Integrating Relational and Object-Oriented Databases in Multidatabase Systems

Ching-Ming Chao
Department of Computer and Information Science, Soochow University
chao@cis.scu.edu.tw

Abstract

A multidatabase system maintains a global schema that is the integration of component database schemas. This paper proposes an approach to database integration between relational and object-oriented databases in multidatabase systems. We first define correspondence assertions that specify semantic correspondences between schema objects of two component databases. The correspondence assertions are in the form of predicates in the first-order logic. We then present integration rules that describe how to construct the integrated schema according to the specified correspondence assertions. The integration rules consist of algorithmic steps and use primitive integration operators to restructure, transform, and integrate component databases. The primitive integration operators are algebraic operators. The primary objective of our approach is to facilitate the automation of database integration.

Keywords: Database Integration, Multidatabase Systems, Heterogeneous Databases, Distributed Databases.

1 Introduction

A *multidatabase system* (MDBS) is a database system that resides unobtrusively on top of existing *component database systems* (CDBSs) and presents a single database illusion to its users [2,8]. An MDBS usually maintains a single *global database schema*, which is the integration of all *component database schemas* and against which its users issue queries and updates. Note that the MDBS maintains only the global schema and the CDBS actually maintain all user data. Creating and maintaining the global schema, which requires the use of *database integration* technique, is a critical issue in multidatabase systems.

According to [1], the term *schema integration* can be used in two contexts: *view integration* in the process of database design and *database integration* in the process of integrating existing databases. A variety of approaches to schema integration have been proposed; e.g., [3,4,5,7]. Early approaches are mainly concerned with traditional and semantic data models. Today, as relational and object-oriented databases are the majority of databases, it is vital to have approaches to database integration for multidatabase systems with both relational and object-oriented component database systems. Database integration in such a multidatabase system includes integrating two relational databases, integrating a relational database and an object-oriented database, and integrating two object-oriented databases. Integration of relational databases and integration of object-oriented databases have been investigated a lot. However, research on integrating a relational database and an object-oriented database is rarely seen in the literature.

In this paper, we propose an approach to database integration between a relational database and an object-oriented database. We first define *correspondence assertions* that specify semantic correspondences between *schema objects* of two component databases. We then present *integration rules* that provide algorithmic steps for constructing an integrated schema from two component databases according to the specified correspondence assertions. These integration rules use *primitive integration operators* to restructure, transform, and integrate component databases.

Our approach is similar to that proposed in [7]. However, our approach is concerned with the integration between relational and object-oriented databases while theirs is concerned with the integration of object-oriented databases. In [6], Kim *et al.* provided a classification of schematic conflicts that may arise when integrating relational and object-oriented databases. They also gave a classification of conflict resolution techniques. Our approach extends their

classification of schematic conflicts and specifies the correspondence between component databases based on the *semantic domain* of schema objects. Besides, our approach uses primitive integration operators to construct the integrated schema while theirs uses a multidatabase language called SQL/M to define the integrated schema.

We adopt an object-oriented data model as the data model of the multidatabase system. The global schema is a *virtual schema* because no actual data are stored for this schema. Thus, the classes in the global schema are called *virtual classes* and the objects associated with the virtual classes are called *virtual objects*. Note that we assume an inheritance model in which the objects in a superclass are those that do not belong to its subclasses.

The remainder of this paper is organized as follows. Section 2 introduces the correspondence assertions between relational and object-oriented databases. Section 3 defines the primitive integration operators for restructuring, transforming, and integrating component databases. Section 4 presents the integration rules for constructing the integrated schema. Section 5 gives an example to illustrate the database integration process. Section 6 concludes this paper.

2 Correspondence Assertions

Correspondence assertions specify semantic correspondences between *schema objects* of two component databases. Schema objects in relational databases include tables and columns. Schema objects in object-oriented databases include classes and attributes. Correspondence assertions between a relational database and an object-oriented database are classified into the following four categories:

- Correspondences between tables and classes
- Correspondences between columns and attributes
- Correspondences between columns and classes
- Correspondences between tables and attributes

The first stage in our approach to database integration is to specify correspondence assertions as many as possible. However, there can be some correspondence assertions that cannot be specified until the stage of constructing the integrated schema. The correspondence assertions are in the form of predicates in the first-order logic. This facilitates automatic generation and human validation of assertions.

2.1 Table-versus-Class Correspondences

The correspondence between tables and classes is based on their semantic domains. The *semantic domain* of a table (or class) is the set of real world entities it can represent. Correspondence assertions in this category are further classified into one-to-one and many-to-one correspondences.

A *one-to-one correspondence* is one between a table and a class. Such a correspondence can be distinguished into *equivalent*, *related*, or *homonymous*. The related correspondence is further distinguished into *containment*, *overlap*, *disjoint*, or *component*.

- A table T is equivalent to a class C, denoted as *Entity-Equivalent (T, C)*, if their semantic domains are the same.
- A table T is contained in a class C, denoted as *Entity-Containment (T, C)*, if the semantic domain of T is a subset of that of C. Similarly, that a class C is contained in a table T is denoted as *Entity-Containment (C, T)*.
- A table T and a class C overlap, denoted as *Entity-Overlap (T, C)*, if the set-intersection of their semantic domains is not empty.
- A table T and a class C disjoint, denoted as *Entity-Disjoint (T, C)*, if the set-intersection of their semantic domains is empty but their semantic domains are both subsets of a common superset.
- A table T is a component of a class C, denoted as *Entity-Component (T, C)*, if a real world entity of T is a component of that of C. Similarly, that a real world entity of C is a component of that of T is denoted as *Entity-Component (C, T)*.
- A table T and a class C are homonymous, denoted as *Entity-Homonymous (T, C)*, if they are not equivalent or related but they have the same.

A *many-to-one correspondence* is one between a set of tables and a class. Due to normalization or other reasons,

several tables of a relational database together may represent the same set of real world entities as a class of an object-oriented database.

z A set of tables *TS* is equivalent to a class *C*, denoted as *Table-Set-Class-Equivalent (TS, C)*, if the semantic domain of *TS* is the same as that of *C*.

2.2 Column-versus-Attribute Correspondences

Correspondence assertions between columns and attributes are specified only when their respective owner table and class are equivalent or related. The correspondence between columns and attributes is based on their semantic domains. The *semantic domain* of a column (or attribute) is the set of real world entities it can represent. Correspondence assertions in this category are further classified into one-to-one and many-to-many correspondences.

A *one-to-one correspondence* is one between a column and an attribute. Such a correspondence can be distinguished into *equivalent*, *related*, or *homonymous*. The related correspondence is further distinguished into *containment* or *overlap*.

z A column *C* is equivalent to an attribute *A*, denoted as *Attribute-Equivalent (C, A)*, if their semantic domains are the same.

z A column *C* is contained in an attribute *A*, denoted as *Attribute-Containment (C, A)*, if the semantic domain of *C* is a subset of that of *A*. Conversely, that an attribute *A* is contained in a column *C* is denoted as *Attribute-Containment (A, C)*.

z A column *C* and an attribute *A* overlap, denoted as *Attribute-Overlap (C, A)*, if the set-intersection of their semantic domains is not empty.

z A column *C* and an attribute *A* are homonymous, denoted as *Attribute-Homonymous (C, A)*, if they are not equivalent or related but they have the same name.

A *many-to-many correspondence* is one between a set of columns and a set of attributes. Two component databases may use different numbers of columns and attributes, respectively, to represent the same set of real world entities.

z A column set *CS* is equivalent to an attribute set *AS*, denoted as *Attribute-Set-Equivalent (CS, AS)*, if the semantic domain of *CS* is the same as that of *AS*.

2.3 Column-versus-Class Correspondences

The same set of real world entities can be represented as one or more columns in a relational database and as a class in an object-oriented database.

z A set of columns *CS* of a table *T* is equivalent to a class *C*, denoted as *Column-Set-Class-Equivalent (T, CS, C)*, if the semantic domain of *CS* is the same as that of *C*.

2.4 Table-versus-Attribute Correspondences

The same set of real world entities can be represented as one or more attributes in an object-oriented database and as a table in a relational database.

z A set of attributes *AS* of a class *C* is equivalent to a table *T*, denoted as *Attribute-Set-Table-Equivalent (C, AS, T)*, if the semantic domain of *AS* is the same as that of *T*.

3 Primitive Integration Operators

Constructing an integrated schema is achieved by applying *primitive integration operators* to component schemas. It should be noted that component schemas do not change in the process of constructing the integrated schema. Primitive integration operators for integrating a relational database and an object-oriented database are classified into three categories: *restructuring*, *transforming*, and *integrating operators*. These operators are algebraic to facilitate the automation of constructing the integrated schema.

3.1 Restructuring Operators

Restructuring operators are used to rename or restructure schema objects of component databases to resolve conflicts between them.

z The *Rename* operator renames a table, column, class, or attribute. It uses the following syntax to rename a table or class: *Rename (entity, new-entity)* where *new-entity* is the new name for the table or class *entity*. It uses the following syntax to rename a column or attribute: *Rename (entity, old-attribute, new-attribute)* where *new-attribute* is the new name for *old-attribute* in *entity*.

z The *Coerce* operator changes the domain type of a column or attribute. It has the following syntax: *Coerce (entity, attribute, new-type)* where *new-type* is the new domain type for *attribute* in *entity*.

z The *Concatenate* operator concatenates several columns (or attributes) to a column (or attribute). It has the following syntax: *Concatenate (entity, {attribute-list}, new-attribute, new-type)* where columns (or attributes) in *attribute-list* of *entity* are replaced by a new column (or attribute) whose name is *new-attribute* and whose type is *new-type*. Types of concatenated columns (or attributes) and the resulted column (or attribute) must all be character strings.

z The *Upgrade-T* operator creates a table from a set of columns. It has the following syntax: *Upgrade-T (owner-table, {column-list}, new-column, new-table)* where columns in *column-list* belong to *owner-table*. Columns in *column-list* are replaced with a single column *new-column* that serves as a foreign key that references *new-table*. A new table *new-table* is created that includes columns in *column-list*. If *new-table* does not have a simple candidate key, *new-column* will be included as its primary key.

z The *Upgrade-C* operator creates a class from a set of attributes. It has the following syntax: *Upgrade-C (owner-class, {attribute-list}, new-attribute, new-class)* where attributes in *attribute-list* belong to *owner-class*. Attributes in *attribute-list* are replaced with a complex attribute *new-attribute* whose domain class is *new-class*. A new class *new-class* is created that includes attributes in *attribute-list*.

z The *Join* operator joins two tables into a table. It has the following syntax: *Join (table1, table2, new-table)* where *table1* and *table2* are the operands and *new-table* is the result. The semantics of the *Join* operator is similar to that of the *natural join* operator in the relational algebra.

3.2 Transforming Operators

Transforming operators are used to transform schema objects of relational databases into those of object-oriented databases.

z The *Transform* operator transforms a table into a class. It has the following syntax: *Transform (table)* where *table* is the table to be transformed. The table is transformed into a class while its columns are transformed into attributes. The resulted class and its attributes have the same names as those of *table* and its columns.

3.3 Integrating Operators

Integrating operators are used to construct schema objects of the integrated schema from those of component databases.

z The *Create* operator creates a virtual class in the integrated schema. It has the following syntax: *Create (class)* where *class* is a class from some component schema. The name, attributes, and virtual objects of the resulted virtual class are the same as those of *class*. Beside, the relationships (i.e., the inheritance and composition hierarchy) of the resulted virtual class remain the same.

z The *Combine* operator combines two classes into a virtual class. Only the resulted virtual class will appear in the integrated schema. It has the following syntax: *Combine (class1, class2, new-class)* where *class1* and *class2* are combined into the virtual class *new-class*. The *Combine* operator is similar to the *outer-join* operation in relational databases. The attributes of *new-class* are the set-union of those of *class1* and *class2*. The virtual objects of *new-class* are the set-union of those of *class1* and *class2*.

z The *Inherit* operator builds an inheritance hierarchy between two classes. It has the following syntax: *Inherit (subclass, superclass)*. Two virtual classes are produced in the integrated schema. One virtual class corresponds to *subclass* and is denoted as *virtual-subclass*. The other virtual class corresponds to *superclass* and is denoted as *virtual-superclass*. The attributes of *virtual-superclass* are the same as those of *superclass*. The attributes of *virtual-subclass* are the same as those of *subclass*; besides, it inherits the attributes of *superclass*. The virtual objects of *virtual-subclass* are the same as those of *subclass*. However,

if there are objects in *superclass*, which represent the same real world entities as some objects in *subclass*, these objects have to be virtually integrated in *virtual-subclass*. The virtual objects of *virtual-superclass* are the set-difference between *superclass* and *subclass*.

z The *Generalize* operator creates a common superclass of two classes. It has the following syntax: *Generalize (class1, class2, superclass)* where the virtual class *superclass* is the common superclass of *class1* and *class2*. The attributes of *superclass* are the set-intersection of those of *class1* and *class2*. The set of virtual objects of *superclass* is empty. Two more virtual classes are produced in the integrated schema as subclasses of *superclass*, whose attributes and virtual objects are the same as those of *class1* and *class2*, respectively.

z The *Specialize* operator creates a common subclass of two classes. It has the following syntax: *Specialize (class1, class2, subclass)* where the virtual class *subclass* is the common subclass of *class1* and *class2*. The attributes of *subclass* are the set-union of those of *class1* and *class2*. The virtual objects of *subclass* are the set-intersection of those of *class1* and *class2*. Two more virtual classes are produced in the integrated schema as superclasses of *subclass*. The attributes of these two virtual classes are the same as those of *class1* and *class2*, respectively. The virtual objects of each of these two virtual classes are the set-difference between each of the virtual objects of *class1* and *class2* and the virtual objects of *subclass*.

z The *Compose* operator builds a composition hierarchy between two classes. It has the following syntax: *Compose (component, composite, link-attribute)*. Two virtual classes *virtual-component* and *virtual-composite* are produced in the integrated schema such that *virtual-component* is the domain class of the attribute *link-attribute* in *virtual-composite*. The attributes and virtual objects of *virtual-component* and *virtual-composite* are the same as those of *component* and *composite*, respectively.

4 Integration Rules

According to the specified correspondence assertions between two component databases, *integration rules* describe how to construct the integrated schema. Each integration rule consists of algorithmic steps and invokes primitive integration operators. In this way, our integration rules facilitate the automation of database integration. We classify integration rules into the following four categories:

1. Integration rules for equivalent tables and classes
2. Integration rules for related tables and classes
3. Integration rules for column-set-class-equivalent and attribute-set-table-equivalent
4. Integration rules for other tables and classes

During the construction of the integrated schema, these four categories are applied in the following order: the third category, the first category, the second category, and the last category. For integration rules of the same category, they are applied to classes in an inheritance hierarchy in the top-down order. Besides, the same virtual class cannot be produced more than once in the integrated schema by different applications of integration rules.

4.1 First Category

Integration rule 1: This rule is applied when a correspondence assertion *Entity-Equivalent (T, C)* is specified.

[Step 1] If *T* and *C* have different names (i.e., they are synonymous), we apply the *Rename* operator to *T* or *C* to make them have the same name.

[Step 2] For each pair of column *CN* in *T* and attribute *A* in *C* such that a correspondence assertion *Attribute-Equivalent (CN, A)* is specified, we do the following two substeps.

[Step 2-1] If *CN* and *A* have different names, we apply the *Rename* operator to *CN* or *A* to make them have the same name.

[Step 2-2] If *CN* and *A* have different atomic types, we apply the *Coerce* operator to either *CN* or *A* to make them have the same type. If the type of *A* is a class, we apply the *Coerce* operator to *CN* to coerce its type to that of *A*.

[Step 3] For each pair of column *CN* in *T* and attribute *A* in

C such that an assertion *Attribute-Containment (CN, A)* is specified, we do the following two substeps.

[Step 3-1] If *CN* and *A* have different names, we apply the *Rename* operator to *CN* to change its name to that of *A*.

[Step 3-2] This substep is the same as step 2-2.

Similar process is performed for *Attribute-Containment (A, CN)* except the roles of *CN* and *A* are interchanged.

[Step 4] For each pair of column *CN* in *T* and attribute *A* in *C* such that a correspondence assertion *Attribute-Overlap (CN, A)* is specified, we do the following two substeps.

[Step 4-1] If *CN* and *A* have different names, we apply the *Rename* operator to both *CN* and *A* to make them have the same name that semantically contains the old names of *CN* and *A*.

[Step 4-2] This substep is the same as step 2-2.

[Step 5] For each pair of column *CN* in *T* and attribute *A* in *C* such that a correspondence assertion *Attribute-Homonymous (CN, A)* is specified, we apply the *Rename* operator to *CN* or *A* to make them have different names.

[Step 6] For each pair of column set *CS* in *T* and attribute set *AS* in *C* such that a correspondence assertion *Attribute-Set-Equivalent (CS, AS)* is specified, we apply the *Concatenate* operator to both *CS* and *AS* to make the resulted column and attribute have the same name and type.

[Step 7] Apply the *Transform* operator to *T* to transform it into a class.

[Step 8] Apply the *Combine* operator to *T* and *C* to produce a virtual class in the integrated schema.

Integration rule 2: This rule is applied when an assertion *Table-Set-Class-Equivalent (TS, C)* is specified.

[Step 1] Apply a sequence of *Join* operations to tables in *TS* to integrate them into a single table (called *TJ*).

[Step 2] Apply a process similar to that of steps 2 to 6 in integration rule 1 to resolve conflicts between the columns of *TJ* and the attributes of *C*.

[Step 3] Apply the *Transform* operator to *TJ* to transform it into a class.

[Step 4] Apply the *Combine* operator to *TJ* and *C* to produce a virtual class in the integrated schema.

4.2 Second Category

Integration rule 3: This rule is applied when a table *T* is related to a class *C*.

[Step 1] If *T* and *C* have the same name, apply the *Rename* operator to *T* or *C* to make them have different names.

[Step 2] Apply a process similar to that of steps 2 to 6 in integration rule 1 to resolve conflicts between the columns of *T* and the attributes of *C*.

[Step 3] Apply the *Transform* operator to *T* to transform it into a class.

[Step 4] This step is different for various correspondence assertions between *T* and *C*.

[Step 4-1] If an assertion *Entity-Containment (T, C)* is specified, we apply the *Inherit* operator to *T* and *C* to make *T* a subclass of *C*. Similarly, if *Entity-Containment (C, T)* is specified, we invoke the *Inherit (C, T)* operation.

[Step 4-2] If an assertion *Entity-Overlap (T, C)* is specified, we apply the *Generalize* and *Specialize* operators to *T* and *C* to create their common superclass and subclass.

[Step 4-3] If a correspondence assertion *Entity-Disjoint (T, C)* is specified, we apply the *Generalize* operator to *T* and *C* to create their common superclass.

[Step 4-4] If a correspondence assertion *Entity-Component (T, C, A)* is specified, we apply the *Compose* operator to *T* and *C* to build a composition hierarchy between them. Similarly, if *Entity-Component (C, T, CN)* is specified, we invoke the *Compose (C, T, CN)* operation.

4.3 Third Category

Integration rule 4: This rule is applied when an assertion *Column-Set-Class-Equivalent (CS, C)* is specified.

[Step 1] Apply the *Upgrade-T* operator to *CS* to create a new table (called *T*).

[Step 2] Specify the correspondence assertion *Entity-Equivalent (T, C)* and correspondence assertions between the columns of *T* and the attributes of *C*.

Integration rule 5: This rule is applied when an assertion

Attribute-Set-Table-Equivalent (AS, T) is specified.

[Step 1] Apply the *Upgrade-C* operator to *AS* to create a new class (called *C*).

[Step 2] Specify the correspondence assertion *Entity-Equivalent (T, C)* and correspondence assertions between the columns of *T* and the attributes of *C*.

4.4 Fourth Category

Integration rule 6: This rule is applied when a correspondence assertion *Entity-Homonymous (T, C)* is specified.

[Step 1] Apply the *Rename* operator to *T* or *C* to make them have different names.

[Step 2] Apply the *Transform* operator to *T* to transform it into a class.

[Step 3] Apply the *Create* operator to both *T* and *C* to produce two virtual classes in the integrated schema.

Integration rule 7: This rule is applied for each table and class without any correspondence assertion. For a table, we apply the *Transform* operator followed by the *Create* operator to it to produce a virtual class in the integrated schema. For a class, we apply the *Create* operator to it to produce a virtual class in the integrated schema.

5 An Integration Example

We now give a simple example to illustrate the process of database integration. Figure 1 shows schemas of two component databases *DB1* and *DB2*. *DB1* is a relational database and *DB2* an object-oriented database. First, correspondence assertions between schema objects of these two component databases are specified as follows.

- *Entity-Equivalent (Person@DB1, People@DB2)*
  - *Attribute-Equivalent (ss#, ssn)*
  - *Attribute-Equivalent (nationality, nationality)*
  - *Attribute-Set-Equivalent ({f-name, l-name}, name)*
- *Entity-Equivalent (Employee@DB1, Employee@DB2)*
  - *Attribute-Equivalent (salary, salary)*
- *Entity-Equivalent (Course@DB1, Course@DB2)*
  - *Attribute-Equivalent (c#, c#)*
  - *Attribute-Equivalent (credit, credit)*
- *Table-Set-Class-Equivalent ({Student, Enrollment}@DB1, Student@DB2)*
  - *Attribute-Equivalent (c#, courses)*
  - *Attribute-Containment (father, parent)*
- *Column-Set-Class-Equivalent (Person@DB1, {nationality}, Country@DB2)*

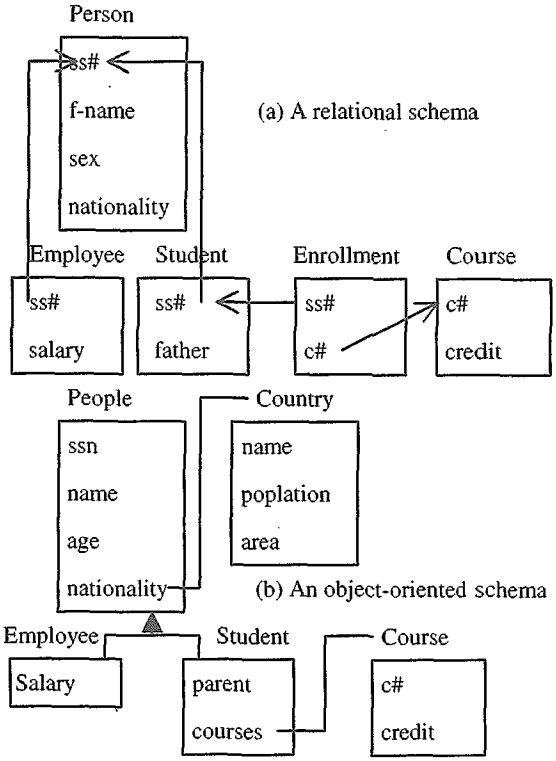

Figure 1 Schemas of Two Component Databases

Then, according to the specified correspondence assertions, integration rules are applied in the following order.

1. Apply integration rule 4 for *Column-Set-Class-Equivalent (Person@DB1, {nationality}, Country@DB2)*.
   - *Upgrade-T (Person@DB1, {nationality}, nationality, Country)*
   - Specify correspondence assertions *Entity-Equivalent (Country@DB1, Country@DB2)* and *Attribute-Equivalent (nationality, name)*.
2. Apply integration 1 for *Entity-Equivalent (Person@DB1, People@DB2)*.
   - *Rename (People@DB2, Person)*
   - *Rename (Person@DB2, ssn, ss#)*
   - *Coerce (Person@DB1, nationality, Country)*
   - *Concatenate (Person@DB1, {f-name, l-name}, name,*

*string-type**)

– *Transform (Person@DB1)*

– *Combine (Person@DB1, Person@DB2, Person)*

3. Apply integration rule 1 for *Entity-Equivalent (Employee@DB1, Employee@DB2).*

– *Transform (Employee@DB1)*

– *Combine(Employee@DB1,Employee@DB2,Employee)*

4. Apply integration rule 1 for *Entity-Equivalent (Course@DB1, Course@DB2).*

– *Transform (Course@DB1)*

– *Combine (Course@DB1, Course@DB2, Course)*

5. Apply integration rule 1 for *Entity-Equivalent (Country@DB1, Country@DB2).*

– *Rename (Country@DB1, nationality, name)*

– *Transform (Country@DB1)*

– *Combine (Country@DB1, Country@DB2, Country)*

6. Apply integration rule 2 for *Table-Set-Class-Equivalent ({Student, Enrollment}@DB1, Student@DB2).*

– *Join (Student@DB1, Enrollment@DB1, Student)*

– *Rename (Student@DB1, c#, courses)*

– *Coerce (Student@DB1, courses, Course)*

– *Rename (Student@DB1, father, parent)*

– *Transform (Student@DB1)*

– *Combine (Student@DB1, Student@DB2, Student)*

The integrated schema is shown in Figure 2.

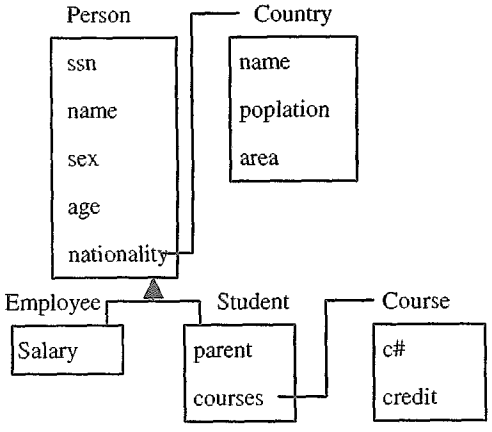

Figure 2 The Integrated Object-Oriented Schema

6 Conclusion

This paper proposes an approach to database integration between relational and object-oriented databases in multidatabase systems. In our approach, correspondence assertions between schema objects of component databases are specified first. Then integration rules are applied to construct the integrated schema. Our approach has three salient features that facilitate the automation of database integration. First, the correspondence assertions are in the form of predicates in the first-order logic. Second, the integration rules consist of algorithmic steps and use primitive integration operators. Last, the primitive integration operators are algebraic operators. In the future, we plan to address the issue of schema mapping in such multidatabase systems.

REFERENCES

[1] C. Batini, M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Surveys, Vol. 18, No. 4, pp. 323-364, December 1986.

[2] M.W. Bright, A.R. Hurson, and S.H. Pakzad, "A Taxonomy and Current Issues in Multidatabase Systems," IEEE Computer, Vol. 25, No. 3, pp. 50-60, March 1992.

[3] B. Czejdo and M. Taylor, "Integration of Database Systems Using an Object-Oriented Approach", in Proceedings of 1st International Workshop on Interoperability in Multidatabase Systems, April 1991, pp. 30-37.

[4] W. Gotthard, P.C. Lockemann, and A. Neufeld, "System-Guided View-Integration for Object-Oriented Databases," IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 1, pp. 1-22, February 1992.

[5] M. Kaul, K. Drosten, and E.J. Neuhold, "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views," in Proceedings of 6th International Conference on Data Engineering, 1990, pp. 2-10.

[6] W. Kim, I. Choi, S. Gala, and M. Scheevel, "On Resolving Schematic Heterogeneity in Multidatabase Systems," in Modern Database Systems: The Object Model, Interoperability, and Beyond, Addison-Wesley, 1995, pp. 521-550.

* *string-type* is the domain type of the attribute *name* in *Person@DB2*

[7] J.L. Koh and A.L.P. Chen, "Integration of Heterogeneous Object Schemas," in Entity-Relationship Approach, Springer-Verlag, 1994, pp. 297-314.

[8] E. Pitoura, O. Bukhres, and A. Elmagarmid, "Object-Oriented in Multidatabase Systems," ACM Computing Surveys, Vol. 27, No. 2, pp. 141-195, June 1995.