Integration of Heterogeneous Object Schemas*

Jia-Ling Koh and Arbee L.P. Chen

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email : alpchen@cs.nthu.edu.tw

Abstract. In a heterogeneous database system which consists of object databases, a global schema created by integrating schemas of the component databases can provide a uniform interface and high level location transparency for the users to retrieve data. The main problem for constructing a global schema is to resolve conflicts among component schemas. In this paper, we define *corresponding assertions* for the database administrators to specify the semantic correspondences among component object schemas. Based on these assertions, *integration rules* are designed, which use a set of primitive integration operators to restructure the component schemas for resolving the conflicts and do the integration. The principle of our integration strategy is to keep the data of component databases retrievable from the global schema without losing information. Moreover, more informative query answers may be derived from the heterogeneous databases due to schema integration. Finally, the mapping strategy between global schema and component schemas is provided.

1 Introduction

To provide a uniform interface and high level location transparency for the users to retrieve data in a heterogeneous database system, a global schema is usually created by integrating schemas of the component databases. A variety of approaches to schema integration have been proposed [1], [2], [7], [8], [10], [11], [14], [16]. Batini et al. discussed twelve methodologies for database or view integration [1]. Czejdo, et al. used a language with graphical user interface to perform schema integration in federated database systems [6]. Schema and domain incompatibilities were considered in [6], [9] and [15]. The issues on implementing schema integration tools were reported in [11], [12] and [19]. In [12], a knowledge based system was developed to support the view integration. On the other hand, an interactive tool to get the information required for the integration from a database administrator (DBA) and to integrate schemas according to the provided semantics was presented in [19]. In [11], for automating much of the integration process, the idea is to embed corresponding tools within the view integration process. As a similar approach to [19], the assertion-based approach was used in [17] and [20]. In [20], this approach developed integration rules for a variety of data models. A generic description of schema correspondences among different data models

* this work was partially supported by the Republic of China National Science Council under Contract No. NSC 83-0408-E-007-030.

was provided. Other approaches defined a set of operators to build a virtual integration of multiple databases or to customize virtual classes [18], [23], [24]. Yet another approach asserted that the different constructs of component schemas be standardized before the integration. Several transformation rules were then proposed for the view integration process [13].

In this paper, we present a schema integration mechanism to achieve a global object schema for existing object databases. The main problem is to resolve different conflicts among component object schemas, such as attribute conflicts, class hierarchy conflicts, etc. We first define *corresponding assertions* for the DBA to specify the semantic correspondences among component schemas. Based on these assertions, *integration rules* are designed, which use a set of primitive integration operators to do the integration. The integration operators are used to restructure or integrate the component schemas, and the rules specify what integration operators should be applied in what order in different situations. The principle of our integration strategy is to keep the data of component databases retrievable from the global schema without losing information. Besides, we may derive more informative query answers from the heterogeneous databases due to schema integration. In the process of schema integration, we use *mapping tables* to store the mapping information between the restructured or integrated virtual schemas and the component schemas. Finally, when the schema integration process is finished, the mappings between the global schema and the component schemas are recorded in a data dictionary and directory (DD/D). The mapping information will be used for the decomposition of the queries against the global schema.

Our approach is similar to the assertion-based approach in [20]. However, we consider only object schemas; there is no need to transform the component schemas to ones in certain generic data model. Besides, we consider the construction of class hierarchies in the integrated schema, which was not discussed in [20]. In the process of schema integration, we use integration operators to restructure and integrate the component schemas. Different from the operators provided in [18] and [24], our *class restructuring operators* can be used to restructure the attributes and class hierarchies of a class. Thus, the conflicts in component schemas can be resolved before the integration. The *class integration operators* can be used to integrate the information in two classes or a set of attributes and a class. In addition, inheritance models of objects in the class hierarchy are considered in our discussion. Furthermore, our integration operators contrast with the integration and transformation primitives in [11]. In the definition of the integration and transformation primitives, only the conditions and the result when they are applied are specified. No explicit primitive operators are provided for the process of schema integration.

The paper is organized as follows. The next section clarifies some basic concepts used throughout this paper. Then Section 3 introduces the corresponding assertions between component schemas. Section 4 defines the primitive integration operators for integrating component schemas. Integration rules are discussed in Section 5. Finally, Section 6 concludes this paper with a discussion on the future work.

2 Basic Concepts

Inheritance model In a class hierarchy, classes are linked according to the class hierarchy (IS-A) relationship among them. There are two kinds of object inheritance models in a class hierarchy. One model describes that the objects in the subclasses are also in their superclass. Thus, when a query accesses a class, all objects in the

class hierarchy rooted at the class will be accessed. The other model defines that the objects in a class are those objects which do not belong in its subclasses. When a query accesses a class, only the objects in the class are accessed unless some special notation on the class in the query is specified; in which case, all classes in the class hierarchy rooted at the class will be accessed. In this paper, we will adopt the latter inheritance model.

Concept of virtual global schema As the concept in [18], the global object schema is virtual. That is, no actual data are stored for this schema. A mapping is available from this global schema to the schemas in component databases. Thus, the classes in the global schema are called *virtual classes*. Besides, the objects associated with the virtual classes are called *virtual objects*.

Concept of object polymorphism In a heterogeneous database system, a real-world entity may exist in different databases as different objects. In the process of schema integration, we integrate the objects representing the same entity among component databases into a single virtual object in the global schema. We have discussed a strategy in [4] to find such objects, named *polymorphic objects*, in heterogeneous databases. In this paper, we assume that the objects in different component databases denoting the same entity have been determined. Thus, the statement describing *set-operations* (*set-union*, *set-intersection*, and *set-difference*) among two sets of objects are actually the *set-operations* among the sets of the corresponding real-world entities.

3 Corresponding Assertions

In order to provide the information for schema integration, the DBAs have to specify the corresponding assertions between schemas in different component databases. In addition to specifying the corresponding assertions, the DBAs have to specify the *division characteristics* for the classes which have subclasses. The *division characteristics* of a class are properties which can denote the differences among the subclasses of the class. The information is useful in the processing of schema integration. For example, class **Person** has subclasses **Man** and **Woman**. Then, *sex* is the *division characteristic* of **Person**.

There are four kinds of corresponding assertions. The first kind describes the correspondence of classes. The second one describes the correspondence of attributes. The third kind characterizes the semantic relationships of classes and attributes. Finally, the last kind of assertions formulates the semantic equivalence of composition hierarchies. Details of each assertion are described as follows.

3.1 Corresponding assertions among classes

The correspondence among classes is based on *semantic domains* of the classes. The *semantic domain* of a class is the set of real world entities that the class can represent. According to the relationship among the *semantic domains* of the classes, the corresponding assertions among classes are specified. The classes which have corresponding assertions with other classes will be integrated in the process of schema integration because of their related semantics.

- *Class-Equivalent*
  Classes **X1** and **X2** are *class-equivalent* means that classes **X1** and **X2** are semantically equivalent. In other words, the semantic domains of **X1** and **X2** are

the same. There are two kinds of *class-equivalent*: *implicit class-equivalent* and *explicit class-equivalent*. If the identities of the databases where **X1** and **X2** come from need to be kept for the virtual objects in the integrated virtual class, the two classes are *explicit class-equivalent*. Otherwise, they are *implicit class-equivalent*.
For example, assume database1 and database2 store information for differ-ent schools. Class **Student** of database1 and class **Student** of database2 are *explicit class-equivalent* because after the integration, the school information for the virtual objects in the integrated class may be needed to identify which school a student comes from.

- *Class-Correspondent*
  Classes **X1** and **X2** are *class-correspondent* if classes **X1** and **X2** are semantically related but not equivalent. Based on the relationship between *semantic domains* of two classes, three kinds of class correspondences are identified: *class_containment*, *class_overlap* and *class_disjointness*.
  For example, class **Woman** in database1 and class **Person** in database2 have *class_containment- correspondence*. The reason is that, according to the *semantic domains* of the classes, **Woman** is contained in **Person**. The objects which can be represented in class **Student** and class **Teacher** may be overlapping. Thus, they have *class_overlap-correspondence*. Finally, for the classes **Woman** and **Man** in different databases, they have *class_disjointness-correspondence*.

When specifying the corresponding assertions among classes, *class-equivalent* assertions are specified first. Then *class-correspondence* assertions are considered.

3.2 Corresponding assertions among attributes

Only when classes are specified as *class-equivalent* or *class-correspondent* can the DBA specify the corresponding assertions among attributes in these classes. This is because the class to which an attribute belong will affect the actual meaning of the attribute. According to the corresponding assertions among attributes, the conflicts of attributes can be identified.

- *Attribute-Equivalent*
  Two primitive attributes **A1** and **A2** are *attribute-equivalent* if they are semantically equivalent. For example, attribute **Student**. *name* in database1 and attribute **Student**. *s-name* in database2 are *attribute-equivalent*. They both denote the name of students.
- *Attribute_Set-Equivalent*
  Attribute sets **S1** = { a_1, a_2, ..., a_n } and **S2** = { a'_1, a'_2, ..., a'_m } are *attribute_set-equivalent* if they are semantically equivalent. For example, attribute set { *city, street, no* } of class **Person** in database1 and attribute set { *address* } of **Person** in database2 are *attribute_set-equivalent*. They denote the same meaning for address.

3.3 Corresponding assertions among classes and attributes

In the object data model, it is flexible to use classes or attributes to represent the same kind of information. Thus, the DBA has to specify the corresponding assertions among classes and attributes.

- *Attribute_Set - Class-Equivalent*
  Attribute set **S** = { a_1, a_2, ..., a_n } and class **X** are *attribute_set - class-equivalent*

if attribute set **S** and class **X** are semantically equivalent. For example, attribute set { *blood-type* } of class **Person** in database1 and class **Blood** in database2 are *attribute_set - class-equivalent*. Both of them denote the information about blood.

- *Attribute - Class_Set-Equivalent*
  If attribute **A** is semantically equivalent to a *division characteristic* of class **C**, then **A** and the associated subclasses of **C** are *attribute - class_set equivalent*. By this information, **A** can be used to derive a set of subclasses for its associated class. Each subclass will be *class-equivalent* to a subclass of **C**. For example, attribute **Person**.*sex* in database1 and the subclasses { **Man**, **Woman** } of class **Person** in database2 are *attribute - class_set-equivalent*. This is because the *division characteristic* of class **Person** in database2 for the subclasses is *sex*.

3.4 Corresponding assertions among composition hierarchies

In addition to the corresponding assertions among classes and attributes, the semantics in the composition hierarchies should be also considered.

- *Composition_Hierarchy-Equivalent*
  Let **P1** be a path from class **C1** to class **C1** in the composition hierarchy of database1, and **P2** be a path from class **C2** to class **C2** in the composition hierarchy of database2. Paths **P1** and **P2** are *composition_hierarchy-equivalent* if both **C1** and **C2**, and **C2** and **C1** are *class-equivalent*. Besides, paths **P1** and **P2** are semantically equivalent. For example, both **Person**.*car* in database1 and **Car**.*owner* in database2 denote the ownership of a car. Thus, the two paths are *composition_hierarchy-equivalent*.

4 Primitive Integration Operators

We define a set of operators for integrating object schemas. These operators can be categorized into *class restructuring* and *class integration operators*. *Class restructuring operators* are used to restructure the classes in component schemas to resolve their conflicts. After the restructuring process, *class integration operators* are then used to integrate classes. The class hierarchies and class composition hierarchies among classes in different component schemas will also be built by the *class integration operators*.

4.1 Class restructuring operators

Class restructuring operators may change the structure of attributes in a class and the structure of a class hierarchy. We provide seven *class restructuring operators* as follows. Note that the first five *class restructuring operators* are used to restructure the attributes of a class. Also, all subclasses rooted at the class inherit the restructured attributes.

1. *Refine*
   The *Refine* operator adds an attribute to a class. Besides, the added attribute is assigned a constant value for saving certain semantic information. It has the following syntax: $Refine(source\text{-}class, new\text{-}attribute, constant\text{-}value)$; with *constant-value* being the value of the newly defined attribute *new-attribute* to the *source-class*.

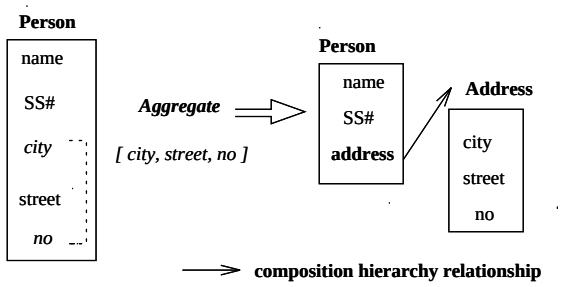


Figure 1: example for *Aggregate*

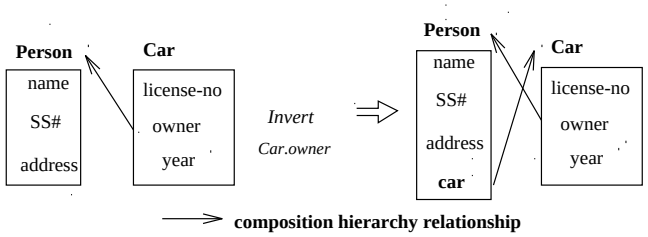


Figure 2: example for *Invert*

2. *Hide*
   The *Hide* operator removes an existing attribute from a class. It has the following syntax: $Hide(source\text{-}class, hidden\text{-}attribute)$; with *hidden-attribute* being an existing attribute in the *source-class*.
3. *Rename*
   The *Rename* operator renames a class name or attribute name by a new name. It has the following syntax: $Rename\ (source\text{-}class\ (\ .source\text{-}attribute), new\text{-}class\ (.attribute\text{-}name))$; with $new\text{-}class(attribute)\text{-}name$ being the new name for *source-class* or *source-attribute* in *source-class*.
4. *Aggregate*
   The *Aggregate* operator aggregates a set of primitive attributes of a class into a complex attribute. It has the following syntax: $Aggregate(source\text{-}class, [attribute\text{-}list], new\text{-}complex\text{-}attribute, new\text{-}domain\text{-}class)$; with the *attribute-list* in *source-class* being aggregated into the *new-complex-attribute*. Besides, a new virtual class *new-domain-class* is created to be the domain class of the *new-complex-attribute*. The attributes of *new-domain-class* are exactly the aggregated set of primitive attributes. The virtual objects in *new-domain-class* are the *projection* of *source-class* and the subclasses rooted at *source-class* on the set of attributes in *attribute-list*.
   For example, as Figure 1 shows, attribute set { *city, street, no* } of class **Person** is *Aggregated* to a complex attribute *address*. Besides, a new virtual class *Address* is created as the domain class of *address*. The operation is denoted as: $Aggregate(\mathbf{Person}, [city, street, no], address, \mathbf{Address})$.

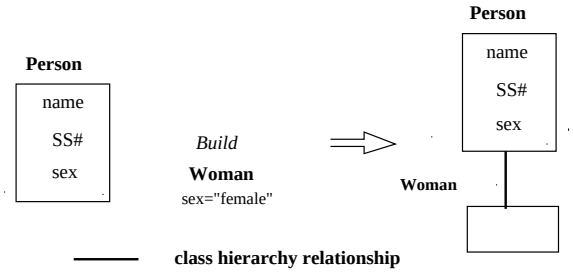


Figure 3: example for *Build*

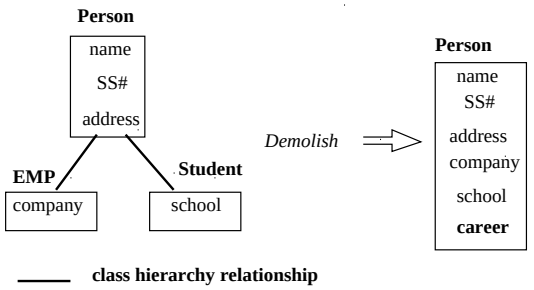


Figure 4: example for *Demolish*

5. *Invert*
   The *Invert* operator defines a new complex attribute which is the inverse of a complex attribute in another class. It has the following syntax: $Invert(source\text{-}class, inverted\text{-}attribute, new\text{-}complex\text{-}attribute)$; with *new-complex-attribute* being the inverse for the *inverted-attribute*. Note that the *source-class* is the domain class of the *inverted attribute*. After the operation, the domain class of the *new-complex-attribute* is the class in which the *inverted-attribute* belongs.
   For example, as Figure 2 shows, **Person** is the domain class of **Car**.*owner*. The *Invert* on the **Car**.*owner* will create a new complex attribute **Person**.*car* in **Person** with class **Car** as its domain class. The operation is denoted as: *Invert*(**Person**, **Car**.*owner*, *car*).
6. *Build*
   The *Build* operator creates a new virtual class containing virtual objects satisfying a predicate clause from a given class. It has the following syntax: $Build(source\text{-}class, new\text{-}class, [predicateclause])$; with the *predicate clause* being a simple predicate clause on an attribute in *source-class*. The attributes of the *new-class* are the same as the attributes of the *source-class*. In addition, the *source-class* becomes a virtual class, too. It contains the virtual objects which are the *set-difference* of the objects in the original *source-class* and the *new-class*. In the class hierarchy, it is the superclass of the *new-class* in the global schema.
   For example, the schemas before and after the operation *Build* (**Person**, **Woman**, [*sex*=female]) are shown in Figure 3.

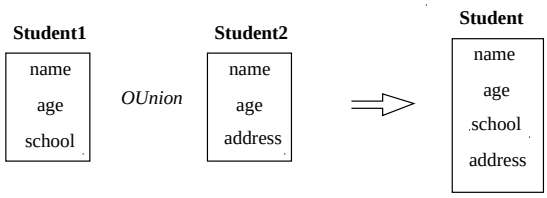


Figure 5: example for *OUnion*

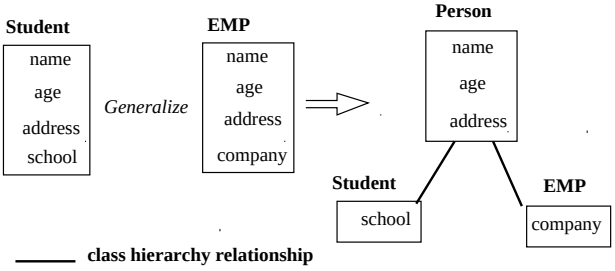


Figure 6: example for *Generalize*

7. *Demolish*
   The *Demolish* operator demolishes the subclasses of some class to a set of attributes in the new virtual class. It has the following syntax: $Demolish(source\text{-}class)$. The attributes of the new class are the *set-union* of the attributes in the *source-class* and its subclasses. Besides, an attribute which denotes the division characteristic of the class is added. This attribute will be set a constant value for the objects in each subclass. The virtual objects of the new class are also the *set-union* of the objects in the *source-class* and its subclasses. Note that the *Demolish* operator is recursive. That is, the subclasses will be *Demolished* if they also have subclasses. For example, the schemas before and after the operation $Demolish(\mathbf{Person})$ are shown in Figure 4.

4.2 Class integration operators

Class integration operators are used to integrate classes from different component databases. Some operators are used to build the classes of global schema by integrating two classes. The other operators can be used to build more complete class hierarchies and class composition hierarchies in the global schema. Five *class integration operators* are introduced in the following.

1. *OUnion*
   The *OUnion* operator is the *set-union* of objects in two equivalent classes. It has the following syntax: $OUnion(source\text{-}class\mathbf{1}, source\text{-}class\mathbf{2}, new\text{-}class)$; with *new-class* being created to be the result. Only *new-class* will appear in the global schema. The attributes of the new virtual class are the *set-union* of the attributes of the two operands. The virtual objects of the *new-class* are the *set-union* of the objects in *source-class1* and *source-class2*. Note that we do not consider the

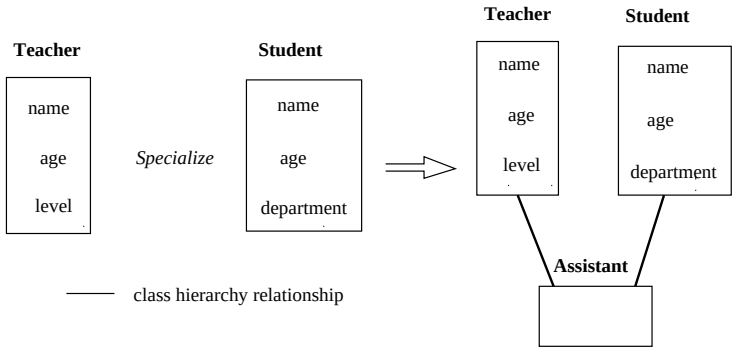


Figure 7: example for *Specialize*

situations of value conflicts as in [21] and [22]. If there are some objects in the *source-class1* and *source-class2* denoting the same real world entities, the values of the attributes in both classes will be integrated for the virtual objects in the new class. However, for other virtual objects, null values will be filled in those originally absent attributes. *OUnion* is similar to the *outerjoin* operation in relational database systems [3].

For example, as shown in Figure 5, we integrate class **Student1** of database1 and class **Student2** of database2 using *OUnion*. Class **Student** is the result of *OUnion*. The operation is denoted as: *OUnion* (**Student1**, **Student2**, **Student**).

2. *Generalize*

The *Generalize* operator creates a common superclass of two classes. It has the following syntax: $Generalize(source\text{-}class\mathbf{1}, source\text{-}class\mathbf{2}, common\text{-}superclass)$; with the *common-superclass* being the common superclass. The attributes of the new virtual class – *common-superclass* are the *set-intersection* of the attributes in the operand classes. Besides, the set of virtual objects in the *common-superclass* is empty. The two operand classes will become two virtual subclasses under the *common-superclass*.

We show the *Generalize* of class **Student** of database1 and class **EMP** of database2 in Figure 6. A new class **Person** will be created. The operation is denoted as: *Generalize*(**Student**, **EMP**, **Person**).

3. *Specialize*

The *Specialize* operator creates the common subclass of two classes. It has the following syntax : $Specialize(source\text{-}class\mathbf{1}, source\text{-}class\mathbf{2}, common\text{-}subclass)$; with the *common-subclass* being the common subclass. The attributes of the new virtual class – *common-subclass* are the *set-union* of the attributes in the two operand classes. Moreover, the *common-subclass* contains the virtual objects which are the *set-intersection* of the objects in the two operands. Then there will be two virtual classes produced as the superclasses of the *common-subclass* in the global schema. The attributes in the two virtual superclasses correspond to the attributes of *source-class1* and *source-class2*, respectively. However, the superclasses each contains the *set-difference* of the objects in a source class and the *common-subclass*.

As Figure 7 shows, class **Assistant** is the *Specialize* of classes **Student** and **Teacher**. The operation is denoted as: *Specialize*(**Student**, **Teacher**, **Assistant**).

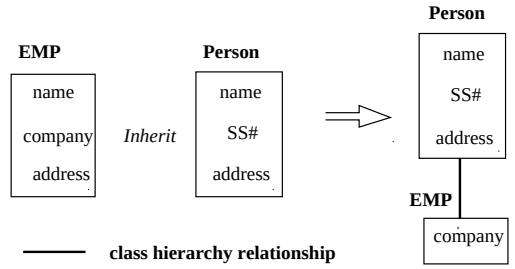


Figure 8: example for *Inherit*

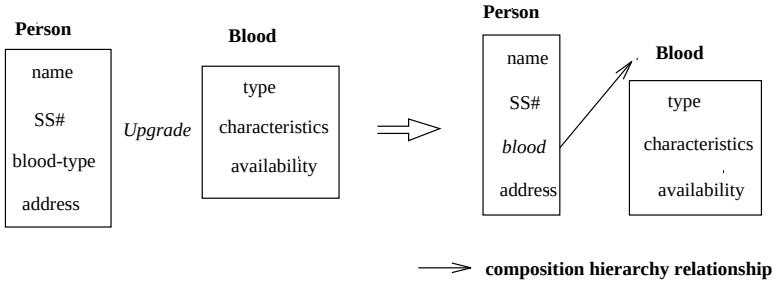


Figure 9: example for *Upgrade*

4. *Inherit*
   The *Inherit* operator builds the class hierarchy relationship of two classes. It has the following syntax: $Inherit(source\text{-}subclass, source\text{-}superclass)$; with *source-superclass* being built as the superclass of *source-subclass*. Two virtual classes are produced in the global schema. One class corresponds to the *source-superclass* and is denoted as *virtual-superclass*. The attributes in the *virtual-superclass* are the same as those in *source-superclass*. The other class corresponds to the *source-subclass* and is denoted as *virtual-subclass*. The attributes in *virtual-subclass* are the same as the *source-subclass*. Besides, it will inherit the attributes of the *source-superclass*. In *virtual-subclass*, the virtual objects are the same as those in the *source-subclass*. However, if there are objects in the *source-superclass*, which represent the same real world entities as some objects in the *source-subclass*, these objects have to be virtually integrated in *virtual-subclass*. For the other virtual objects in *virtual-subclass*, the inherited attributes will be filled with null values. On the other hand, the virtual objects in *virtual-superclass* will be the *set-difference* of the original *source-superclass* and *source-subclass*.
   An example which shows the *Inherit* of class **EMP** with class **Person** appears in Figure 8. The operation is denoted as: *Inherit*(**EMP**, **Person**).
5. *Upgrade*
   The *Upgrade* operator upgrades a set of attributes in a class to a complex attribute. The domain class has to be an existing class in another component database. It has the following syntax: $Upgrade(source\text{-}class, [attribute\text{-}list], new\text{-}complex\text{-}attribute, domain\text{-}class)$; with the *attribute-list* in *source-class* being upgraded to the *new-complex-attribute*. Besides, the domain class of the *new-complex-attribute* is an existing class – *domain-class*. After the *Upgrade* process, the *source-class* is

modified to a virtual class with the same attributes except the *attribute-list* being replaced by *new-complex-attribute*. The virtual objects in the new class are the same as the *source-class*, which can get more information from the *domain-class*. Note that all subclasses rooted at *source-class* will inherit the effect of *Upgrade*. Figure 9 shows that attribute *blood-type* in class **Person** is *Upgraded* to a complex attribute with class **Blood** as its domain class. Additional information for a person about blood can be obtained from the new domain class. The operation is denoted as: *Upgrade*(**Person**, [*blood-type*], *blood*, **Blood**).

5 Integration Rules

Integration rules are the major part that guides the integrator to do the actual schema integration. According to the specified corresponding assertions, there are three kinds of situations where the classes have to be integrated or certain relationships built between them. First, the classes which are *class-equivalent* have to be integrated into a single class. Next, when the classes are *class-correspondent*, some structures of class hierarchies have to be built. Finally, if a set of attributes and a class have the *attribute_set - class-equivalent*, the set of attributes will be changed to a complex attribute and linked to the semantically equivalent class. In the following, details of the three situations will be discussed, followed by an overall application of the integration rules to an example.

5.1 Integration rules for class-equivalent classes

When classes **X1** and **X2** are *class-equivalent*, they can be integrated into a single virtual class. In the integration process, the first four steps resolve some possible conflicts among these two classes, and the last step builds the integrated virtual class. These five steps are repeated for all class-equivalent assertions between the component schemas. The integration rules are applied to classes in the top-down order of the class hierarchies.

Step 1 is used to resolve the conflicts of class hierarchies among classes **X1** and **X2**.

[step 1] Consider when **X1** or **X2** has subclasses.

[step 1-1] If there exists one-to-one *class-equivalent* correspondence between direct subclasses of **X1** and **X2**, we do nothing at **step 1**. Else, there are class hierarchy conflicts between **X1** and **X2** and we continue to **step 1-2**. We call those subclasses which do not have corresponding *class-equivalent* classes in the other schema *unmatched subclasses*.

[step 1-2] If there exist direct subclasses for both **X1** and **X2** and the *division characteristics* of the two classes are not semantically equivalent, the DBA selects one class to perform *Demolish* operator.

[step 1-3] Consider if the *attribute - class_set-equivalent* assertion exists between an attribute in a class and the direct subclasses of the other class. If the *attribute - class_set-equivalent* assertion exists, we use *Build* operator on the class which contains the attribute. The attribute is used as the predicate attribute.

Step 2 is used to resolve the conflicts of attribute structures among classes **X1** and **X2**.

[step 2] Consider when there are *attribute_set-equivalent* assertions between **X1** and

X2.

[step 2-1] If each attribute set contains a single complex attribute, we do nothing in **step 2**. Else, **step 2-2** continues.

[step 2-2] If the assertion is between a complex attribute and a set of primitive attributes, the set of primitive attributes are *Aggregated*. The aggregated new class and the domain class of the complex attribute are specified to be *class-equivalent*. Otherwise, both sets of primitive attributes are *Aggregated*, and the aggregated new classes are specified to be *class-equivalent*.

Step 3 is used to resolve the conflicts of composition hierarchies between classes **X1** and **X2**.

[step3] Consider when there are *composition_hierarchy-equivalent* assertions between the composition hierarchies of **X1** and **X2**. In this situation, the complex attributes which form the equivalent composition hierarchies should be *Inverted*.

[step 4] Consider when **X1** and **X2** are *explicit class-equivalent*. In this situation, both **X1** and **X2** have to be *Refined*. The values of the new attributes specify which database an object comes from after the integration. Thus, both attributes are assigned a constant value.

[step 5] Processes the integration of **X1** and **X2**. We use *OUnion* operator to integrate these two classes. The virtual objects in the integrated class may contain more information if they denote the same real-world entities.

5.2 Integration rules for class-correspondent classes

When classes **X1** and **X2** are *class-correspondent*, we have to build a class hierarchy for these two classes. Some conflicts in the structures of the attributes have to be resolved too.

Step 1 is used to resolve the conflicts of attribute structures among classes **X1** and **X2**.

[step 1] Consider when there are *attribute_set-equivalent* assertions between **X1** and **X2**.
This step is the same as **step 2** in Section 5.1.

Step 2 builds a class hierarchy structure for classes **X1** and **X2**.

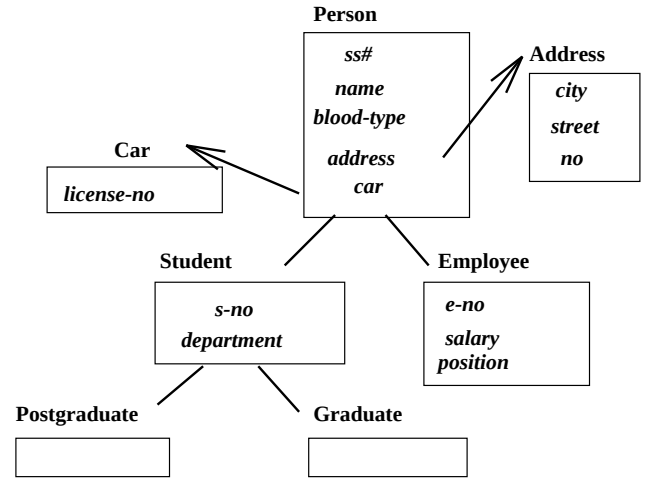
[step 2] Consider the different kinds of *class-correspondent* assertions.

[step 2-1] If **X1** and **X2** are *class_containment-correspondence*, *Inherit* operator is used to build a class hierarchy relationship between them.

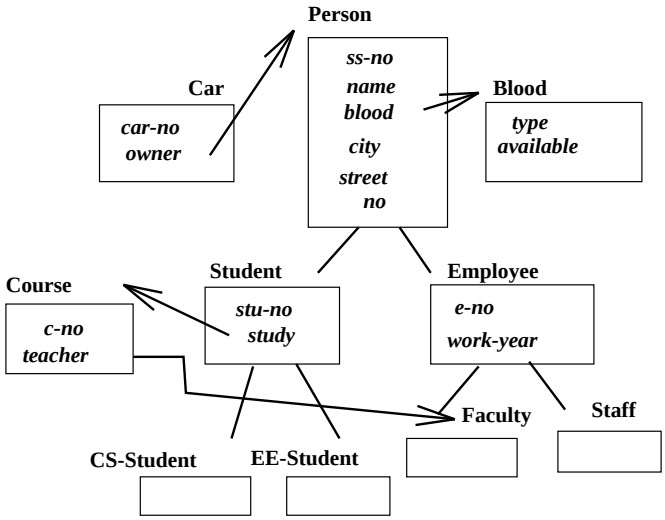
[step 2-2] If **X1** and **X2** are *class_overlap-correspondence*, *Generalize* and *Specialize* operators are used to build the common superclass and subclass of both classes, respectively.

[step 2-3] Otherwise, **X1** and **X2** are *class_disjointness-correspondence*. We use *Generalize* operator to build the common superclass of these two classes.

These two steps are repeated for all the specified *class-correspondent* classes to build the structure of class hierarchies in the global schema.



DB1



DB2

Figure 10: Component schemas in database1 and database2

5.3 Integration rules for attribute_set - class-equivalent

When attribute set $\mathbf{S} = \{ a_1, a_2, ..., a_n \}$ and class $\mathbf{X}$ are *attribute_set - class-equivalent*, we try to upgrade the attribute set $\mathbf{S}$ to get more information from $\mathbf{X}$. However, the purpose can be satisfied only in some situations. When each set of values for $\mathbf{S}$ corresponds to an object in $\mathbf{X}$, the attribute set $\mathbf{S}$ can be *Upgraded* to a complex attribute and linked to class $\mathbf{X}$. Otherwise, the set of attributes in $\mathbf{S}$ have to be *Aggregated*. The aggregated new class and class $\mathbf{X}$ are specified to be *class-equivalent*.

5.4 The overall application of different integration rules

When applying the different integration rules to integrate component schemas, the following order is followed:

- Integration rules for *attribute_set - class_equivalent*
- Integration rules for *class-equivalent*
- Integration rules for *class-correspondent*
- Integration rules for the classes without corresponding assertions
  The classes in component schemas without corresponding assertions are put in the global schema as virtual classes. The attributes and virtual objects in the virtual classes are the same as those of the original classes. Besides, the relationships (class hierarchies or composition hierarchies) of these classes with other classes remain the same.

Now, we consider an example to describe the overall processing of schema integration. Figure 10 shows the schemas of two databases: database1 and database2. They are databases in two different schools, used to store the personal information.

First, the corresponding assertions among these two schemas are specified. The *class-equivalent* classes and their *attribute-equivalent* or *attribute_set-equivalent* attributes are specified in the following tables.

class-equivalent (explicit)	**Person@DB1**	**Person@DB2**
division characteristic	career	career
attribute-equivalent	ss#	ss-no
	name	name
attribute_set -equivalent	{blood-type}	{blood}
	{address}	{city, street, no}

class-equivalent (implicit)	**Car@DB1**	**Car@DB2**
attribute-equivalent	license-no	car-no

class-equivalent (explicit)	**Student@DB1**	**Student@DB2**
division characteristic	degree	department
attribute-equivalent	s-no	stu-no

class-equivalent(explicit)	**Employee@DB1**	**Employee@DB2**
division characteristic		position
attribute-equivalent	e-no	e-no

Besides, **Person@DB1**.car and **Car@DB2**.owner are *composition_hierarchy-equivalent*. In addition, **Student@DB1**.department and **CS-Student@DB2**, **EE-Student@DB2** are *attribute - class_set-equivalent*. **Employee@DB1**.position and **Faculty@DB2**, **Staff@DB2** are *attribute - class_set-equivalent*, too.

Then, according to the corresponding assertions specified, the integration rules are applied.

- Apply integration rules for *class-equivalent* classes:

1. Integrate **Person@DB1** and **Person@DB2**:
   - *Upgrade*(**Person@DB1**, [blood-type], blood, **Blood@DB2**) (step 2-2)
   - *Aggregate*(**Person@DB2**, [city, street, no], address, **Address@DB2**) (step 2-2)
   - *Invert*(**Car@DB1**, **Person@DB1**.car, owner) (step 3)
   - *Invert*(**Person@DB2**, **Car@DB2**.owner, car) (step 3)
   - *Refine*(**Person@DB1**, school, "NCTU") (step 4)
   - *Refine*(**Person@DB2**, school, "NTHU") (step 4)
   - *OUnion*(**Person@DB1**, **Person@DB2**, **Person**) (step 5)
2. Integrate **Address@DB1** and **Address@DB2**:
   - *OUnion*(**Address@DB1**, **Address@DB2**, **Address**) (step 5)
3. Integrate **Car@DB1** and **Car@DB2**:
   - *OUnion*(**Car@DB1**, **Car@DB2**, **Person**) (step 5)
4. Integrate **Student@DB1** and **Student@DB2**:
   The *division characteristic* of **Student@DB1** is *degree* and the *division characteristic* of **Student**
   @DB2 is *department*. We select *department* as the *division characteristic* of **Student** in the global schema.
   - *Demolish*(**Student@DB1**) (step 1-2)
   - *Build*(**Student@DB1**, **CS-Student@DB1**, [department="CS"]) (step 1-3)
   - *Build*(**Student@DB1**, **EE-Student@DB1**, [department="EE"]) (step 1-3)
   - *OUnion*(**Student@DB1**, **Student@DB2**, **Student**) (step 5)
5. Integrate **Employee@DB1** and **Employee@DB2**:
   The *division characteristic* of **Employee@DB2** is *position*.
   - *Build*(**Employee@DB1**, **Faculty@DB1**, [position="faculty"]) (step 1-3)
   - *Build*(**Employee@DB1**, **Staff@DB1**, [position="staff"]) (step 1-3)
   - *OUnion*(**Employee@DB1**, **Employee@DB2**, **Employee**) (step 5)
6. Integrate **CS-Student@DB1** and **CS-Student@DB2**:
   - *OUnion*(**CS-Student@DB1**, **CS-Student@DB2**, **CS-Student**) (step 5)
7. Integrate **EE-Student@DB1** and **EE-Student@DB2**:
   - *OUnion*(**EE-Student@DB1**, **EE-Student@DB2**, **EE-Student**) (step 5)
8. Integrate **Faculty@DB1** and **Faculty@DB2**:
   - *OUnion*(**Faculty@DB1**, **Faculty@DB2**, **Faculty**) (step 5)
9. Integrate **Staff@DB1** and **Staff@DB2**:
   - *OUnion*(**Staff@DB1**, **Staff@DB2**, **Staff**) (step 5)

– Put in those classes without corresponding assertions:
Blood@DB2 and **Course@DB2** are put into the global schema.

The resultant global schema is shown in Figure 11.

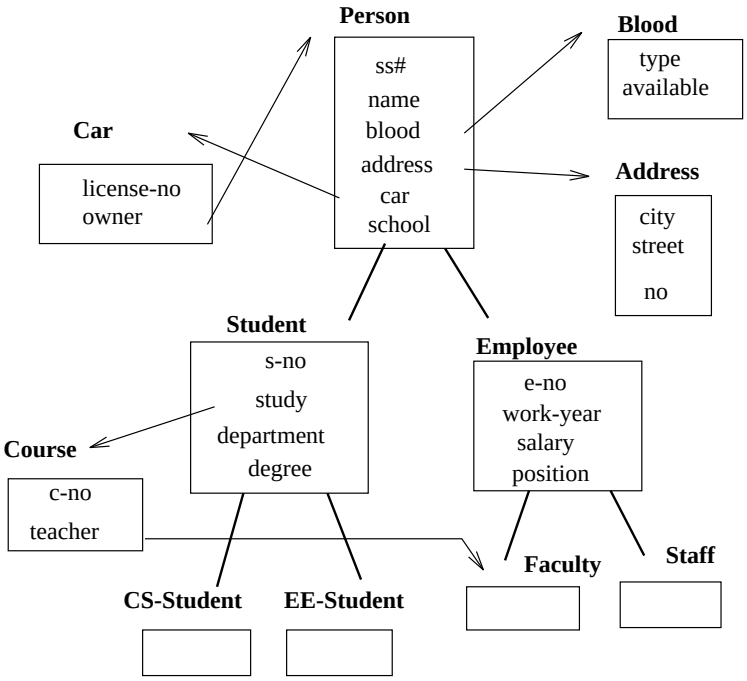


Figure 11: The integrated global schema

6 Conclusion and Future Work

In this paper, we present an object schema integration mechanism. We provide simple corresponding assertions for a DBA to specify the semantic correspondences between component schemas. Then, a set of primitive integration operators are defined to modify or integrate the component schemas. Finally, the integration rules are formulated to guide the integration process. Our strategy resolves conflicts of attribute structures or class hierarchies in component schemas, provides schema transparency, keeps the data in component databases retrievable without losing information, and allows queries to get more information from the integrated schema. *Mapping tables* are produced in the process of schema integration to store the mapping information between the global schema and component schemas for global query processing. They are documented in a separate paper [5].

The *composition_hierarchy-equivalent* is a noticeable structure conflict among component schemas. The integration rules for the general composition hierarchy equivalence and other more complex conflicts among component schemas will be studied in the future. In addition, *methods* in object schemas can also be considered in the process of schema integration. Furthermore, some restrictions can be involved in schema integration to provide a global schema which allows update operations.

References

1. C. Batini, M. Lenzerini, and S.B. Navathe, A comparative analysis of methodologies for database schema integration, *ACM Computing Surveys*, 18 (4) (1986) pp.323-364.
2. Y. Breitbart, P.L. Olson, and G.R. Thompson, Data integration in a distributed heterogeneous database system, *IEEE Second International Conference on Data Engineering*, (1986).
3. A.L.P. Chen, Outerjoin optimization in multidatabase systems, *Proc. IEEE International Symposium on Databases in Parallel and Distributed Systems* (DPDS), 1990.
4. A.L.P. Chen, P.S.M. Tsai, and J.L. Koh, Identifying object isomerism in multiple databases (submitted to Distributed and Parallel Databases Journal), 1993.
5. J.L. Koh and A.L.P. Chen, Mapping strategies for object schemas, *NTHU Technique Report* (1993).
6. B. Czejdo, M. Rusinkiewicz and D.W. Embley, An approach to schema integration and query formulation in federated database systems, *IEEE Third International Conference on Data Engineering*, (1987) pp.477-484.
7. U. Dayal and H.Y. Hwang, View definition and generalization for database integration in a multidatabase system, *IEEE Transactions on Software Engineering*, 10 (6) (1984) pp.628-644.
8. S.M. Deen, R.R. Amin, and M.C. Taylor, Data integration in distributed databases, *IEEE Transactions on Software Engineering*, SE-13 (7) (1987) pp.860-864.
9. L.G. DeMichiel, Resolving database incompatibility: an approach to performing relational operations over mismatched domains, *IEEE Transactions on Knowledge and Data Engineering*, 1 (4) (1989) pp.485-493.
10. R. Elmasri and S. Navathe, Object integration in logical database design, *IEEE First International Conference on Dta Engineering*, (1984) pp.426-433.
11. W. Gotthard, P.C. Lockemann, and A. Neufeld, System-guilded view integration for object-oriented databases, *IEEE Transactions on knowledge and Data Engineering*, 4 (1) (1992) pp.1-22.
12. S. Hayne and S. Ram, Multi-User view integration system (MUVIS) : An expert system for view integration, *IEEE Sixth International Conference on Data Engineering*, (1990) pp.402-409.
13. P. Johannesson, Schema transformations as an aid in view integration, *Stockholm Univ. Working Paper*, (1992).
14. M. Kaul, K. Drosten, and E.J. Neuhold, View System : integrating heterogeneous information bases by object-oriented views, *IEEE Sixth International Conference on Data Engineering*, (1990) pp.2-10.
15. W. Kent, Solving domain mismatch and schema mismatch problems with an object-oriented database programming language, Seventeenth International Conference on Very Large Data Bases, (1991) pp.147-160.
16. J.A. Larson, S.B. Navathe, and R. Elmasri, A theory of attribute equivalence in database with application to schema integration, *IEEE Transactions on Software Engineering*, 15 (4) (1989) pp.449-463.
17. M.V. Mannino and W. Effelsberg, Matching techniques in global schema design, *IEEE First International Conference on Data Engineering*, (1984) pp.418-425.
18. A. Motro, Superviews : Virtual integration of multiple databases, *IEEE Transactions on Software Engineering*, 13 (7) (1987) pp.785-798.

19. A. Sheth, J. Larson, A. Cornelio, and S. Navathe, A tool for integrating conceptual schemas and user views, *IEEE Fourth International Conference on Data Engineering*, (1988) pp.176-183.
20. S. Spaccapietra, C. Parent, and Y. Dupont, Model independent assertions for integration of heterogeneous schemas, *VLDB Journal*, (1) (1992) pp.81-126.
21. P.S.M. Tsai and A.L.P. Chen, Query uncertain data in heterogeneous databases, to appear in *Proc. IEEE International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems*, (1993).
22. F.S. Tseng, A.L.P. Chen, and W.P. Yang, Answering heterogeneous database queries with degrees of uncertainty, *Distributed and Parallel Databases Journal*, Kluwer Academic (to appear).
23. E.A. Rundensteiner and L. Bic, Set operations in object-based data models, *IEEE Transactions on Knowledge and Data Engineering*, 4 (3) (1992) pp.382-398.
24. E.A. Rundensteiner, Multiview : A methodology for supporting multiple views in object-oriented databases, *Eighteenth International Conference on Very Large Data Bases*, (1992) pp.187-198.

This article was processed using the LaTeX macro package with LLNCS style