds_1 to Ds_n, Dt_1 to Dt_3 and Du_1 to Du_3 are the data types (domain) associated with each attribute while the underlined attribute is the primary key. Note that relations *S, T* and *U* might have a primary key, k, which is defined over more than one attribute of these relations.

In *Fig. 3, S, T, U, V, W, X* and *Y* are the names of the classes. The interactions among classes are shown by keywords *inherit, inherited_by, assemble, participate_in, depend, has_dependent, set( )* and *inverse is.* Every class has its attributes and methods or operations.

S ($\underline{s_1}$: Ds_1,	T ($\underline{t_1}$: Dt_1,	U ($\underline{u_1}$: Du_1,
s_2: Ds_2,	t_2: Dt_2,	u_2: Du_2,
...	t_3: Dt_3)	u_3: Du_3)
s_n: Ds_n)		

Fig. 2: Examples of the format of relational conceptual schema

```
class S
    inherit          T
    participate_in   U
    depend           V

    Attributes
        s1 : string;
        s2 : set (W);
        s3 : X.x1;
        s4 : set(Y) inverse is Y.y1;
        ⋮
        sm : string;

    Methods
        Create(...);
        ...
end S.
```

```
class U
    assemble    S

    Attributes
        u1 : char;
        u2 : set (W);

    Methods
        Create(...);
        Destroy(...);

end U.

class W
    Attributes
        w1 : char;
        w2 : char;

    Methods
        Create(...);
        ...
end W.
```

Fig. 3: Object-oriented conceptual schema

The format of the OO schema is modified from the standard object-oriented database schema to a more easy-to-understand format. As can be seen in *Fig. 3*, the first part of the schema is the declaration of the beginning of a class, which is *class S*. This is followed by the declaration part for the inheritance (*inherit*) and aggregation (*assemble* and *depend*) of the class. The word *depend* shows the way of presenting weak entity type in OO data modelling. Even though relationship between the weak entity and its parent entity is considered as a kind of aggregation, the keyword *depend* is used for the purpose of better understanding. All these three keywords have their own inverse versions, which are *inherited_by*, *participate_in* and *has_dependent*.

In this example, we have other six classes, namely *T*, *U*, *V*, *W*, *X* and *Y*. Class *S* inherits from class *T*, aggregated by class *U* and is the dependent of class *V*. In other words, class S *is-an* instance of class *T* and is part of the aggregation of class *U*. Class *V* is the parent entity of weak entity *S*. In contrast, these inverse classes will have the inverse versions of the keywords, for example in class *U*, it has *assemble S*.

Another keyword in the schema, i.e. *set* is used when attribute's type is a set of attributes. In *Fig. 3*, s_2 consists of a set of attributes from class *W*. Notice that there is an attribute, which is s_3 with the data type of $X.x_1$, this means the attribute is 'mapped' from attribute x_1 of class *X*. Besides these, the keyword

```
class Account
    Attributes
        name, acc_number: String;
    Methods
        Create();
        Deposit(amount: Money);
        Withdraw(amount: Money)
end Account.
```

Fig. 4: Class with extra methods

inverse is is used to specify the interaction between classes. Attribute s_4 in class *S* corresponds with the attribute y_1 of class *Y*. Hence, the integrity constraints are clearly shown in this schema.

Followed subsequently is the declaration of the operations in the class with the heading *Methods*. One of the most common method for classes is the creator, which will create instances of that class. However, some classes might have other methods representing their behavior, as shown in *Fig. 4*. Finally, the closing of class is done by using the keyword *end*.

RELATIONAL TO OBJECT-ORIENTED DATABASE SCHEMA TRANSLATION APPROACH

The translation rules proposed by us are based on two characteristics of database schema, they are: (i) inclusion dependency and (ii) key attributes and types of attributes. Two phases are involved in translating relational to object-oriented database conceptual schema, which are: (i) identifying classes and (ii) identifying the operations. Both phases especially the second phase operate semi-automatically, since the information regarding the behavior of each class is not provided in the relational data model. To perform the translation process we have identified ten translation rules which are based on the mapping and normalization process in relational data modelling.

Identifying Classes

To identify objects or classes, there are four steps as presented below.

Step 1: Translating Relation into Class

The first rule is:

Rule 1: *If* R is a relation with attributes A_1, A_2, ...A_n,
then create a class R with attributes A_1, A_2, ...A_n.

In this step all relations are formed into classes. Each class will have attributes and types of attributes. Below is an example:

```
Surgeon(  SName     :  String,
          Street    :  String,
          City      :  String,
          Country   :  String,
          Phone-No  :  String     )
```

After translation from Step 1, we have class *Surgeon* as shown below:

```
class Surgeon
      Attributes
            SName     :  String;
            Street    :  String;
            City      :  String;
            Country   :  String;
            Phone-No  :  String;
end Surgeon.
```

Step 2: Identifying Composite Attributes

The general guideline to decide what an object is and what an attribute of an object is lies in the theory of data abstraction. This theory states that something should only be represented by a class if it represents a set of similar objects or concepts with meaningful properties and operations, which are required to be maintained by the system (Hughes 1991).

Composite attributes are attributes that can be divided into smaller subparts, which represent more basic attributes with independent meanings of their own (Elmasri 2003). Composite attributes represent a set of objects with meaningful simple attributes. There are three cases to be considered, namely: relation that consists of *m* composite attributes with (i) no overlapping attribute between the composite attributes; (ii) at least two of the composite attributes have a common attribute and (iii) at least one of the composite attribute consists of attributes which are common to another composite attribute. Each case is discussed below.

Case 1: No overlapping attribute between the composite attributes.

The second translation rule is stated as:

Rule 2: If relation R consists of m composite attributes CA_i, where $1 \leq i \leq m$ and $CA_i = \{A_{i1}, A_{i2}, \ldots, A_{in}\}$ with no overlapping attributes between the CA_i, i.e. $\cap^{m}_{i=1} CA_i = \{\ \}$,

then
- the attributes forming the composite attribute CA_i are taken out from class R, and are formed as a newly defined class, say T_i;
- in class R, attributes $A_{i1}, A_{i2}, \ldots, A_{in}$ forming the composite attribute CA_i are replaced by statement RCA_i: *set*(T_i), where RCA_i is an attribute in class R referring to class T_i.

Referring to the example in Step 1, there is a composite attribute *Address*, which consists of three attributes, namely: *Street*, *City*, and *Country*. As a result, these three attributes are taken out from the class *Surgeon* and formed as another class *Address*, as shown below:

class Address
    Attributes
        Street, City, Country : String;
end Address.
class Surgeon
    Attributes
        SName : String;
        SAddress : set(Address);
        Phone-No : String;
end Surgeon.

If there exists the same non-key composite attributes in another relation, redundancies can be solved by referring to the same new class formed.

To illustrate cases 2 and 3, let say we have a relation with attributes as follows:

Surgeon(ID No : String,
FName : String,
MInit : String,
LName : String,
Phone-No : String

Case 2: At least two of the composite attributes have a common attribute.

The third translation rule is stated as:

Rule 3: *If* relation R consists of a composite attribute CA_i with attributes $\{A_{i1}, A_{i2}, \ldots, A_{in}\}$ and another composite attribute CA_j with attributes $\{A_{j1}, A_{j2}, \ldots, A_{jm}\}$, and there exists at least an attribute in CA_j, say A_{jk}, which exists in both CA_i and CA_j,[1]

then - the attributes forming CA_i are taken out from class R and formed as a newly defined class, say T_i;
- the attributes forming CA_j are also taken out from class R and formed as another newly defined class, say T_j;
- in class T_j, attribute A_{jk} is defined as A_{jk}: $T_i.A_{jk}$;
- in class R, attributes A_{i1}, Ai_2, ..., A_{in} are replaced by statement RCA_i: $set(T_i)$, representing composite attribute CA_i;
- similarly, statement RCA_j: $set(T_j)$ is used to replace attributes A_{j1}, A_{j2}, ..., A_{jm}, representing composite attribute CA_j.

---

[1] And if there is an attribute in CA_j, say A_{jl} which is a simple attribute by itself, then in class T_j, attribute A_{jl} is defined as A_{jl}:$R.A_{jl}$; and attribute A_{jl} will remain in class R.

Let's assume that there are two composite attributes in this relation, which are:

- Name: FName, MInit, LName
- Staff_No: FName, Phone-No

In this case, we have an attribute *FName* that exists in both composite attributes *Name* and *Staff_No*. The attributes that form these composite attributes will be taken out from the original relation and formed as classes, same as the simpler case discussed earlier. Therefore, after the translation process, we will get the following three classes:

```
class Name
        Attributes
                FName, MInit, LName   :   String;
end Name.
class Staff_No
        Attributes
                FName      :  Name.FName;
                Phone-No   :  String;
end Staff_No.
class Surgeon
        Attributes
                ID_No      :  String;
                SName      :  set(Name);
                Staff_No   :  set(Staff_No);
end Surgeon.
```

Case 3: At least one of the composite attributes consists of attributes which are common to another composite attribute.

The fourth translation rule is stated as:

Rule 4: *If* relation R has a composite attribute $CA_i = \{A_{i1}, A_{i2}, \ldots, A_{in}\}$ and another composite attribute $CA_j = \{A_{j1}, A_{j2}, \ldots, A_{jm}\}$ where CA_j C CA_i (CA_j is a subset of CA_i),

then - the attributes $A_{i1}, A_{i2}, \ldots, A_{in}$ forming CA_i are taken out from class R and formed as a newly defined class, say T_i;
- the attributes $A_{j1}, A_{j2}, \ldots, A_{jm}$ forming CA_j are also taken out and formed as another newly defined class, say T_j;
- in class T_j, attribute A_{jk} where $1 \leq k \leq m$ is defined as A_{jk}: $T_i.A_{jk}$;
- in class R, attributes $A_{i1}, A_{i2}, \ldots, A_{in}$ are replaced by statement RCA_i: $set(T_i)$ representing composite attribute CA_i;
- in class R, statement RCA_j: $set(T_j)$ is used to represent composite attribute CA_j.

In this case, let's assume that we have another two sets of composite attributes in the same relation *Surgeon*.

- Full_Name: FName, MInit, LName
- Name: FName, LName

Applying rule 4 will derive the following three classes:

class Full_Name
Attributes
FName, MInit, LName : String;
end Full_Name.
class Name
Attributes
FName : Full_Name.FName;
LName : Full_Name.LName;
end Name.
class Surgeon
Attributes
ID_No : String;
SFull_Name : set(Full_Name);
SName : set(Name);
Phone-No : String;
end Surgeon.

Step 3: Identifying Relations with Foreign Keys only

In this step, we identify relations, which have only foreign keys. According to the mapping process in relational data modelling, a relation will have only foreign key attributes when the relation is formed as a result of an interaction between or among other relations in M:N relationship. These foreign keys, which originated from the key attributes of the relations involved in that interaction will form the primary key of this newly formed relation.

Thus, when translating these relations, we will regard them as an object resulting from the interaction between or among the classes that the foreign key attributes refer to, as reflected in Rule 5:

Rule 5: *If* relation R consists of n attributes A_1, A_2, ..., A_n where each A_i is the foreign key that refers to relations U_i, where $1 \leq i \leq n$,

then - class R is treated as interactions of all the classes {U_1, U_2, ..., U_n};
- in class U_i, statements {*R: set(U_1) inverse is U_1.R, R: set(U_2) inverse is U_2.R, ..., R: set(U_n) inverse is U_n.R* } – {*R: set(U_i) inverse is U_i.R*} are stated;
- class R is abolished.

The example below illustrates this translation step.

Paper(P#, Title, Issue# : String,
Institute_Name, Vol# : String)
Author(AName, Nationality : String,
Date_of_Birth : Date)
Writes(P#, AName : String)
ID: Writes.P# ⊆ Paper.P#
ID: Writes.AName ⊆ Author.AName

The *Writes* relation consists of two foreign key attributes where *P#* refers to the *P#* in relation *Paper* and *AName* refers to the *AName* in relation *Author.* Therefore, the relation *Writes* is representing the interaction between relations *Paper* and *Author.* Class *Writes*, which was formed in translation step 1 will be abolished.

class Paper
    Attributes
        P#, Title, Issue# : String;
        Institute_Name, Vol# : String;
        Written_by : set(Author)
            inverse is Author.write;
end Paper.
class Author
    Attributes
        AName, Nationality : String;
        Date_of_Birth : Date;
        Write : set(Paper)
            inverse is Paper.written_by;
end Author.

Step 4: Identifying Foreign Keys and Candidate Keys Being Referenced

In this step, we shall focus on the referential integrity, which includes identifying foreign keys and candidate keys being referenced. There are two possibilities identified regarding the referential integrity, as shown in Table 1.

TABLE 1
Foreign key

Foreign Key	Candidate Key being Referenced
Key Attribute	Key Attribute
Non-key Attribute	Key Attribute

The first case (Case 1) occurs when both the foreign key and the candidate key being referenced are key attributes in both relations. The second case (Case 2) occurs when the foreign key is a non-key attribute whereas the attribute being referenced is a primary key attribute in the original relation.

Case 1: Both the foreign key and the candidate key being referenced are key attributes in both relations.

In this case, we can further divide it into four categories, as shown in Table 2.

Based on the definition of key constraint in relational modeling (Elmasri 2003), we know that when the key attribute of a relation R_1 is a foreign key, it implies that this relation refers to the whole relation R_2 that contains the key being referenced. Therefore, R_1 is an instance of R_2 whereby besides the

TABLE 2
Categories of case 1

Foreign Key	Candidate Key being Referenced
Simple primary key	Simple primary key
Composite primary key	Composite primary key
Composite primary key	Simple primary key
Part-of composite primary key	Simple/Composite primary key

attributes in R_2, R_1 has its own attributes. In OO modelling, this situation is similar to one of the OO concepts, which is inheritance. A subclass is said to be inherited from a superclass if the subclass "is-an" instance of the superclass.

For category one, if both the foreign key and the candidate key being referenced are simple primary key attributes of the relations, our translation rule will consider the foreign key's relation inherits from another being referenced relation. This applies correctly even if both of the foreign key and the key being referenced are composite primary keys, which is the second category, as stated in Rule 6:

Rule 6: *If* both the foreign key in relation R and the candidate key being referenced in relation V are simple primary key attributes or composite primary keys,

then - class R is treated as an inheritance of class V;
- statement *inherit V* is included in class R;
- statement *inherited_by R* is included in class V.

For example, the *SName* attribute in *Consultant* is the foreign key, which refers to the primary key of *Surgeon*. In this case, we can say that the *Consultant* "is-a" *Surgeon*.

Surgeon(SName, Street, City : String,
Country, Phone_No : String)
Consultant(SName, Speciality : String)
ID: Consultant.SName ⊆ Surgeon. SName

After translation, we shall get the following OO schema:

class Surgeon
inherited_by Consultant
Attributes
SName : String;
SAddress : set(Address);
Phone_No : String;
end Surgeon.
class Consultant
inherit Surgeon
Attributes
Speciality : String;
end Consultant.

Category three indicates that there might exist a relation with more than one foreign key and all the foreign keys formed the primary key of the relation. Besides that, this relation also has its own attribute(s). If the subclass "is-an" instance of both the superclasses, we will treat the relationships among the relations as multiple inheritance. Based on this third category, we have the following rule:

Rule 7: *If* relation R has a set of foreign keys $\{fk_1, fk_2, \ldots, fk_n\}$ where $n > 1$ and fk_i where $1 \leq i \leq n$ formed the primary key of R, and after being translated into class R, class R is an instance of the classes $C_1, C_2, \ldots, C_m$ where its foreign keys are referred to, i.e. $R.fk_i \subseteq C_i.pk$[2], where pk is the primary key of C_i,

then - class R is treated as an inheritance of classes $C_1, C_2, \ldots, C_m$;
- in class R, statements *inherit* C_i, where $1 \leq i \leq m$ are included;
- statements *inherited_by R* are included in classes $C_1, C_2, \ldots, C_m$.

For example, in a factory, it produces a *Toy*, which is a *CommercialProduct* and at the same time, it is also a *Gift* for customer:

CommercialProduct(CommercialID : String,
Packaging : String,
Price : Integer)
Gift(GiftID, Category : String,
Coupon : Integer)
Toy(CommercialID : String,
GiftID : String,
Age : Integer)
ID: Toy.CommercialID ⊆ CommercialProduct.CommercialID
ID: Toy.GiftID ⊆ Gift. GiftID

The *Toy* "is-a" *CommercialProduct* and also "is-a" *Gift* to the factory. As a result, the three classes will be formed as below:

class CommercialProduct
inherited_by Toy
Attributes
CommercialID : String;
Price : Integer;
Packaging : String;
end CommercialProduct.
class Gift
inherited_by Toy
Attributes
GiftID : String;
Category : String;
Coupon : Integer;
end Gift.

---

[2] The symbol ⊆ shows the inclusion dependency.

class Toy
inherit CommercialProduct
inherit Gift
Attributes
Age : Integer;
end Toy.

However, not all relations that have foreign keys as primary key will be considered as having multiple inheritance as presented in the following rule:

Rule 8: *If* relation R has a set of foreign keys $\{fk_1, fk_2, \ldots, fk_n\}$ where $n > 1$ and fk_i where $1 \leq i \leq n$ formed the primary key of R, and after being translated into class R, class R is an aggregation of classes $C_1, C_2, \ldots, C_m$ where its foreign keys are referred to, i.e. $R.fk_i \subseteq C_i.pk$, where pk is the primary key of C_i,
then - class R is treated as an aggregation of classes $C_1, C_2, \ldots, C_m$;
- statements *assemble* C_i where $1 \leq i \leq m$ are included in class R;
- statement *participate_in R* are included in classes $C_1, C_2, \ldots, C_m$.

Refer to the example below:

Programmer(SSN, Salary, Sex : String,
BDate : Date)
Project(P#, PName : String,
StartDate, DueDate : Date)
Works_On(SSN, P# : String,
Hours : Integer)
ID: Works_On.SSN ⊆ Programmer.SSN
ID: Works_On.P# ⊆ Project. P#

In this case, *Works_On* is neither "is-a" *Programmer* nor "is-a" *Project*. Rather, *Works_On* would be more suitable to be identified as an aggregation or assembler of the two classes. If we refer back to the mapping process in relational modeling, *Works_On* resulted from an interaction of M:N relationship of both *Programmer* and *Project*, in which the attribute *Hours* is an attribute obtained from the relationship between *Programmer* and *Project*.

In terms of aggregation, the important point is that, user of *Works_On* does not need to be concerned about the representation details of *Programmer* and *Project*. All the properties of the *Programmer* and *Project* associated with a particular *Works_On* are encapsulated by the class and may be accessed without explicit joins (Hughes 1991).

Therefore, the translation result would be:

class Programmer
participate_in Works_On
Attributes
SSN, Salary, Sex : String;
BDate : Date;
end Programmer.

class Project
    participate_in Works_On
    Attributes
        P#, PName : String;
        StartDate, DueDate : Date;
    end Project.
class Works_On
    assemble Programmer
    assemble Project
    Attributes
        Hours : Integer;
end Works_On.

Lastly, for the fourth category of this case, we identify another situation whereby the foreign key is a part of primary key. The candidate key(s) being referenced might be simple or composite primary key(s). According to the mapping and normalization process in relational data modelling, this situation happens when the relation that contains the foreign key(s) is a weak entity. The key attribute of the parent entity is included as a foreign key in the weak entity and will be part of the key attribute in the weak entity.

Thus, Rule 9 states that:

Rule 9: *If* part of the primary key of relation R is a foreign key attribute, which refers to a relation Q,
then - class R is treated as a weak entity, which depends on class Q;
- statement *depend Q* is included in class R;
- statement *has_dependent R* is included in class Q.

An example is shown below, the class *Children* is a weak entity that depends on its parent entity *Employee.*

Employee(SSN#, Name, Sex : String)
Children(SSN#, Child_Name, Sex : String,
Age : Integer)
ID: Children.SSN# ⊆ Employee. SSN#

As a result, we will get the following two classes:

class Employee
    has_dependent Children
    Attributes
        SSN#, Name, Sex : String;
end Employee.
class Children
    depend Employee
    Attributes
        Child_Name, Sex : String;
        Age : Integer;
end Children.

Case 2: The foreign key is a non-key attribute whereas the attribute being referenced is a primary key attribute in the original relation.

In the third and fourth step of the mapping process in relational modelling, for each regular binary 1:1 and 1:N relationship type *R*, identify the relation *S* that represents the participating entity type at the full participation or N-side of the relationship type. Include as foreign key in *S* the primary key of the relation *T* that represents the other entity type participating in *R* (Elmasri 2003).

Thus, the existence of the non-key attribute in relation *S* that refers to the key attribute of relation *T* means that the foreign key in *S* is merely referring to relation *T* and not an instance of relation *T* or even assembling relation *T*. Thus, the existence of this foreign key as non-key attribute will be treated as an interaction between *S* and *T*.

We shall conclude our translation approach with Rule 10:

Rule 10: If relation R has a foreign key fk which is not a key attribute, that refers to a relation P,

then - attribute fk shows the interaction between class R and class P;
- in class R, statement *fk: set(P) inverse is P.R* replaces attribute fk;
- in class P, statement *R: set(R) inverse is R.fk* is included.

Below is an example demonstrating our approach:

Employee(SSN, Sex : String,
Salary, DeptNo : String,
BDate : Date)
Department(DeptNo, DName, Location : String)
ID: Employee.DeptNo ⊆ Department.DeptNo

After being translated in this step:

class Employee
Attributes
SSN, Sex, Salary : String;
BDate : Date;
Work_in : set(Department)
inverse is Department.
Worked_by;
end Employee.
class Department
Attributes
DeptNo, Dname : String;
Location : String;
Worked_by : set(Employee)
inverse is E
mployee.Work_in;
end Department.

Identifying the Operations

Operations that are applicable to a data abstraction are classified into three categories: (i) constructor/destructor functions; (ii) accessor/query functions and (iii) transformer/update functions. Since the information for the declaration of operations for each object or relation is not provided in the relational data model, user's information is very important in this phase. Initially, our approach will suggest two operations for each class, which are the constructor and destructor operations. Below is an example that shows these two basic operations applied to a class:

class Hotel
    Attributes
        name, owner : String;
        location : set(Address);
        manager : String;
        ...
    Methods
        create(...);
        destroy(...);
end Hotel.

RESULTS AND DISCUSSION

In this section, we will compare the translation approaches proposed by Castellanos *et al.* (1994), Stanisic (1999) and Fong (1997) with our translation approach using two sets of relational database conceptual schema, as shown in *Fig. 5.*

Castellanos *et al.* (1994) worked on translation from relational to object-oriented model known as BLOOM OO model. Their approach creates extra

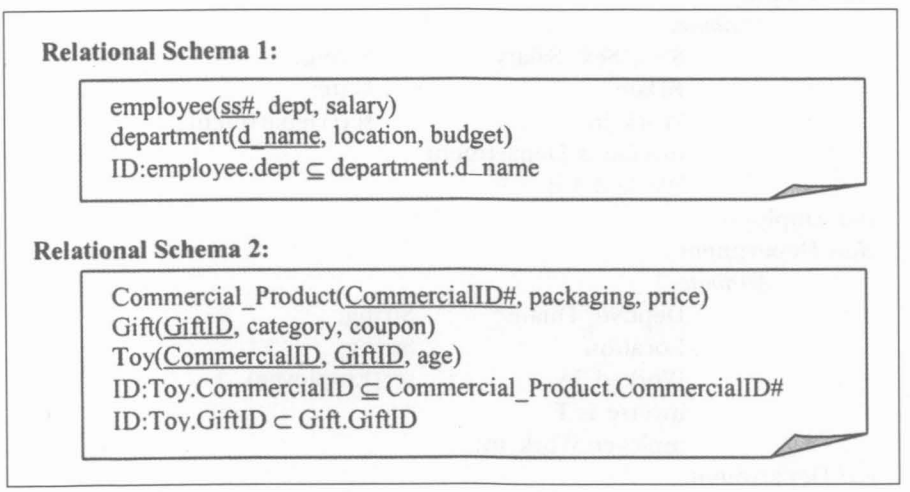

Fig. 5: Relational schemas used for comparisons

classes, which are sometimes not necessary. The translation of the first relational schema used for the comparison demonstrates this weakness.

The translated into BLOOM OO Model:

```
class employee                class privileged              class department
    subclass privileged           superclass employee           s_agg_of manager
    id ss#                    exception_on dept                 id d_name
    atrs  dept                end_class                         atrs  budget
          salary                                                end_class
end_class
```

In this example, one extra class *privileged* has been created. According to Castellanos *et al.* this class is created because *employee.dept* is not null-constrained. Therefore, it can exist as null value. For those employees whose *dept* attribute is null, they are considered as "privileged-employees".

According to the definition of key constraint in relational database design, foreign key either exists as a value of the candidate key it refers to or is null. Therefore, the forming of class *privileged* is not necessary since the existence of null value for *dept* is perfectly fine. In our approach, the existence of *dept* in class *employee* will be indicated as *work_in:department.worked_by*, showing the interaction between these two classes. The translated OO conceptual schema from Relational Schema 1 using RETOO is shown in *Fig. 6*, while the comparison between Castellanos *et al.*'s and our approach on Relational Schema 2 is shown in *Fig. 7*.

We have also studied the translation approach proposed by Stanisic, which translates relational to object-oriented model. However, his translated OO schema is not semantically rich enough as he only considered inheritance and aggregation among the classes. Besides, the relationships among the classes are

```
class department
    Attributes
        d_name      :   string;
        location    :   string;
        budget      :   string;
        worked_by   :   set(employee) inverse is
                        employee.work_in;
end department.
class employee
    Attributes
        ss#         :   string;
        work_in     :   set(department) inverse is
                        department.worked_by;
        salary      :   integer;
```

Fig. 6: Translated OO schema using RETOO approach

By Castellanos

```
class Commercial_Product
    partic_in Toy
    id CommercialID#
    atrs packaging, price
end_class
class Gift
    partic_in Toy
    id GiftID
    atrs category, coupon
end_class
class Toy
    cart_aggr_of
        Commercial_Product
        Gift
    atrs
        age
end_class
```

By RETOO

```
class Commercial_Product
    inherited_by Toy
    Attributes
        CommercialID#   : string;
        packaging       : string;
        price           : string;
end Commercial_Product.
class Gift
    inherited_by Toy
    Attributes
        GiftID          : string;
        category        : string;
        coupon          : string;
end Gift.
class Toy
    inherit     Commercial_Product
    inherit     Gift
    Attributes
        age             : integer;
end Toy.
```

Fig. 7: Comparison of translation result on relational schema 2

not shown clearly in the translated OO schema. His translated OO schemas are shown in *Figs. 8* and *9*.

As shown in *Fig. 9*, class *Gift* and class *Commercial_Product* do not state their relationship with class *Toy*. However, with RETOO, the interactions among classes are specified clearly (refer to *Fig. 7*).

Fong (1997) also proposed an approach to translate relational to object-oriented model. Similarly, the translated OO schema is not semantically rich enough. For instance, his translation approach did not support multiple inheritance among classes. *Fig. 10* illustrates Fong's approach in translating Relational Schema 1.

```
class department
        d_name      :   string;
        location    :   string;
        budget      :   string;
end;
class employee
        ss#         :   string;
        dept        :   ref department;
        salary      :   string;
end;
```

Fig. 8: Translated OO schema using Stanisic's approach on relational schema 1

```
class commercial_Product
    CommercialID#        : string;
    packaging            : string;
    price                : string;
end;
class gift
    GiftID               : string;
    category             :string;
    coupon               :string;
end;
class toy
    com_product          : ref commercial_product;
    gift                 : ref gift;
    age                  : number;
end;
```

Fig. 9: Translated OO schema using Stanisic's approach on relational schema 2

```
class department
    attr d_name      : string
    attr location    : string
    attr budget      : string
    association attr hire ref
                     set(Employee)
end
class employee
    attr ss#         : string
    attr salary      : string
    association attr hired_by ref
                     department
end
```

Fig. 10: Translated OO schema using Fong's approach on relational schema 1

Although there is not much difference shown in translating the first relational schema, according to his approach, the Relational Schema 2 will be translated into the OO schema, as shown below:

```
Class Commercial_Product
        attr CommercialID:string
        attr packaging:string
        attr price:integer
end
```

```
Class Gift
        attr GiftID:string
        attr category:String
        attr coupon:integer
end
Class Toy
        attr age:integer
        association attr CommercialID ref Commercial_Product
        association attr GiftID ref Gift
end
ID:CommercialID ⊆ Commercial_Product.OID
ID:GiftID ⊆ Gift.OID
```

From the above example, we can see clearly that class *Toy* is an instance of class *Commercial_Product* and also an instance of class *Gift*. Therefore, the relationship among these three classes would be more precisely labeled as multiple inheritance. If translated by RETOO, *Toy* will be considered as inheritance of both *Commercial_Product* and *Gift*, as clearly shown in *Fig. 7.*

SUMMARY

We have proposed a methodology to translate relational database conceptual schema into object-oriented database conceptual schema. The translation approach is developed based on the understanding of mapping and normalization processes in relational database modelling. Undoubtedly, the relational semantics are maintained perfectly when the relational model is translated into an object-oriented model. The determiners used in developing the translation rules are inclusion dependencies, key attributes and types of attributes. There are four main steps in the translation approach, which operate based on the ten translation rules.

Besides maintaining the relational semantics, the semantics of our translated object-oriented conceptual schema is also enhanced with richer object-oriented concepts such as aggregation and inheritance. Interaction between or among classes is shown clearly. We also reveal the behavior of every class by adding the methods in the OO conceptual schema. The translation rules differ from previous works in terms of simplified translation approach yet producing a complete and a better-understood object-oriented database conceptual schema.

REFERENCES

CASTELLANOS, M., F. SALTOR and M. GARCÍA-SOLACO. 1994. Semantically enriching relational databases into an object oriented semantic model.

CASTELLANOS, M. and F. SALTOR. 1991. Semantic enrichment of database schemas: an object-oriented approach. *Publication of IEEE*: 71-78.

ELMASRI, NAVATHE. 2003. *Fundamentals of Database Systems.* 4th edition. The Benjamin/ Cummings Publishing Company, Inc.

FONG, J. 1997. Converting relational to object-oriented databases. Publication of SIGMOID Record, 26, No.1.

HUANG, S.M., H.H. CHEN, C.H. LI and J. FONG. 1997. A data dictionary system approach for database schema translation. *Publication of IEEE*: 3966-3971.

HUGHES, J.G. 1991. *Object-oriented Databases.* 1st edition. Prentice Hall.

LUKOVIC, I. and P. MOGIN. 1996. An approach to relational database schema integration. *Publication of IEEE*: 3210-3215.

MCBRIEN, P. and A. POULOVASSILIS. 1998. Automatic migration and wrapping of database applications – A schema transformation approach. Department of Computer Science Technical Report, King's College London.

SEOL, Y.H. 1997. NAMCIC virtual repository schema translation. In *National Academic Medical Center Information Consortium. http://cat.cpmc.columbia.edu/namcic/trans.html.*

SOON, L.K., H. IBRAHIM, A. MAMAT and C.S. PUA. 2001. Translating from relational model to object-oriented model. In the *International Conference on Information Technology and Multimedia (ICIMU 2001).*

STANISIC, P. 1999. Database transformation from relational to object-oriented database and corresponding query translation. In *Workshop on Computer Science and Information Technology CSIT*, p. 199-208.