Adding an Object-Oriented Interface to Relational Database Using Frame Model[1]

Joseph Fong
Department of Computer Science, City University of Hong Kong
Tat Chee Avenue, Hong Kong, E-mail : csjfong@cityu.edu.hk

San Kuen Cheung
Department of Computer Science, City University of Hong Kong
Tat Chee Avenue, Hong Kong, E-mail : cheungsk@asiaonline.cityu.edu.hk

An object-oriented interface can be implemented into a RDBMS as an application program interface for communicating between object-oriented concepts and relational database. We apply frame model approach for constructing the object-oriented interface. It includes static and dynamic behaviors of data modeling in a meta-data as a knowledge representation, and is user friendly for the purpose of transforming a RDBMS into an ORDBMS. Its function consists of schema translation from OODB to frame model and OFMA (Object Frame Model Agent) to interpret frame model for OO operations and method calls.

The schema translation consists of schema and method translations. The object-oriented schema can be translated to frame model and relational schema. The method translation consists of three steps: translating method signature to Persistent Stored Modules routine, translating method source language to routine source language, translating method invocation to routine invocation. According to our approach, object-oriented schema can be executed in the relational environment assisted by the frame model and case statement, which lists all possible cases of binding conditions and actions.

The Object-Oriented Interface is the knowledge representation constructed by frame model. OSQL (Object Structural Query Language) statements can be translated into SQL (Structural Query Language) statements from the static data of the frame model. OO operations in method calls can be preformed from the dynamic data of the frame model. The OFMA consists of Server API. The significance of the finding in the reengineering of object-oriented database system into relational database system is for database interoperability and for developing an object-relational database management system.

Keywords : Schema Translation, Method Translation, Object Frame Model Agent, Meta-data, Knowledge Representation, Persistent Store Modules, Middleware.

1. INTRODUCTION

In the current situation, many companies are using the relational concepts and products for assisting their business decision. OODB is still immature in the industry due to reliability, performance, cost of migration, and lack of expertise. Therefore, a legacy system using OODB is our objective. Research papers have introduced related works regarding the migration from OODB to RDB and vice versa. The current market has many ORDBMS (Object-Relational Database Management Systems). In this paper, we have developed an object-relational database using a frame model to handle the object features. After implementing frame model in a RDB, users can apply object features in their relational database, by adding an Object-Oriented interface in the relational model. The interface acts as a virtual interface.

[1] This paper is funded by Strategic Research Grant 7001078 of City University of Hong Kong

It has four classes: Header, Attribute, Method, and Constraint. Users can access the data either in OSQL or SQL.

By adding up the object technology in relational database, the database can be increased in the reusability and portability. On the other hand, the existing data of the relational model can be reused without interrupting daily operations of the existing system. After implementing our interface, companies can realistically utilize the benefits of both object-oriented and relational database systems.

2. Related Works

Research papers have introduced related works such as schema translation, data type mapping, query translation and meta-data. In the schema translation, M. Blaha[1] discussed converting OO Models into RDBMS schema. The approach combined OMTool with Schemer. J. A. Orenstein[2] designed a schema mapping gateway for accessing RDB through an OODB interface. In the data type mapping, S. DeFazio[3] extended RDBMS for handling complex domains. The approach required applications to integrate layered products from independent software vendors (ISVs) to deliver specific functionality for a domain. In the query translation, C. Yu[4] discussed the translation from object-oriented queries to relational queries. They proposed a method that used the extended OODB predicate graphs and relational predicate graphs to facilitate the query translation. In the meta-data, J. Lim[5] used meta-data and mapping libraries to extend ODBMSs by using metaclasses. It could turn traditional classes into normal objects, which could receive messages containing attributes and methods that described the database's structure and behavior.

Krishnamurthy[6] stated that Oracle server can be enhanced by developers to create their own application-domain-specific data types so as to bring the Object-Relational technology to the mainstream. It introduced a server-based components called Data Cartridges to integrate these new domain types as closely as possible with the server so that they could be treated at par with the built-in types like Number or Varchar.

Informix[7] packages a collection of data types. Their associated functions and operators access methods into DataBlade modules using the metaphor. The DBMS is a razor into which DataBlade modules are inserted. An Object-Relational Database can be formed using the DataBlade modules.

Software AG's Bolero provides an enterprise-wide middleware called EntireX for integrating new generation application components with the existing database systems. Users can apply Object-Oriented applications in a relational DBMS.

In Informix[7], an Object-Relational DBMS has four characteristics including Base Data Type Extension, Complex Objects, Inheritance, and Rules. These characteristics deal with the domain integrity. It also introduced how to code user-defined functions. Functions could be written by host language such as C, VB, and Java. To register a function, we must indicate its name, arguments, and return type. And then, the function could be freely used in any query statement.

3. FRAME MODEL (Meta-data)

In this paper, we employ the Frame Model Approach[8] to form an OO API for handling the OO operations in a RDBMS.

Advantages of using the Frame Model Approach

The frame model constructs the object-oriented paradigm and acts as meta-classes[9]. All conceptual entities are modeled as objects. The same attribute and behavior objects are classified into an object type. An object belongs to one, and only one class. Both facts and actions are objects in the frame model. The frame model is implemented with a knowledge representation that includes object structure classes, user-defined relationships between entities, and structure inheritance classes defined by taxonomies of structure that support data and behavior inheritance.

Procedural attachment is a useful concept because it allows frames to tie together the procedural and the declarative knowledge about an entity or class of entities. A complete translation of frame languages into logic also has to deal with this procedural information. One way of achieving this would be by providing a translation algorithm into logic in which the routines can be defined[10,11].

We employ the Frame Model because the model can easily be modified according to different situations. It can be formed as a meta-data during the schema translation and can be constructed as an application program interface for database interoperability. Also, it allows designers to group the rules together into a class and to associate the classes as a knowledge-based system.

Overview of the Architecture

The approach consists of frame model, schema translation, method translation, transaction translation, object frame model agent and relational database engine. The architecture of ORDBMS is outlined in the Figure 1. A relational schema and the frame model will be generated in the process of the schema translation. All the table definitions will be defined in the relational schema according to the OO schema. The frame model contains four system classes which are called header, attribute, method, and constraint. The interoperability between an object message and a relational database engine relies on the object frame model agent. An object-oriented interface is a virtual interface by combining the transaction translation and the object frame model agent.

The mechanism of the **Schema Translation** is that translating the Object Definition Language (ODL)[12] to Data Definition Language (DDL) will be run and the result will be stored into the frame model and the relational schema. The process will also implement a relational schema for a relational database.

In the **Method Translation,** a methodology is introduced for translating methods of an object-oriented database into routines (or stored procedures) of a relational database. The approach consists of three steps. The first step translates the method signature to the Persistent Stored Modules(PSM) signature. The second step translates the source language of methods to the target language of routines (consisting of either functions or procedures) including Host Language, OSQL's Qualification, Query Translation, Update Transaction Translation and Objects Inside Object. The third step translates method invocation to routine invocation. According to this approach, methods can be translated to routines and executed in the relational database environment assisted by a frame model and case statements which list all possible cases of binding conditions and actions.

Since our base engine is a RDBMS, the database will be in the relations. The **Transaction Translation** is to convert OSQL statements to SQL statements. Users can apply OSQL on a relational database because the translator will translate OSQL statements to target SQL statements for the relational engine.

The **Object Frame Model Agent (OFMA)** receives SQL statement from the result of the transaction translation. The SQL statement is executed by Server API.

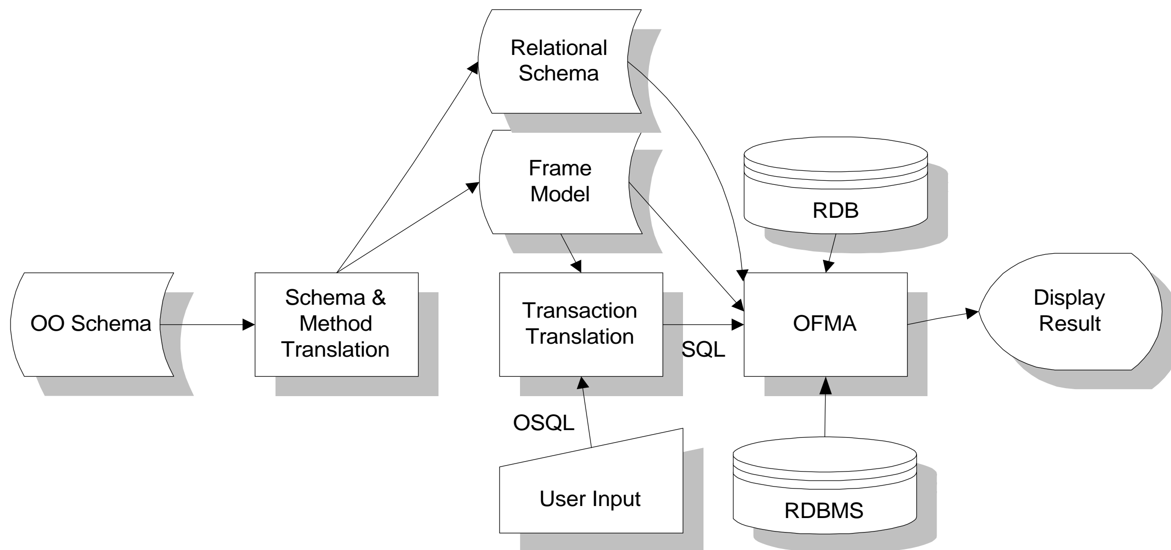

Figure 1 - Architecture of Object-Oriented Interface

Frame Structure

During the schema translation, the proposed translator will generate two semantics of a frame model[5]: static and dynamic semantics. The static semantic consists of Header and Attribute classes and the dynamic semantic consists of Method and Constraint classes. The Header class includes basic object information to represent the class entity. The Attribute class represents the properties of classes. The Method class includes actions to extend the semantics of the data. The Constraints class consists of properties of data that cannot be captured in the form of structure.

Header Class

```
{   Class_name          // a unique name in all system
    Parents             // a list of superclass names
    Operation           // program call for operations
    Class_type          // active or static class }
```

Attribute Class

```
{   Attributes_name     // an attribute in this class
    Class_name          // reference to header class
    Method_name         // a method in this class
    Attributes_type     // attribute data type
    Associated_class    // linkage class
    Default_value       // predefined value
    Cardinality         // single or multi-valued
    Description         // description of the attributes }
```

Method Class

```
{   Method_name         // a method component in this class
    Class_name          // reference to header class
    Parameters          // No. of arguments for the method
```

```
        Method_type         // return type of method
        Condition           // the rule conditions
        Action              // the rule actions }
```

Constraint Class

```
{       Constraint_name     // a constraint component of this class
        Class_name          // reference the header class
        Method_name         // method name constrict
        Parameters          // No. of argument for the method
        Ownership                 // the class name of method owner
        Event               // triggered event : create update or delete
        Sequence            // method action time : before or after
        Timing              // the method action timer }
```

The Header class includes basic object information to represent the class entity. The header class includes the class name, parents, operation, and class type attributes. The **Class_name** is a unique identifier of the class in the system. The **Parents** lists the parents of this class. The **Operations** shows all the triggered events. The **Class_type** describes the type of the class.

The Attribute class represents the properties of a class. The structure of the attribute class includes class name, attribute name, attribute type, default value, cardinality, method name and description attributes. The **Attribute_name** is a unique identifier of the attribute in a class. The **Class_name** is the name of owner class for this attribute. The **Method_name** is the method name where the attribute is derived or calculated from. The **Attributes_type** describes the data structure type of the attribute. The **Associated_class** describes the linkage between classes. The **Default_value** specifies default value of an instance. The **Cardinality** is used to deal with the single-valued or multiple valued attribute situation. The **Description** explains the purpose of this attribute.

The Method class includes rules to extend the semantics of the data. The structure of the method class includes method name, class name, parameters, method type, condition, and action attributes. The **Method_name** is a unique identifier of the method in a class. The **Class_name** is the name of owner class for this method. The **Parameters** is the number of parameters to be need for this method. The **Method_type** is type of the action of the method. The **Condition** stores the constraint name for taking the action. The **Action** contains the execution statement in the method.

The Constraints class consists of many properties of data that cannot be captured in the form of structure. The structure of the constraint class includes constraint name, class name, method name, parameters, ownership, event, sequence, and timing attributes. The **Constraint_name** is a unique identifier of the constraint in a class. The **Class_name** is the name of owner class for this constraint. The **Method_name** is the name of method for this integrity constraint and can be found in the method class. The **Parameters** represents number of parameters to be used in the method. The **Ownership** indicates that the method belongs to the class itself, the system class or another external class. **Event** shows the specific condition of the method to be triggered. The **Sequence** represents the sequence in which the method will be active. The **Timing** specifies the life of the method.

4. SCHEMA TRANSLATION

Translating O-O Schema to Relational Schema and Frame Model

For constructing a relational schema and a frame model, we must resolve the object-oriented features for RDBMS and create relationships among tables.

Step 1 – Classes to Relations/Frame Model
When converting class names in the OODB to table names in the RDB, all the class names will be mapped to the corresponding table names in the relational model. We define this mapping as follows.

Class C1 (…) , Class C2 : C1 (…) → Relation R_{c1} (…), Relation R_{c2} (…)

Step 2 – Inheritance/Generalization Mapping
Identifying a superclass and a subclass relationship, we expand attributes and methods of the superclass to its subclasses in class hierarchy. For the single inheritance, the attributes of the superclass are positioned first in the attributes definition list, followed by those that are locally defined. For multiple inheritance, the attributes of each superclass are concatenated in the order in which the superclasses were defined for that entity.

Class C1 (A1:type1, A2:type2) with void MC1 (void),
Class C2 : C1 (A3:type3, A4:type4) with void MC2 (void) →
Relation R_{c1} (.. .., $A1_{rc1}$:type1, $A2_{rc1}$:type2),
Relation R_{c2} (.. .., $A1_{rc1}$:type1, $A2_{rc1}$:type2, $A3_{rc2}$:type3, $A4_{rc2}$:type4)

Frame Model - Header Class

Class name	Parents	Operation	Class type
C1			Static
C2	C1		Static

Frame Model – Attribute Class

Attribute name	Class name	Method name	Attribute type	Attributed class	Default	Cardinality
A1	C1		Type1			
A2	C1		Type2			
A3	C2		Type3			
A4	C2		Type4			

Frame Model – Method Class

Method_name	Class	Parameters	Method type	Action
MC1	C1	void	void	C1_MC1
MC2	C2	void	void	C2_MC2

Step 3 – Identity Mapping

When creating object identifier for each table and recognizing the object identifier as a key field, the concept of primary key of the relational database can be established so as to link up the relationships among tables. The value of an OID is used for object identification for inter-object references. A key of corresponding relation should be defined to create a unique attribute. Map OID in OODB to key in RDB by adding extra atomic type attribute, denoted as K_{rc1}. OID for the object of class C1 is represented by K_{rc1} as the key in Relation R_{c1}.

Class C1 (OID) → Relation R_{c1} (K_{rc1})

Step 4 – Attribute Mapping

When translating the attribute names in the OODB to the attribute names in the RDB, we know that the field of the "attribute names" may contain actual names of attributes or methods. Therefore, we also extract method signatures from OO schema to the method class of the frame model for creating a method definition in the RDB. We define this mapping as follows.

Class C1 (A1:type1, A2:type2) with void MC1 (void) →
Relation R_{c1} (K_{rc1}:key, $A1_{rc1}$:type1, $A2_{rc1}$:type2)

Frame Model – Attribute Class

Attribute name	Class name	Method name	Attribute type	Attributed class	Default	Cardinality
A1	C1		Type1			
A2	C1		Type2			

Frame Model – Method Class

Method_name	Class	Parameters	Method_type	Action
MC1	C1	void	void	C1_MC1

Step 5 – Multiple-valued Attribute Mapping

When reorganizing the set attributes and creating new tables for replacing the set values, we map each element in multiple-valued attribute S_c of class C1 to a new relation R_{sc1} which contains key attribute K_{rc1}:key and the attribute of original type of multiple-valued element. This is 1 : N relationship in the relational database.

Class C1 (A1:type1, S_c:set(type2)) →
Relation R_{c1} (K_{rc1}:key, A1:type1)
Relation R_{sc1} (K_{rc1}:key, S_c:type2)

Frame Model - Header Class

Class name	Parents	Operation	Class type
C1			Static
SC1			Static

Frame Model – Attribute Class

Attribute name	Class name	Method name	Attribute type	Attributed class	Default	Cardinality
A1	C1		Type1	SC1		M
S_c	SC1		Type2	C1		S

Step 6 – Aggregation Attribute Mapping

To atomize the user-defined data types and create object identifier to link the relationship with composite objects, we map class type attribute A_c of class C2 to a key type attribute as A_{rc} in relation R_{c2}. A foreign key references to the relation is mapped from other class C1 where attribute A_c of class C2 is referred to. This is 1: 1 relationship in relational database

Class C1 (A1:type1, A2:type2), Class C2 (A3:type3, A_c:C1) →
Relation R_{c1} (K_{rc}1:key, A1:type1, A2:type2)
Relation R_{c2} (K_{rc}2:key, A3:type3, A_{rc}: K_{rc}1)

Frame Model – Attribute Class

Attribute name	Class name	Method name	Attribute type	Attributed class	Default	Cardinality
A1	C1		Type1	C2		S
A2	C1		Type2			
A3	C2		Type3			
A_c	C2		Type4	C1		M

Step 7 – Cardinality/Association Attribute Mapping

Creating two new tables for n:m relationship, the values point to another tables. We define this mapping as follows.

Class C1 (A1:type1, …, S_{c1}:set(C2)), Class C2 (A2:type2, …, S_{c2}:set(C1)) →
Relation R_{c1} (K_{rc1}:key, A1:type1, …)
Relation R_{c2} (K_{rc2}:key, A2:type2, …)
Relation R_{sc1} (K_{rc1}:key, $S_{c1:}K_{rc2}$)
Relation R_{sc2} (K_{rc2}:key, $S_{c2:}K_{rc1}$)

Frame Model - Header Class

Class name	Parents	Operation	Class type
C1			Static
C2			Static
SC1			Static
SC2			Static

Frame Model – Attribute Class

Attribute name	Class name	Method name	Attribute type	Attributed class	Default	Cardinality
A1	C1		Type1	SC1		M
A2	C2		Type2	SC2		M
S_{c1}	SC1		Type3	C1		S
S_{c2}	SC2		Type4	C2		S

We must change the data types in the target relational model; but they depend upon specific product of the relational model. Different products provide different data types.

5. METHOD TRANSLATION

In the object model, the method definition is defined in the object schema. In the relational model, the routine definition is not a part of the relational schema. During the schema translation, a method can be considered as an attribute in which the value is not statically stored in database, but the body is dynamically calculated by executing from a related program. The method signature must be identified and stored in meta-data (see Figure 2). A combination of the method name and its parameters within a class can be identified as a specified method uniquely. This is a feature of the object model. In the relational schema, there is no such a mechanism that supports user-defined functions and procedures. Each routine name must be identified a routine uniquely. Commercial SQL products have been offering such a capability for years[13] in the form of stored procedures and the SQL standard has also begun considering to include this capability of PSM (Persistent Store Modules) routine[14], in both user-defined functions and procedures.

In the object model, source code of methods can be defined by the object definition language (ODL) in the schema level. A method source can be constructed by combining embedded OSQL and host language (e.g. C or C++). In the relational model, source code of routines is defined in the operational level. A routine source is constructed by combining embedded SQL and host language. In this situation, we do the method translation.

Methods can be operated by OSQL in which a method can be invoked by an object feature of the navigation. Routines can be either functions or procedures. A user-defined function is invoked by scalar expressions. A user-defined procedure is invoked by a new SQL statement (typically "do"). During the method translation, all the object methods will be translated to relational procedures. The routine invocation is assisted by a case statement which lists all possible cases of binding procedures. The method/routine invocation is the final step of our approach.

Meta-data (i.e. Frame Model) is created for containing information of the object model during the schema translation. The frame model in the method translation is composed by Header, Attribute, and Method classes. The Header class is to represent class relationship. The Attribute class represents the properties of classes. The Method class contains information of each method.

Architectural of Method Translation

The mapping process consists of three parts. The first is to translate a method signature[12] to a frame model. The second is to translate a method source to a PSM routine source. The third is the invocation

translation which relies on the meta-data (i.e. frame model). We treat the frame model as the meta-data during the method translation. A combination of a method name and its parameters within a class can locate an "action" attribute which can invoke a specified routine call.

The main procedure of the translation can be divided into three steps. Step 1 is the Schema Translation where the method signature will be mapped to the PSM routine and stored in a frame model. Step 2 is the Method Source Translation where object methods will be converted to relational routines. Step 3 is the Invocation where dynamic binding of an object feature can be applied in the relational database.

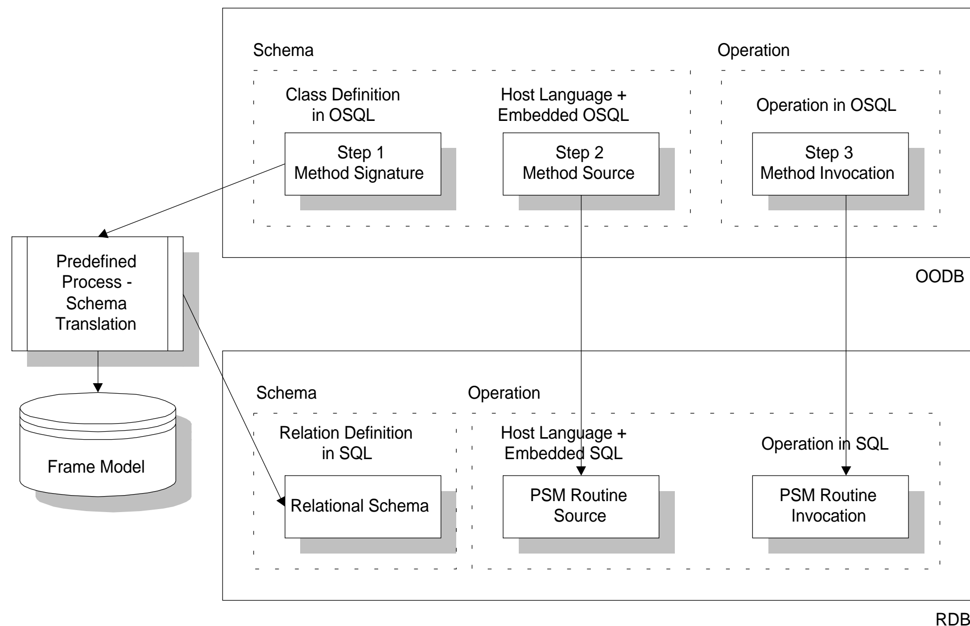

Figure 2 – Mapping from Object Method to PSM Routine

Method signature to PSM's routine

A method signature is composed by three elements of class name, method name, and parameter list. The class name represents the ownership of the method. The method name is the name for the execution area. The parameter list contains major elements of the execution. Mapping the method name must avoid the name conflicts because a routine name is not bound by a specific relation in the relational model, whereas a method name is bound by a class. It is necessary to assign a unique name for each routine and the name will be recorded in the meta-data.

Mapping the parameters listed in a method requires each data type of parameter related to class to be converted to a corresponding data type related to relation that spans the same data content as the class. This mapping can be determined by observing the data type mapping from classes to relations.

Mapping the return types is the same as mapping the parameters. The return types of methods are stored in the "method" class as a meta-data. The relational types of the return values are defined by mapping the object data types and the relational data types. Inside the meta-data, the return types are the relational preferences.

Method Source Translation

The statements of embedded OSQL must be mapped to the embedded SQL DML (data manipulation language). The statements of host programming language which deal with the flow control and operations on volatile data do not need to change (assuming that host languages are the same for both OODBMS and RDBMS, otherwise only key words and statement formats must be adjusted). The statements of host programming language which deal with operations on non-volatile data must be modified according to data type mapping table from classes to relations. Mapping the body of a method requires distinguishing the above statements and different statements may employ different mapping rules.

Rule 1 : Path expression operand translation
Rule 2 : Set operand translation
Rule 3 : Query translation
Rule 4 : Update transaction translation
Rule 5 : Host language data type translation (for non-volatile data)
Rule 6 : Objects inside object

An object method consists of embedded OSQL statements and host language[4]. The main problem of the method translation is on the embedded OSQL statements. There are many issues such as qualification translation, query translation[10], and update transaction translation. We use path elements to handle the qualification translation (i.e. where-clause). The idea of the path elements is that we will locate the path expression of an attribute or a method. The meta-data contains all the information of each attribute or method from the schema translation. The path elements can be located from the meta-data. After forming a path logic of a specific attribute, we can evaluate all the elements in the path logic. The elements of the path logic can be translated to a join operation in the RDB system if it has three or more elements. The qualification translation is not only on the path expression operand, but also on the set operand.

The set operand in the object model can be treated as an object feature of multiple-valued attribute. The feature provides a searching algorithm for the multiple values. Our resolution for the multiple-valued attribute in the relational schema is to create a new individual table in which each value of the multiple-valued attribute can be stored in each individual tuple. The location of this attribute can be obtained from the semantic formed by the meta-data.

We must consider the method compiler from two systems (object and relational engine). A common host language such as C or C++ can be accepted by both systems. The host language may not be necessary to convert. Variables of the host language which deal with operations on nonvolatile data must be considered according to data type mapping table from classes to relations. During the schema translation, classes of the object model are converted to tables of the relational schema. The object definition and class attributes will no longer exist on the relational database. In this case, we must extract these variables from the object method and convert the data structures of these variables for operations on the relational database.

Rule 1 - Path Expression Operand Translation

In path expression, address of a method or an attribute is an important part of the statement. The translator will go through the path logic for locating the method or attribute and will execute the method or retrieve the value of the attribute. We must know how to form a path logic and to reconstruct the logic

for relational SQL statement. The first step is to decompose the object path which consists of composite attributes or other user-defined methods. The composite attribute will return its value to operator in comparison with other value. The method will execute its calculation and return the result to operator for doing the same comparison as the composite attribute. The path logic of a class can be formed according to the meta-data because important information is stored on the meta-data and on the static tables. The second step is to organize the path logic for relational SQL statement. If the path logic has three elements, we must perform a join operation in the relational SQL statement.

The meta-data is a definition of the relational schema and the relational tables are static data. The path logic can be constructed by these two components. We must know the relationship among tables. The schema definition can provide a whole picture of linkage of the object model. The static data is an actual storage. The path logic of the relational database is shown in the following algorithm.

Algorithm :

```
program path_expression   // This is for decomposing path expression//
WHILE not at end of comparison operator’s elements DO // operators in “where-clause”//
begin
     get elements from operator; // left or right side totaling 3 elements (e.g. A.ar.arr = 11)//
     get components of each element; //each element has its components  (e.g. A.ar.arr)//
     map components from header & attribute class of frame model;
    //collecting table name & association (i.e. A table and ar’s table as a path logic)//
     save table and its sub-ordinate tables to &tables;
     //for constructing join operation for SQL statement//
     get data type from attribute class of frame model; //e.g. attribute arr’s data type//

     CASE data type of
          “preliminary data type” : save attribute to &preliminary
          IF attribute in “where-clause”
          THEN begin
               put join operator(s) for &tables into SQL if necessary
               put &preliminary into SQL for comparing with “11”
                  end;
          “method parameter” : save the name of action attribute to &method
          IF attribute in “where-clause”
          THEN put &method and parameters into SQL;
     END-CASE;
end;
```

In this path expression translation, we analyse the composite attributes in the where-clause. The translation can be applied to other composite attributes within the OSQL statements. For instance, you can apply Rule 1 to analyse the composite attributes in the “select-clause” in query statement and “set-clause” in update statement. Now, the where-clause has been constructed a true relational syntax according to the composite attribute of the object-oriented navigation.

Rule 2 - Set Operand Translation

We are focusing to analyze a set attribute on the where-clause. The path logic of the set attribute involves a new individual table that contains the multiple values of a set attribute. We decompose the

multiple values into a new table during the schema translation. The new table is created for containing the multiple values of each multiple attribute. Primary key on the existing table and composite key on the new table link the connection between the existing table and the new table up. The composite key consists of two attributes. The following is an algorithm for the path logic of the set attribute.

Algorithm :

```
program set_operand // This is for decomposing the set operand//
WHILE not at end of comparison operator's elements DO // operator in "where-clause"//
begin
    get elements from operator; // left and right side totaling 3 elements//
    get components of each element; //each element has its components//
    map components to header and attribute classes of frame model;
    //collecting table names & association between existing and new tables//
    get data type from attribute class of frame model;
    IF attribute data type = table name with prefix of "new" located in header class
    // i.e. set attribute by verifying against data type in attribute class//
    THEN begin
        save table and sub-tables from attribute class of frame model to &tables;
        get attribute from attribute class of frame model;
            IF attribute type = preliminary data type //.i.e. multiple-value attribute//
            THEN save attribute to &preliminary;        // i.e. attribute from new table//
                IF attribute in "where-clause"
                THEN begin
                    put join operator(s) for &tables into SQL if necessary
                    put &preliminary into SQL
                end;
            end;
    end;
end;
```

The &table and &preliminary are temporary variables that are important part in constructing the path logic. The existing and new tables will be stored in the array named &tables. The name of the multiple-value attribute in the new table may be changed for internal identification. We use &preliminary as a temporary storage for this name. The decision of identifying an attribute with multiple-value is relied upon the prefix with "new" in the "data type" attribute of the "attribute" class. The name will be confirmed by locating the name in the "header" class. A true relational syntax can be formed on the SQL statement according to the set operand of the object-oriented navigation.

Rule 3 - Query Translation

A query statement consists of range part (select-clause), target part (from-clause), and qualification part (where-clause). From the range part, we apply rules 1 & 2 on composite and set attributes. After getting some relevant tables from the range part, these tables will be placed on the target part. As for the qualification part, we also need to consider (i) tables to be involved, (ii) methods to be invoked, (iii) join operations to be executed. Table names on the target part are factor for forming join operations. Rules 1 & 2 are solutions for decomposing the navigation on the qualification part. Methods will execute their calculations and return results to operator before the comparisons with other values.

Algorithm :

```
program query_translation
// This is for translating OSQL query statement to SQL query statement//
WHILE not at end of attribute list DO // analyzing the “select-clause” of the OSQL//
begin
        CASE attribute of
                “preliminary data type” : begin
                        save table from attribute class of frame model to &tables;
                        save attribute from attribute class of frame model to &preliminary;
                        end;
                “multiple value” : begin
                        call set_operand                //applying rule 2//
                        end;
•               “composite value” : begin
                        call path_expression  //applying rule 1//
                        end;
        END-CASE;
end;
put &tables.&preliminary into SQL;           //putting in “select-clause”//
put &tables into SQL;                        //putting in “from-clause”//
WHILE not at end of logical operator’s elements DO  // for “where-clause”//
begin        //e.g. “A.ar = 11 and”, the logical operator = “and”//
        save logical operator to &logical_operator;
        CASE attribute of
                “preliminary data type” : begin
                        save table from attribute class of frame model to &tables;
                        save attribute from attribute class of frame model to &preliminary;
                        end;
                “multiple value” : begin
                        call set_operand                //applying rule 2//
                        end;
•               “composite value” : begin
                        call path_expression  //applying rule 1//
                        end;
        END-CASE;
        put &tables.&preliminary into SQL
        put &logical_operator into SQL; //putting in “where-clause”//
end;
```

Firstly, we extract the attributes in the “select-clause” one by one for query translation. Some relevant tables will be stored in a temporary variable named &tables. The attribute name will also be stored in a temporary variable named &preliminary and will later be printed in the SQL’s “select-clause”. Secondly, we print all the tables involved from &tables to the “from-clause”. Finally, we analyze the qualification part (i.e. “where-clause”) that may be divided by several sectors. These sectors will be broken by the logical operators. Some join operations may be formed according to how many path elements in the path logic.

Rule 4 - Update Transaction Translation

In the update transaction translation, the target part (update-clause), range part (set-clause), and qualification part (where-clause) compose an update transaction statement. To identify attributes of the range part, we apply rules 1 & 2. This part is to determine how many update SQL statements to be performed. If an attribute contains multiple values, then we execute more than one update SQL statement. An attribute with single value is easy to update as we do not have to consider the existing value. We have to consider the existing values before going to update an attribute with multiple values. These values are stored in an individual table in which the key is a composite key. To update a tuple with multiple values, we must know the old value and OID value. The qualification part is similar to the query translation.

Algorithm :

```
program update_transaction        // This is for translating update statement//
WHILE not at end of comparison operator's element DO //analyzing "set-clause"//
begin
        save comparison operator to &comp_operator;
        get elements from the comparison operator;                //left or right side//
        get components from each element;
       IF data type = multiple value
       THEN call set_operand              //applying rule 2//
       ELSE IF data type = path expression
       THEN call path_expression         //applying rule 1//
       ELSE IF data type = preliminary data type
       THEN begin
                save table from attribute class of frame model to &tables;
                save attribute from attribute class of frame model to &preliminary;
       end;
       put new value from OSQL to &new_value   //It can be a multiple-value attribute//
       IF value = multiple value
       THEN begin
                get oid value and old value from RDB table;
                save oid value and old value to &oid_value and &old_value;
        end;
end;
WHILE not at end of logical operator's elements DO          // for "where-clause"//
begin
       save logical operator to &logical_operator;
       IF "preliminary data type"
       THEN begin
                save table from attribute class of frame model to &tables;
                save attribute from attribute class of frame model to &preliminary;
                 end
        ELSE
                call path_expression and set_operand //applying rules 1 and 2//

end;
WHILE not at end value of &new_value DO
```

```
begin
    put &tables into SQL;        //putting in "update-clause"//
    put &preliminary, &comp_operator, &new_value into SQL; //putting in "set-clause"//
    put &oid_value and &old_value into SQL;          //putting in "where-clause"//
end;
```

In OSQL's update statement, a multiple-value attribute can be updated by one statement. In SQL's update statement, we need to do many steps when updating a multiple-value attribute. Firstly, we must know the cardinality of an attribute to be updated. For a multiple-value attribute, we analyze tables to be involved and attribute names to be changed respectively. The tables and attributes will temporarily be stored in variables named &tables and &preliminary. We save the new values to an array named &new_value. Secondly, we extract the old values and an oid value from database and store these values in variables named &old_value and &oid_value. Thirdly, we check that the "where-clause" is the same as the query statement. Finally, we can form a SQL statement by grouping all the variables.

Rule 5 - Host Language Data Type Translation (for Non-volatile Data)

The problem is on the data type. We do not know which host variable will be stored data extracted from the relational database or compared with data from the relational database. We will convert all the data types of the host variables to the relational model. When the data type of host language is not matched to the data type of the host variable, the target application will then help us to solve the problem. An error procedure will be started, when a dismatch message of the data type occurs. The error procedure consists of a mapping table and a conversion process. The function of the mapping table is to map the relational data type to a corresponding object data type. And then, the relational data will be converted to object data (see Figure 3). In this approach, we do not have to worry about the host variable which deals with nonvolatile data.

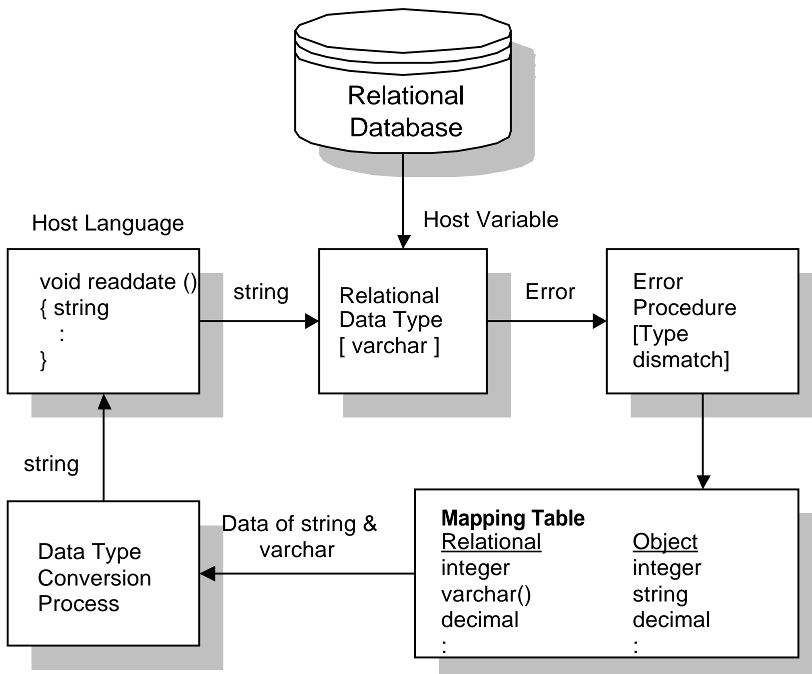

Figure 3 – Mapping Data Types

Algorithm :

```
program host_language      // This is for translating data type//
begin
        get value from database;
        get mapping table;
        map source data type to target data type;
        map target data to source data;
        copy the value to source;
        copy the source value to program;
end;
```

Rule 6 - Objects Inside Object

An object can be a method. The method can invoke itself or other objects. We must convert the signatures of the object model to relational model so as to invoke the appropriate routines. All the routine definitions are defined in the "method" class of the frame model instead of the relational schema. We collect the information of these methods from the case statement. We find the routine names from the "method" class of the frame model.

Algorithm :

```
program objects_inside      // This is for translating objects inside object//
begin
        get method name from OSQL;
        map the method name from case statement;
        map the method name from frame model;
        get the routine name from frame model;
        put the routine name into SQL;
end;
```

Invocation

We must enlarge the existing features of the relational database because the RDBMS cannot dynamically resolve the polymorphism. If we don't change the method name during the schema and the method translations, the name of routine or stored procedure will not be a unique key in the standard of the SQL-92 and will not overcome the object effect of the polymorphism. The routine names must be unique in the relational database. We form an "action" attribute to store the routine name. It can be a code such as OID stored in the "method" class of the frame model. Since the RDBMS does not have the dynamic binding, if we translate the method name directly to routine name without any changes, a specified routine may not be invoked by PSM's "call" or "do" statement. The "call" or "do" statement is for calling a procedure and the "select" statement is for calling a function.

Methodology of Invocation Translation

The method invocation can appear in two different formats. The first one is like a procedure call, an invocation statement. The second one is like a function call, an operand in a scalar expression which is part of an OSQL statement. Mapping the method invocation must consider the polymorphism (dynamic binding) feature. It is necessary to map a method invocation to a corresponding PSM routine invocation because RDBMS cannot dynamically resolve PSM routines in the same way that OODBMS can do for

methods at run time. It requires that mapped target for a method invocation should involve dynamic resolution of a PSM routine as well as invocation of the routine. One way is to use, in case statement which lists all possible cases of binding conditions and calls of related PSM routines under each case.

The Polymorphism, the Inheritance, and the Multiple Inheritance are features of the object model. We are going to discuss each possible case occurred in the relational model.

Polymorphism
Class definition
A (A1,…, M1 (a), M1 (a,b))

Relational definition
A (A1,…) with A_M1_1, A_M1_2

Inheritance
Class definition
A (A1,…, M1 (a), M1 (a,b))
B (B1,…, M1 (a)) subclass of A
C (C1,…, M1 (a,b,c) subclass of A

Relational definition
A (A1,…) with A_M1_1, A_M1_2
B (A1, B1,…) with B_M1_1, A_M1_2
C (A1, C1,…) with C_M1_1, A_M1_1, A_M1_2

Multiple inheritance
Class definition
A (A1,…, M1 (a), M1 (a,b))
B (B1,…, M1 (a)) subclass of A
C (C1,…, M1 (a,b,c) subclass of A, B

Relational definition
A (A1,…) with A_M1_1, A_M1_2
B (A1, B1,…) with B_M1_1, A_M1_2
C (A1, B1, C1,…) with C_M1_1, A_M1_1, A_M1_2

Meta-data (Method class)

method name	class name	parameters	method type	condition	action
M1	A	a	void	nil	A_M1_1
M1	A	a,b	void	nil	A_M1_2
M1	B	a	void	nil	B_M1_1
M1	C	a,b,c	void	nil	C_M1_1

The method invocation in the case of the multiple inheritance : t.M1(a) (“t” is a variable of class C) is a method with parameter “a”. We use a case statement which lists all possible cases of binding routines. In this example, the routine name “M1(a)” should be read “A_M1_1”.

```
switch (t)
{       case t = class A : A_M1_1, A_M1_2
        case t = class B : B_M1_1, A_M1_2
        case t = class C : C_M1_1, A_M1_1, A_M1_2 }
```

After finishing the schema and the method translations, we have to introduce a translator for interpreting the incoming message that may be a DML or a method call. The process of the transaction translation will convert an OSQL statement to a SQL statement and pass it to an Object Frame Model Agent (OFMA) for executing the statement. The system will evaluate the statement from four frame classes and invoke the constraint and the action (if necessary).

6. TRANSACTION TRANSLATION

To translate OSQL in OODB to SQL in RDB, we may use the symbolic transformation technique. It contains two parts, syntax translation and semantic translation. In our model, the syntax descriptions of source and target languages have been built for the syntax translation. After the syntax of source OSQL has been verified, we can create a navigation path in the OODB that allows us to navigate the query graph[4] of the OSQL (OODB Class Table). Afterwards, referring to the result of the schema mapping, the navigation statement of the OSQL can be mapped to that of the SQL by the query graph of the SQL (RDB Relation Table) and generate the navigation statement of the SQL. Particular semantic rules for transformating from the OSQL to the SQL will be applied. The target SQL will be produced. The output SQL should be semantic equivalent to the source OSQL.

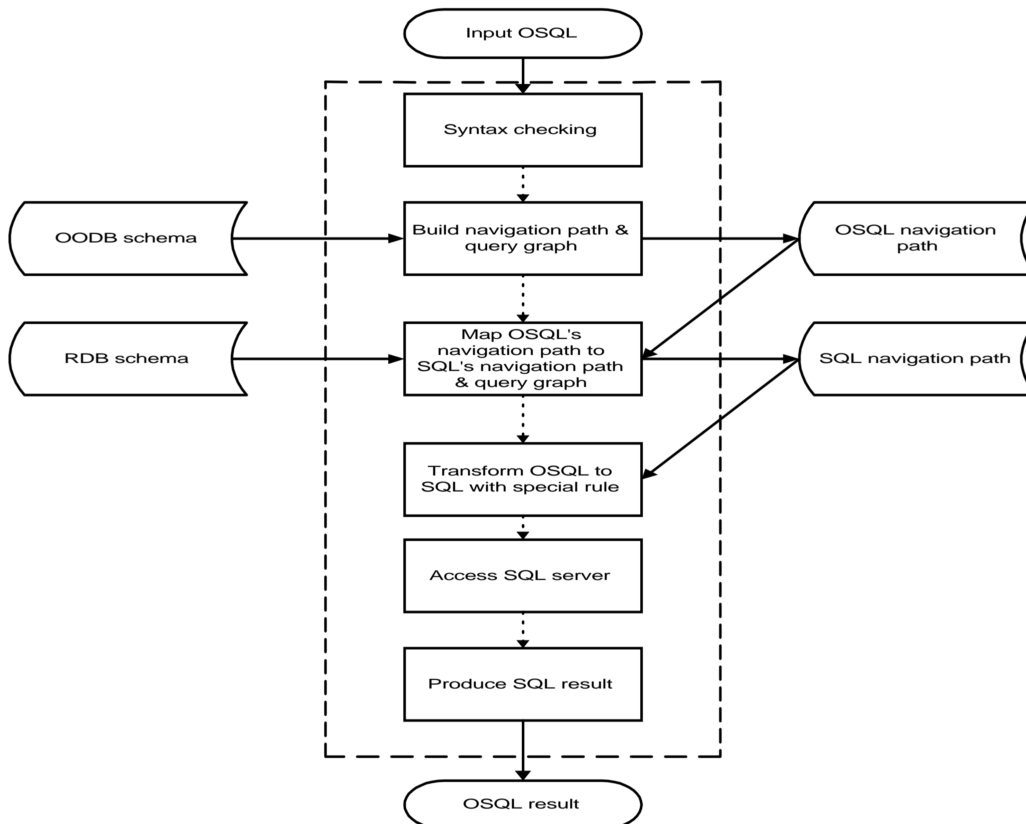

Figure 4 – Flow Chart of Transaction Translation

Overview of Transaction Translation

The following section will mainly show the particular semantic rules of transaction applying on **SELECT, UPDATE, INSERT** and **DELETE** transactions. As for the invocation statement, the process of translation is the same as the method invocation. Each table has its routines bounded by the case statement and the meta-data. This is a new feature in the relational model. We can apply the dynamic bounding in the relational database. The syntax shown below is the portion of query or the DML that can be translated to the SQL during this study. And the following is the flow chart of transaction translation from the OSQL to the SQL process.

Navigation Path

A navigation path is a function that maps one set of addresses (departure point) into another set of addresses (arrival point). The concept of the database navigation can help us to build relations access path in the translation process. To ensure the result is preserved after the translation, the output of the SQL statement must be the same as that obtained by the OSQL statement.

The Proof of Navigation Path

Let A be a set of addresses, N be a set of data names and V be a set of atomic values. Suppose that A, N and V are disjoint sets. We consider the content of a database as a database content that is a collection of addresses, data names and data values such that $CON \subseteq A \times N \times (A \cup V)$.

Let S be the set of navigational statements[15]. We denote by ||S|| the semantics of the statements. The function ||S|| maps a set of addresses into another set of addresses. ||S|| depends on particular database content CON, and is regarded as mapping of $\|S\| : \rho(A) \times CON \Rightarrow \rho(A)$, from one address of the database content, we can navigate to another address within the database content through pointers.

Let us consider a navigational statement $s \in S$, where S = S1, S2 .. Sk. If the statement S is applied to the set of addresses $A0 \subseteq A$, then we can define a trajectory of navigation in a sequence of address sets: <A0,A1..Ak> where A1 = ||S1||(A0), A2 = ||S1.S2||(A0), Ak = ||S1.S2…Sk||(A0).

To verify the equivalence between the OSQL and its translated SQL statement, we can show that their data are the same by expressing them in the same domain of navigation statement.

Assume the schema mapping preserved the nature of database, the domain of navigation is from same database, then navigation paths should be equivalent.

The technique of creating Navigation Path

The syntax of a navigation path is :

```
    Caluse_content [{; clause_content | inner_clause_content }…]
    Where,
       Clause_content :
           Table_name_list
           Attribute_name_list
           Expressions_list
       Inner_clause_content :
```

```
    '[' clause_content { ; clause_content | inner_clause_content … ']'
table_name_list :
     [path_expression . ]table_name [, table_name_list … ]
attribute_name_list :
     [path_expression . ]attribute_name [, attribute_name_list …]
expression_list :
    expression [, expression_list … ]
```

Translation of SELECT Statement

The basic syntax of OSQL query transaction is as follows

Step 1 Decompose OSQL query transaction – separating the transaction components into different listings as follows.

```
SELECT {attribute-list-1}
FROM {class-list}
WHERE {search-condition-1}
ORDER BY {attribute-list-2}
GROUP BY {attribute-list3}
HAVING {search-condition-2}
```

Step 2 Build navigation path of OSQL in order to navigate the query graph[16] (OODB Class Table) – based on the input OODB schema, an navigation path can be established to indicate the relationship between association/composite attribute and the referenced classes it related to.

Step 3 Map the OSQL query graph to SQL query graph – according the navigation path of OSQL, we locate the class to its corresponding relations in RDB by the pre-process schema mapping (RDB Relation Table), and generate the navigation path of SQL.

Step 4 Transform OSQL to SQL query transaction – according the navigation path of SQL and the particular semantic rules, place the target attributes and relations in the corresponding clause, such as SELECT, WHERE, and FROM clause, and use JOIN query technique to reference the corresponding relations that listed in SELECT clause_content.

Query with Path Expression

A path expression in OSQL begins with an optional class prefix to identify the class containing the attribute specified in the expression. Path expression is valid in any part of a query except FROM clause. The syntax of a path expression is as follow.

```
[class_prefix.] attr_name1['['var']'].attr_name2['['var']']…
```

the class_prefix is a correlation variable or the name of a user-defined class in the database. The attr_name1 can be the name of an instance attribute in the prefixed class, or interpreter variable. A selector variable can optionally be assigned to any attribute referenced in the path expression.

Scalar-valued Expressions (Query with Associate/composite Attributes)

The simplest form of path expression identifies scalar values. It can be specified in SELECT statements to access scalar attributes of user-defined classes in OODB. The JOIN query technique is used for translation process.

Set-valued Expression

A path expression can have more than one value if the domain of some attribute in the path expression is defined as a set. The set of types for any attribute can be a single built-in data type or single user-defined class. This set type is homogeneous. On the other hand, a set may also be heterogeneous; that is, the set may contain more than one built-in data type, or more than one user-defined class.

Translation of UPDATE Statement

The UPDATE statement allows you to change values of attributes associated with the instances of a class. A WHERE-clause can be specified if you want to conditionally update the value of an instance.

Once instances have been inserted into OODB, the attribute values of those instances can be updated. An update can be applied to a single instance or multiple instances. When you want to update a single instance, you have to construct one update SQL statement. When you want to update a set-valued instance, you have to construct more than one update SQL statement.

Translation of INSERT Statement

Instances are created when you insert data into class. In OSQL, when the domain of an attribute is a user-defined class, values can be inserted using nested INSERT statements.

The general semantic rules of inserting attributes are

1. As the OID of instance in class C1 does not exist in relation Rc1, a unique key value of corresponding relation Rc1 should be created.

2. When nested INSERT statements exist, there are two cases :

- Inserting associated attribute – treat the nested INSERT statement as individual INSERT transaction in SQL, and replace the position of the nested INSERT statement by a unique key value Kr. It acts as the foreign key of the associate attribute. The key Kr is now the unique key of the relation that the associated attribute referenced to.

- Inserting set-value attribute – treat the nested INSERT statement as individual INSERT transaction in SQL of the derived relation, R_{set}, (relation for set-valued attribute after schema mapping). The foreign key K_{set} in original relation should be equal to the K_{set} in derived relation R_{set}.

Translation of DELETE Statement

The DELETE statement allows you to discard data that comprises an instance of a class.

When an instance is deleted from the class C1, its OID is also cancelled. For instance of other class C2, if its stored associated attribute (its domain as the class C1) is referred to the delete instance, it will also be cancelled; i.e. set its value to NULL. In order to preserve the data consistence of OODB after the translation to RDB, the steps must be taken before the DELETE transaction :

1. In *Query* statement, the value of key attribute of the tuple is deleted.

2. Stored associated attribute – we need to set the foreign key attribute Kr1 (which refers to R1) in relation R2 to NULL, when the value of the domain in foreign key Kr1 is equal to the key of tuple being deleted.

3. If the instance being deleted has a set-valued attribute, query the value of foreign key of derived relation for set-valued attribute of the tuple being deleted.

4. Set-valued attribute – after schema mapping, the set-valued attribute of the class is mapped to the other relation R_s for set-valued attribute. If the tuple is deleted in relation R1, its corresponding tuples in relation R_s are also deleted.

The original delete transaction can then be performed.

7. OBJECT FRAME MODEL AGENT (OFMA)

The Object Frame Model Agent (OFMA)[17] is constructed by application program interface (API) that can manipulate both standard SQL and routine operations. The API is incorporated with an RDBMS in which the Frame Model, Routines (i.e. Store Procedures), and RDB's Tables are stored. The converted SQL statement has two ways to go. The first way is to process the relational algebra such as restrict, project, product, etc. The second way is to process the routine calls. Theoretically, the OFMA consists of a Server API and a RDBMS.

We can make the process of command scanner simple by illustrating in the following pseudo code. In this stage, the incoming message is identified for delivering to an API.

```
Begin
    Get input message
    Map the message to mapping table
    IF no syntax error
        case 1 = DML without data modification
            call DML interpreter in Server API
        case 2 = Query method with DML function type
            call DML interpreter in Server API
        case 3 = Query method with additional method type
            call Method interpreter in Server API
        case 4 = DML with data modification
            call DML command interpreter in Server API
    ELSE
        return with error
    ENDIF
End
```

The Server API Process

The Server API will scan the input message by the Command Scanner. When the message is a DML command without data modification, OFMA will parse the DML directly to RDBMS. When the message is a method, OFMA will decode the message and execute the “action name” from the “method class” of the Frame Model located in the routine file. The process of the Server API is shown in the Figure 5.

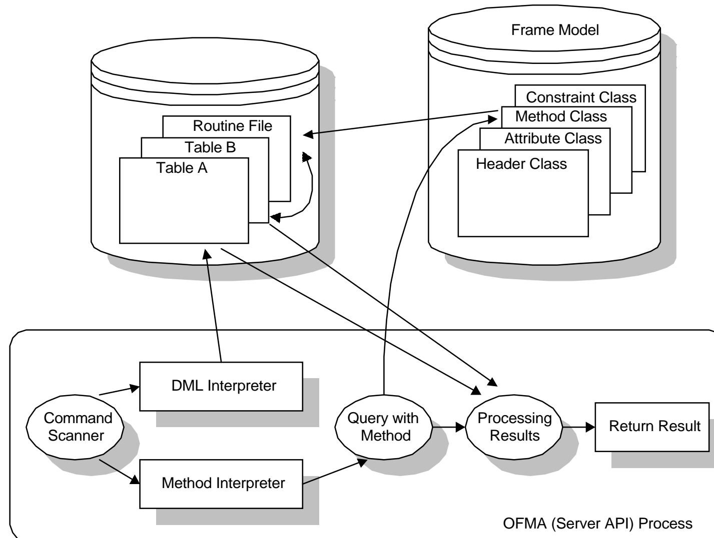

Figure 5 - Process Diagram for OFMA Server API

8. PROTOTYPE

Schema and Method Translation

A prototype is provided for testing our methodology in the object-oriented (Jasmine) and the relational (FoxPro) systems. We built up same definition and data in the two systems and translated a method to a routine by our methodology. Then we ran the method and the routine in two systems. As a result, two systems provided the same output. The data flow (see Figures 6 and 7) is as follows :

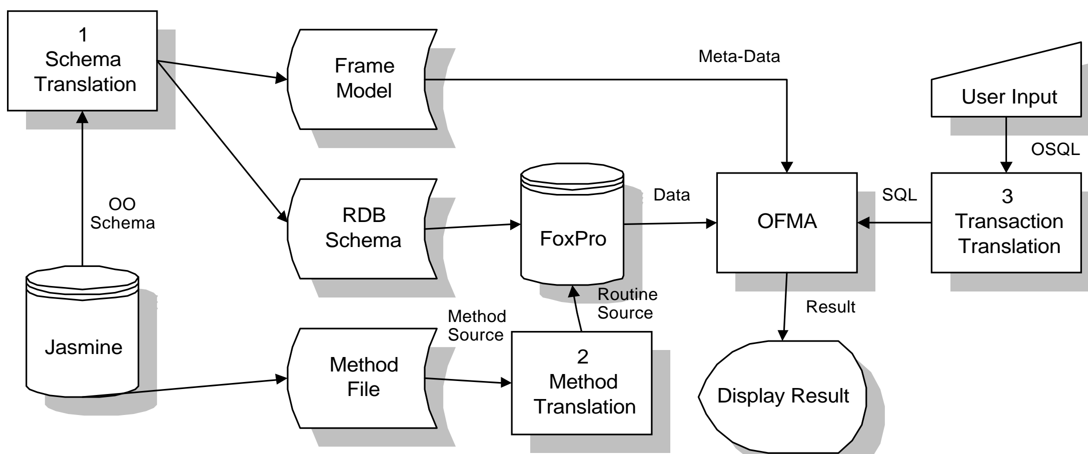

Figure 6 – Data Flow of the Prototype

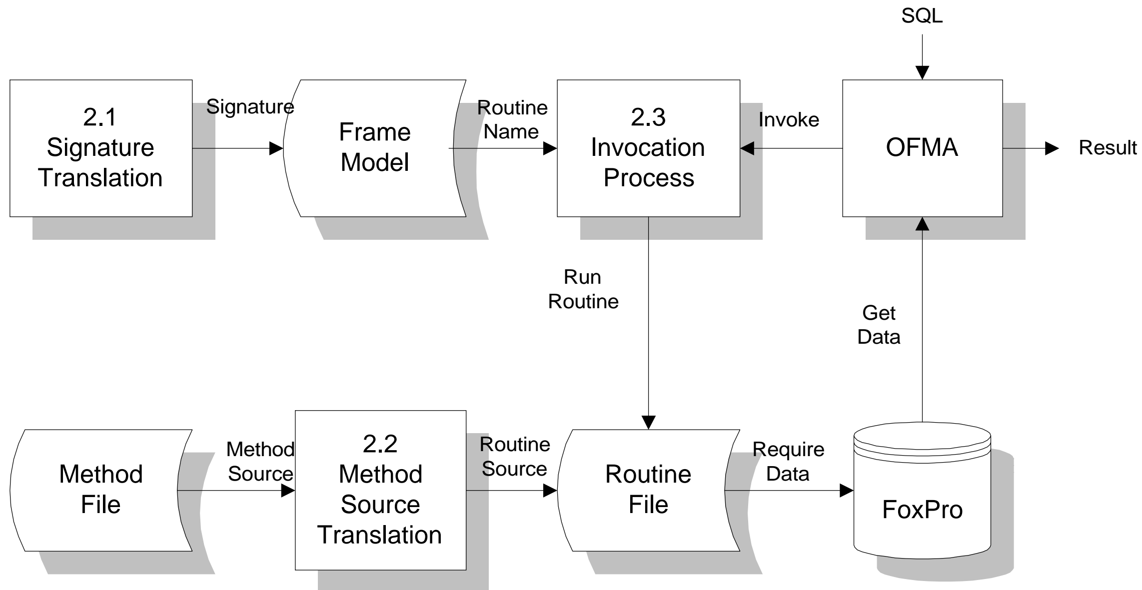

Figure 7 – Data Flow of the Method Translation

The OODBMS (Jasmine)[18] contains an OODB schema and a method file. In the first step, a frame model and RDB schema of the RDBMS (FoxPro)[19] are created through the process of the schema translation. On the other hand, the method file is read and converted to a routine file by the process of the source code translation. The source code of routines will be kept in the routine file. The last step is the invocation process. According to the signature, each call can be referred to the frame model and an action name can be located that is a routine name for executing the specified routine stored on the routine file. Each routine can be called by other routines within the routine file. The process of the signature is as follows.

Source Signature

```
defineClass staff
{       instance :
                String      firstname;
                String      surname;
                Void        findsurname (integer a);
                Void        findsurname (string name, integer a);
                Void        findsurname (string b);};
```

Frame Model (Target Signature)

Method_name	Class	Parameters	Method_type	Action
findsurname	staff	varchar	void	staff_findsurname1
findsurname	staff	varchar,integer	void	staff_findsurname2
findsurname	staff	integer	void	staff_findsurname3

During the schema translation, we also generate the following classes for working on the source code translation. The details are as follows :

Example

This example shows that in the "staff" table, the "staff.id_no" and "staff.name" are attributes copying from the superclass (i.e. "person" object). The "staff.dept.dept_name" is a composite attribute connecting to the "department" object. In the where-clause, the "staff.dept.dept_no" is a composite attribute as well.

Source Jasmine Method

```
void findsurname (integer a)
{       $string s <sval,sstat>;
        $staff p;
        $p = staff from staff where staff.dept.dept_no == a;
        $s = p.name;
        if (sstat == ODB_STATNIL)
          {       printf ("Surname is NIL"); }
        else
          {       printf ("Surname is %s", sval); }}
```

Translated Target FoxPro Routine

```
procedure staff_findsurname1
parameters a
use staff
use department
s = space(30)
sval = space(30)
sstat = space(1)
select * from staff, department;
where (staff.dept_oid = department.department_oid and department.dept_no = a);
into table p
s = p.name
IF s <> null THEN
    sval = s
    sstat = "Y"
ENDIF
Print_surname (sval, sstat)
close all
endproc

void print_surname (string sval, string sstat)&& C Language
 {      if (sstat == ODB_STATNIL)                    && Host Language
        {       printf ("Surname is NIL"); }
        else
        {       printf ("Surname is %s", sval); }}
```

As for the invocation, the action name on the frame is created by the system according to the class name and the method name. Since the routine model name is the same as the action name, a specified routine can be called by the frame from the routine file. The process of the invocation translation is as follows :

<u>**Source Invocation**</u>

```
$staff p                && Table "staff" is subclass of table "person
$p.findage (AP123)

switch (p)
{    case p = class person : person_findage
     case p = class staff  : staff_findsurname, person_findage }
```

<u>**Meta-data (Target Signature)**</u>

Method_name	Class	Parameters	Method_type	Action
findage	person	varchar	integer	person_findage
findsurname	staff	varchar	void	staff_findsurname

<u>**Translated Target Invocation**</u>

do person_findage with AP234

Transaction Translation

<u>**Single Class Query**</u>

The basic form consists of the three clauses, namely the SELECT clause, the FROM clause, and the optional WHERE clause. In the single class query, we SELECT one ore more attribute names or expressions involving attributes FROM a single class in the database. For example, 'find the department that staff '123' is working in'.

Input :
OSQL : SELECT dept.detp_name FROM staff WHERE staff_no = 123;

Result :
SQL : SELECT department.dept_name FROM staff, department WHERE staff.staff_no = '123' and staff.dept_oid = department.dept_oid;

<u>**Nested Insert Transaction**</u>

A nested INSERT statement is given in the value list at the position that corresponds to the attribute whose domain is a user-defined class. The class name as the domain of the attribute is the class_name that is referenced in the nested INSERT statement. If the attribute is a set of user-defined classes, then each element that is placed into the set must have an INSERT statement associated with it.

Input :
OSQL : INSERT INTO staff (staff_id_no, post, office, dept) VALUES (22344, 'Mick', 3333, (INSERT INTO department (dept_name, head) VALUES ('Compute','Ken'))

Result :
SQL : INSERT INTO staff (staff_id_no, post, office, dept_no) VALUES (22344, 'Mick', 3333, 809);
INSERT INTO department (dept_no, dept_name, head) VALUES (809, 'Compute','Ken')

Delete Transaction

The DELETE statement allows you to discard data that comprises an instance of a class.

Input :

OSQL : DELETE FROM student WHERE student_id_no = '10'

Result :

SQL : SELECT student_oid FROM student WHERE student_id_no = '10' into cursor oid ;
DELETE FROM grade WHERE student_oid = oid;

10. CONCLUSION AND FUTURE WORK

Object-Relational Database on a Relational Database Management System can be developed according to our methodology. Our future research will be directed towards creating an ORDBMS as we solve the conflicts between the Object-Oriented Database and Relational Database in this paper. During our research on this topic, we have found that other developers have their ideas regarding Object-Relational Database. They have applied their methodologies implementing the OO interfaces such as Informix's DataBlade, Software AG's Bolero, and Oracle's Data Cartridge. We use the frame model approach[20] to implement the OO interface because we do not have to change the existing system (i.e. Relational Model) and can easily modify the model for different situations. It can be formed as a meta-data during the schema translation and can be constructed as an application program interface for database interoperability. Also, it allows designers to combine the rules together into a class and to associate the classes in frame model as a knowledge-based system. We add the OO interface on top of the existing system. Users can easy apply OO concepts[21] in the relational system without interrupting their daily operations.

REFERENCES

1. M. Blaha et al., "Converting OO Models into RDBMS Schema", IEEE Software, May 1994, pp. 28-39.

2. J.A. Orenstein and D.N. Kamber, "Accessing a Relational Database through an Object-Oriented Database Interface, Proc. of VLDB 95, pp. 702-705.

3. S. DeFazio and J. Srinivasan, "Database Extensions for Complex Domains", Proc. of IEEE Data Engineering, 1996, pp. 200-202.

4. C. Yu et al., "Translation of Object-Oriented Queries to Relational Queries", IEEE Proc. of IEEE on Data Engineering, 1995, pp. 90-97.

5. J. Lim and D. Shin, "A Methodology of Constructing Canonical Form Database Schemas in a Multiple Heterogeneous Database Environment", Journal of Database Management, Vol.9 No.4, Fall 1998, pp. 4-11.

6. V. Krishnamurthy, S. Banerjee, and A. Nori, "Bringing Object-Relational Technology to the Mainstream", Proceedings of the 1999 ACM SIGMOD, pp513-514, 1999.

7. M. Stonebraker, P. Brown, and D. Moore, "Object-Relational DBMSs Tracking the Next Great Wave", Second Edition, Morgan Kaufmann Publishers Inc., 1999.

8. J. Fong and S. Huang, "Information Systems Reengineering", Springer Verlag, 1997.

9. O. Diaz and N.W. Paton, "Extending ODBMSs Using Metaclasses", IEEE Software, May 1993

10. R.G.G. Cattell, "The Object Database Standard : ODMG-93", Morgan Kaufmann Publishers, 1996.

11. H. Reichgelt, "Knowledge Representation", Ablex Publishing Corporation, 1991, pp. 143-176

12. R.G.G. Cattell, "The Object Database Standard : ODMG-2.0", Morgan Kaufmann Publishers, 1997.

13. C.J. Date & H. Darwen, "A Guide to the SQL Standard", Addison-Wesley, 4th edition, pp. 453-493.

14. J. Melton, "An SQL3 Snapshot", Proc. of IEEE Data Engineering, 1996, pp. 666-672.

15. J. Fong & P. Chitson, "Query Translation from SQL to OQL for Database Reengineering", International Journal of Information Technology, Vol. 3, No.1(1997), pp. 83-101

16. S. K. Cheung, "Adding an Object-Oriented Interface to Relational Database Using Frame Model", Proceedings of the 9th International Database Conference, 1999, pp.138-154, ISBN 962-937-046-8.

17. J. Fong and S. K. Cheung, "An Architectural Framework for Translating OODB Method to RDB Routine for ORDBMS", Proceedings of the 2nd ACM HKPRD Conference, 1999.

18. Computer Associates Int., Inc./Fujitsu Ltd., "Jasmine User's Manual", Release 1.1, 1997.

19. J.L. Hawkins, "FoxPro Programmer's Reference", QUE, 1996.

20. R. Fikes and T. Kehler, "The Role of Frame-based Representation in Reasoning", Communications of the ACM, Volume 28, No.9, 1985,pp. 904-920.

21. M. J. Carey, N. M. Mattos, and A. K. Nori, "Object-Relational Database Systems: Principles, Products and Challenges", Proceedings of the 1997 ACM SIGMOD, p502, 1997