A Rule-based Strategy for Schema Integration in a Heterogeneous Information Environment

Wolfgang Benn, Yangjun Chen and Ingo Gringer

Department of Computer Science

Technical University of Chemnitz-Zwickau, 09107 Chemnitz, Germany

Abstract In this paper, we propose a rule-based methodology to integrate databases in a heterogenous information environment. The methodology consists of two parts: schema translation and schema integration. First, based on the schema analysis, a set of rules is constructed to support the process of transforming schemata automatically (from relational schemata to object-oriented ones). Then, based on the algorithm for identifying subgraph isomorphisms, a strategy for integrating component schemata is developed, which identifies the semantic equivalence relationships between two component schemata semi-automatically. As with the schema transformation, a rule set is built to control the process of integrating heterogeneous data schemata.

1. Introduction

Interoperability is becoming a critical issue for many organizations today, due to the increasing dependence and cooperation between them. Therefore, it is desired for many enterprises to be able to access remote as well as local information sources [BOT86, LA86, SK92]. Further, even within a single enterprise several independent information bases may exist as a result of department autonomy [HLM94]. Thus, for many organizations it is important to interconnect existing, possibly heterogeneous, information systems for performing a cooperation. An essential part of this activity is database integration, i.e., the process of constructing a global schema from a collection of existing databases.

As an example, consider the query which enquires, for the statistical purpose, all the information such as payment, tax, family state, pertinent to citizens in Chemnitz who pay a tax of more than 500 DM per month. For answering this query, many databases with different data schemas and different data representations may be involved. Then, a integrated schema for them is needed to support such an application.

The database integration process can be divided into two phases: schema transformation and schema integration [RPRG94]. The schema transformation involves analyzing the local data schemata and then translating them into a single data model for comparison purposes. The schema integration involves resolving the heterogeneities such as naming conflicts, domain conflicts, meta-data conflicts and structural conflicts among the local schemata [CW93] and superimposing the schemata in order to obtain one consistent global schema. Normally, such tasks are very time-consuming processes if they are done manually. Therefore, it is required to develop a mechanism such that most of tasks can be performed automatically or semi-automatically based on it.

The work reported here is supported by DFG (Deutsche Forschungsgemeinschaft) under grant No. Be1786/1-1.

In this paper, a systematic procedure for virtual integration of component databases is proposed for the purpose mentioned above, which consists of an automatical approach to the schema transformation (from relational schemata to object-oriented ones) and a semi-automatical approach to the schema integration of local data schemata. As compared to existing procedures [DH84, DAT87, Jo93, KC93, LA86, LNE89, RPRG94, Te87, TS93], this procedure lays stress on the rule-based method to develop a flexible strategy for translating and integrating local schemata. In addition, a graph algorithm for identifying subgraph isomorphisms [CH81] is utilized to facilitate the establishment of semantic equivalence relationships among the component data schemata.

The paper is organized as follows. In Section 2, we discuss the main problems with respect to the database integration. In Section 3, we propose our methods for transforming and integrating local schemata, along with several examples for illustrating the main ideas. Section 4 details the implementation modules. The last section summarizes our conclusions with recommendations for future work.

2. Main Problems w.r.t. Database Integration

Essentially, the database integration process can be characterized using a three-layered schema architecture: local schemata, local abstraction schemata and global (integrated) schema as shown in Fig. 1.

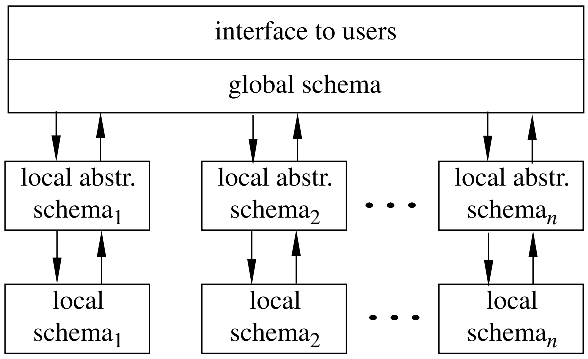

Fig. 1. Three-layered architecture for database integration

The bottom layer consists of a set of local schema. Each local schema can be denoted by DB_i ($i = 1, 2, ..., n$). These schemata provide a description of the data stored in their respective data models. The stored data can be retrieved only by using their respective languages. The second layer consists of a set of local abstraction schemata, which are derived from the corresponding local schemata. There are several motivations for translating a local relational schema to an abstraction schema: first, to achieve data model homogeneity among the component database schemata; second, to present clear and logical views of entities present in the local schemata; and third, to store all the semantic knowledge gathered about the objects in the local schemata. The top layer consists of the integrated schema, which provides a global view over the component data schemata and supports integrated access to these autonomous and heterogeneous information bases.

In order to achieve a global schema, a lot of problems have to be investigated. For example, for the schema transformation, the local schemata should be analyzed and their different structures should be recognized. Then, based on the schema analyses, various strategies should be developed to translate local schemata into a single one. On the other hand, for the schema integration, the heterogeneities among the local schemata should be removed. In fact, the most difficult part of the database integration process is the resolution of heterogeneities. The fundamental problem here is that the same real-world entities can be modelled in different ways. Also, the same phenomenon can be seen from different levels of abstraction, or represented using different properties.

In the following, we discuss these problems in detail.

- Problems w.r.t. Schema Transformation

As the first step of database integration, we translate each local schema into an abstraction schema. To this end, we choose the object-oriented schema to represent the integrated information, due to its natural framework for integrating heterogenous components and its mechanism of the message protocols which is necessary in a distributed environment [Ma90a, Ma90b]. Then, we develop a strategy to transform a local schema into a object-oriented one (in our research, only the transformation from relational schemata to the object-oriented schema is considered.) In general, the following issues should be addressed.

- local schema analysis
  Normally, a relational schema may contain several kinds of relation schemata such as strong entity relations, weak entity relations, strong relationship relations and weak relationship relations [Ch76]. For each kind of relation schemata, the transformation approach should be different due to their distinguished key structures. Therefore, such relations should be first recognized to guarantee that for each of them the accurate schema transformation is performed.

- schema transformation
  For each different kind of relation schemata, a transformation strategy should be developed to build the corresponding objects, the *is-a* links and the *aggregation* links. Besides, in terms of the reference relationships among the relation schemata, some forms of constraints will be established to specify the value ranges of the relevant attributes and the semantic relationships between objects, which can not be expressed by class/subclass or part-of concepts. Further, the inheritance mechanism should be constructed based on the schema analyses and the value distribution of attributes.

- formalization of strategies for the schema transformation
  It is important to formalize the strategies to underlie an automatical process of transforming schemata. Especially, different semantic constraints should be analyzed and recognized so that the corresponding formalization can be done uniformly.

- Problems w.r.t. Schema Integration

After the schema transformation, we derive a global schema by integrating the corresponding elements of the participating local schemata. For this purpose, the following problems should be handled.

- heterogeneity elimination

The semantic heterogeneity occurs in the local schemata due to dissimilar perceptions of the same reality by different individuals, dissimilar data modeling skills of designers of various schemata, and differences in semantic richness of data models used in these schema definitions. In general, four types of conflicts among data schemata modelling the same real-world can be identified: *name conflicts* when different databases use different names to represent the same concepts, *domain conflicts* when different databases use different values to represent the same concepts, *meta-data conflicts* when the same concepts are represented at the schema level in one database and at the instance level in another, and *structural conflicts* when different databases use different data organization to represent the same concepts.

- schema integration
  If the equivalence relationship between two local schemata is established, the schema integration can be done by merging the equivalent elements of the two schemata and restructuring the class hierarchies to accommodate those parts which have no equivalent counterparts in the other local schema.
- formalization of strategies for the schema integration
  As with the schema transformation, we need to formalize the process of integrating schemata. Although the elimination of semantic conflicts can not be automatized completely, the formalization makes a semi-automatic process possible.

3. Rule-based Strategies for Schema Transformation and Schema Integration

In this section, we propose our strategies for schema transformation and schema integration. The key idea behind our methods is to build a knowledge base serving as the control mechanism. In addition, to facilitate the establishment of semantic equivalence relationships between any two component data schemata, a methodology for finding equivalent elements is developed based on subgraph isomorphisms.

3.1 Schema Transformation

To facilitate the identification of equivalent entities in different (relational) database schemata, the schemata must first be translated into a single abstraction data model for comparison purposes. The primary objectives for constructing local object-oriented schema are to achieve data model homogeneity by transforming local schemata expressed in different data model into a single data model and to achieve uniformity in the modeling constructs. The construction of a global schema will be easier if all the local schemata pertain to the same data model. The second goal is to present a clear logical view of the entities in the local schema. In a local schema, information pertinent to an entity may be scattered over more than one data structure (e.g., in a relational schema, over more than one relation). In order to obtain a clear perception of entities, one wish to view all the information available for that entity as a single unit. i.e., to have an integrated view of the entity irrespective of its distribution in the local schema.

3.1.1 Schema Analyses

Here, we consider only relational schemata. As mentioned in the last section, we classify the relations in a relational database into four classes: *strong entity relations*, *weak entity relations*, *strong relationship relations* and *weak relationship relations*, which can be defined as follows.

1) A strong entity relation is a relation whose primary key does not properly contain a key of any other relation. For instance, in Fig. 2 (see below), *university, courses* and *professors* are typical examples of strong entity relations.

2) A weak entity relation is a relation that satisfies all of the following requirements:

   i) A proper subset of its primary key, denoted K_1, contains the keys of other entity relations. Each key (of some other entity relation) contained in K_1 is called a *supporting key.*

   ii) The remaining attributes of the primary key, denoted K_2, do not contain a key of any other relation.

   iii) This relation represents an entity type whose identifier must properly contain identifier attributes of other entity types (i.e. K_1). This requirement must be confirmed in some way (e.g., through the interaction with users).

As an example, consider *department* relation in Fig. 2. Its primary key contains the primary key, 'u-name', of *university* which is a strong relation. Moreover, 'd-name' is not a key of any other relation, and *department* represents an entity type, then it is a weak entity relation.

3) A strong relationship relation is a relation whose primary key is formed fully by the concatenation of keys of some entity relations. For example, the relation *course-prof.* in Fig. 2 is a strong relationship relation, because its primary key, [*course#, ssn#*], is formed fully by the concatenation of the primary keys of *courses* and *professors.*

4) A weak relationship relation is a relation whose primary key is constructed partially by the concatenation of keys of some other entity relations and partially by some attributes of its own. Moreover, it can not be treated as a weak entity relation. Consider the relation *teaching-plan* in Fig. 2. Its key is [*time, classroom, course#*] and "time" is not a key of any other relation. Then *teaching-plan* is a weak relationship relation.

university:	[*u-name, city, president*]
department:	[*d-name, u-name, director*]
courses:	[*course#, c-name*]
person:	[*ssn#, name, age, sex]*
professors:	[*ssn#, name, d-name, u-name*]
course-prof:	[*course#, ssn#*]
teaching-plan:	[*time, classroom, ssn#*]

(In the relational schema, all the keys are underlined.)

Fig. 2. A sample relational schema

For recording this classification information, four meta-relations will be created: *meta-strong-entity*, *meta-weak-entity*, *meta-strong-relationship* and *meta-weak-relationship* as shown in Fig. 3.

In the relation of *meta-strong-entity*, the attribute 'attr-property' is used to describe attribute's properties. In our design, '0' represents a no-key attribute; '1' represents a primary key; and any foreign key appearing in a strong entity relation schema will be numbered with a different *meta-strong-entity*:

attribute	*type*	*description*
rn	string	relartion name
an	string	attribute name
attr-property	integer	attribute property

meta-weak-entity:

attribute	*type*	*description*
rn	string	relartion name
an	string	attribute name
attr-property	integer	attribute property
skl	{*true, false*}	label supporting key

meta-strong-relationship:

attribute	*type*	*description*
rn	string	relartion name
an	string	attribute name
key-label	{*true, false*}	label foreign key
key-no.	integer	number foreign key

meta-weak-relationship:

attribute	*type*	*description*
rn	string	relartion name
an	string	attribute name
key-label	{*true, false*}	label foreign key
key-no.	integer	number foreign key

Fig. 3. Meta-data structures w.r.t. relation classification

integer. The attributes numbered with the same integer belong to the same (multiple) foreign key. In the relation *meta-weak-entity*, the attribute "skl" is used to mark a supporting key. Note that *meta-strong-relationship* and *meta-weak-relationship* have the same structure, in which the attribute "key-label" is used to mark a key, while the attribute "key-no." is used to number a foreign key with the exception that "0" represents a single key (of some other relation schema) appearing in the corresponding relation schema.

For example, for our sample relational schema, the following four relations will be generated.

meta-strong-entity:

rn	an	attr-property
university	u-name	1
university	city	0
university	president	0
courses	course#	1
courses	c-name	0
professor	ssn#	1
professor	name	0
professor	d-name	2
professor	u-name	2
person	ssn#	1
• • •	• • •	• • •

meta-weak-entity:

rn	an	attr-property	skl
department	d-name	1	*false*
department	u-name	1	*true*
department	director	0	*false*

meta-strong-relationship:

rn	an	key-label	key-no.
course_prof.	course#	*true*	0
course_prof.	ssn#	*true*	0

meta-weak-relationship:

rn	an	key-label	key-no.
teaching_plan	time	*false*	1
teaching_plan	classroom	*false*	1
teaching_plan	ssn#	*true*	1

Fig. 4. Meta data for the sample database

Based on the above meta-data structures, a set of rules supporting schema transformation can be defined, which we will explain in the next subsection.

3.1.2 Rule-based Strategy for Schema Transformation

For building a rule set for translating local schemata, some built-in functions and semantic constraints have to be first defined, which are necessary to construct classes and to specify the semantic relationships among these classes. For our purposes, the following functions and constraints are implemented.

- Built-in Functions:

class(x)*:* find the class which has been created in terms of a relation schema named x.

$C(x)$: find the class which has been created in terms of a strong relation whose key attribute is x.

$C_{weak}(x)$: find the class which has been created in terms of a weak relation whose supporting key is x.

- semantic constraints:

$\delta_1(x, y, z)$: if x is a value of attribute y of some object (represented by the actual value of '*self*' variable) of the current class then $o^{-1}(self)$ should be a value of $o(x).z$, where $o(x)$ represents an object identifier which is determined by x (some key value), $o(x).z$ represents attribute z which appears in object $o(x)$ and o^{-1} represents the inverse function of o.

If we use the following structure to represent an object:

$$obj\text{-}id[a_1\text{: } t_1, ..., a_l\text{: } t_l, ma_1\text{: } \{s_{11}, ..., s_{1i_1}\}, ..., ma_k\text{: } \{s_{k1}, ..., s_{ki_k}\}],$$

$\delta_1(x, y, z)$ can be illustrated as follows:

$$self[...\ y\text{: } \{...\ x\ ...\}\ ...] \vee self[...\ y\text{: } x\ ...] \rightarrow o(x)[...\ z\text{: } \{...\ o^{-1}(self)...\}\ ...] \vee o(x)[...\ z\text{: } o^{-1}(self)\ ...].$$

Here *obj-id* represents an object identifier; a_i $(1 \le i \le l)$ represents a single-valued attribute; and ma_j $(1 \le j \le k)$ represents a set-valued attribute. Note that in the semantic constraint *self* can be understood as a variable which is always assigned the current object identifier when the corresponding object is referenced by an application. We demonstrate the usage of $\delta_1(x, y, z)$ using the following example.

Example 1. Consider *course-prof*: [<u>*course#, p-ssn#*</u>] of the sample relational schema. If we have created two classes for *professor* and *courses*, respectively, and choose not to generate a class for *course-prof*, we can establish two semantic constraints of the above form to record the relevant information as shown in Fig. 5(b). These two semantic constraints (along with two attributes: *course#* and *p-ssn#*) are inserted into class *professor* and class *courses*, respectively. Then, two objects shown in Fig. 5(c) will both satisfy the constraints. Of course, we can create a class for *course-prof*, but with another form of semantic constraints inserted (i.e., δ_2, see below). □

$\delta_2(x, y, z)$: if x is a value of attribute y then x should be an instance of object z.

As with semantic constraint λ_1, we use the following representation for illustration.

$\delta_2(x, y, z)$: $self[...\ y\text{: } \{...\ x\ ...\}\ ...] \vee self[...\ y\text{: } x\ ...] \rightarrow isa\text{-}instanceOf(o(x), z)$.

course-prof:

course#	*p-ssn#*
c_1	p_1
. . .	. . .

(a)

class *professor*
attribute
p-ssn#: *integer*
. . .
course#: *integer*
. . .
δ_1(*s*, *course#*, *p-ssn#*)
endclass

class *courses*
attribute
course#: *integer*
. . .
p-ssn#: *integer*
. . .
δ_1(*s*, *p-ssn#*, *course#*)
endclass

(b)

object $o(p_1)$
attribute
. . .
course#: c_1
. . .
endobject

object $o(c_1)$
attribute
. . .
p-ssn#: p_1
. . .
endobject

(c)

Fig. 5. Illustration for the usage of semantic constraint δ_1

Example 2. To demonstrate the usage of δ_2, consider relation schema *department*: [<u>*d-name, u-name,*</u> *director*]. If we generate a class for it, a semantic constraint of the form δ_2 should be created to construct the semantic relationship between it and the class generated in terms of relation schema *university*: [<u>*u-name,*</u> *city, president*] (see Fig. 6). □

class *department*
attribute
d-name: *string*
u-name: *string*
director: *string*
δ_2(*s*, *u-name*, *university*)
endclass

Fig. 6. Illustration for the usage of semantic constraints δ_2

As mentioned in Example 1, δ_2 can be used to establish the semantic relationship between the class created in terms of a strong relationship relation schema and the class created in terms of some foreign key of it. See Example 3 for explanation.

Example 3. To illustrate another use of δ_2, consider *course-prof*: [<u>*course#, p-ssn#*</u>] once again. If we choose to generate a class for it, two semantic constraints of the form δ_2 should be inserted into the class as shown in Fig. 7. (Assume that classes: *professor* and *courses* have been generated.) □

class *course-prof*
attribute
course#: *integer*
p-ssn#: *integer*
. . .
$\delta_2(s_1$, *course#*, *courses*)
$\delta_2(s_2$, *p-ssn#*, *professor*)
endclass

Fig. 7. Illustration for the usage of semantic constraints δ_2

$\delta_3(x, y, z, w)$: if x is a value of attribute y then x should be a value of attribute w of object z.

The following set of rules (or Horn clauses [Ll87]) will be partitioned into six groups. The first four groups are used to generate objects and the corresponding semantic constraints in terms of different kinds of relation schemata. The fifth group is used to generate *is-a* links and aggregation (*agg*) links. At last, the sixth group is used to analyze attribute features (w.r.t. inheritances) based on some epistemologic principles.

- the first group of rules:

r_1: if *meta-strong-entity*(x, y, _) then *generate-class*(x, y),

where "_" means "do not care". This rule states that if x is a strong entity relation name with some attribute y then an class x will be generated with y being an attribute of it. Obviously, by executing this rule repeatedly, we can generate the corresponding class for each strong entity relation in a relational database. One should not be confused that a class will be generated several times. In fact, the predicate *generate-class*(x, y) is defined as follows.

$$generate-class\,(x, y) = \begin{cases} \text{generate class } x \text{ with } y \text{ being an attribute} & \text{if } x \text{ does not exist} \\ \text{insert } y \text{ into } x & \text{if } x \text{ already exists} \end{cases}$$

- the second group of rules:

r_2: if *meta-weak-entity*(x, y, _, _) then *generate-class*(x, y),

r_3: if *meta-weak-entity*(x, y, 1, *true*) then *insert*($\delta_2(z, y, C(y))$, *class*(x)).

The first of these two rules is used to generate classes according to weak entity relation schemata. The second is used to construct the semantic relationship between a weak entity and its supporting entity (a supporting entity of a weak entity is an entity whose key appears as a supporting key in the weak entity). That is, a semantic constraint of the form: $self[...\ y: \{...\ z\ ...\}\ ...] \vee self[...\ y: z\ ...] \rightarrow isa\text{-}instanceOf(o(z), C(y))$ will be inserted into the class represented by *class*(x), expressing that the value of a supporting key attribute of an object which belongs to a class created in terms of a weak entity relation schema corresponds to an instance of the class created in terms of the supporting entity (which is represented by $C(y)$ in r_3.)

- the third group of rules:

r_4: if *meta-strong-relationship*(x, y, u, v), u = *false*, v = 0 then *no-key*(x, y).

This rule is used to check whether a strong relationship relation schema contains an attribute which is not a foreign key. It is important because for strong relationship relation schemata the method for generating objects will be different in terms of whether a strong relationship relation schema contains any no-key attribute. That is, if a strong relationship relation schema contains no no-key attributes, we do not create a class for it (as discussed in Example 1). Instead, we construct a δ_1 semantic constraint for each class which is generated in terms of a relation schema whose key appears in the relationship relation schema as a foreign key (see r_{11}). If a strong relationship relation schema contains a no-key attribute which is not covered (we say that a no-key attribute is covered if it appears in some entity relation schema whose key participates in the relationship relation schema; see r_5), a class will be generated for it. Otherwise, no class will be created at all. As in the case that a strong relationship relation schema contains no no-key attributes, a δ_1 semantic constraint is inserted into the corresponding class (see r_{12}).

r_5: if *no-key*(x, y), *meta-strong-relationship*(x, z, *true*, _), *in*(y, $C(z)$) then *covered*(x, y).

This rule is used to define predicate *covered*(x, y), in which *in*(y, $C(z)$) checks whether 'y' appears as an attribute in '$C(z)$'.

r_6: if *meta-strong-relationship*(x, y, _, _), $\exists z$(*no-key*(x, z), $\neg$*covered*(x, z))

then *generate-class*(x, y).

r_7: if *meta-strong-relationship*(x, y, *true*, 0), $\exists z$(*no-key*(x, z), $\neg$*covered*(x, z))
then *insert*($\delta_2(u, y, C(y))$, *class*(x)).

These two rules are used to create classes and the corresponding semantic constraints, respectively, in terms of the strong relationship relation schemas which contain some uncovered attributes. Note that by executing r_7, corresponding to each foreign key y, a semantic constraint of the form: *self*[... y: {... u ...} ...] $\vee$ *self*[... y: u ...] $\rightarrow$ *isa-instanceOf*($o(u)$, $C(y)$) will be inserted into the class represented by *class*(x). In this way, the semantic relationship between *class*(x) and the class created in terms of an entity relation schema whose key corresponds to y can be established.

r_8: if ($\neg$*no-key*(x, y) $\vee$ $\forall t$(*no-key*(x, t) $\Rightarrow$ *covered*(x, t)) then *pure-relationship*(x),

where '$\Rightarrow$' represents the logic implication. This rule states that if a strong relationship relation schema contains no no-key attribute or all of its no-key attributes are covered, then it is a pure relationship relation schema, for which no class will be generated. As mentioned above, in this case, several pairs of semantic constraints will be established to specify the relevant semantic relationships (see the following rule).

r_9: if *pure-relationship*(x), *meta-strong-relationship*(x, y, *true*, 0),
meta-strong-relationship(x, z, *true*, 0)
then *insert-attr*(y, $C(z)$), *insert-attr*(z, $C(y)$)
insert($\delta_1(u, y, z)$, $C(z)$), *insert*($\delta_1(v, z, y)$, $C(y)$).

For each pair of foreign key attributes, say y and z, appearing in a strong relationship relation schemas, respectively, this rule first creates a new attribute named y in the class represented by $C(z)$ and a new attribute named z in the class represented by $C(y)$ (by executing *insert-attr*(y, $C(z)$) and *insert-attr*(z, $C(y)$)). Then, it inserts δ_1 semantic constraints pairwise into the classes generated in terms of those entity relation schemas whose keys appear in the strong relationship relation schema as foreign keys. That is, if y and z are two foreign keys appearing in x, a semantic constraint of the form: *self*[... z: {... s ...} ...] $\vee$ *self*[... z: s ...] $\rightarrow$ $o(s)$[... y: {... o^{-1}(*self*) ...} ...] $\vee$ $o(s)$[... y: o^{-1}(*self*) ...] will be inserted into the class represented by $C(y)$, while another semantic constraint of the form: *self*[... y: {... t ...} ...] $\vee$ *self*[... y: t ...] $\rightarrow$ $o(t)$[... z: {... o^{-1}(*self*) ...} ...] $\vee$ $o(t)$[... z: o^{-1}(*self*) ...] will be inserted into the class represented by $C(z)$ (once again, see Example 1 for illustration). Note that before inserting a new attribute into a class, it should be checked whether the attribute already exists in the corresponding class. Therefore, the above rule should take a more complicated form to accomplish this work. Here, however, we omit the detailed discussion for simplicity.

The following three rules will be used to deal with more complex situations that some foreign key(s) of a strong relationship relation schema correspond to a supporting key of a weak entity relation.

r_{10}: if *pure-relationship*(x), *meta-strong-relationship*(x, y, *true*, 0),
meta-strong-relationship(x, z, *true*, w), $w \neq 0$
then *insert*($\delta_1(u, y, z)$, $C_{weak}(z)$), *insert*($\delta_1(v, z, y)$, $C(y)$).

r_{11}: if *pure-relationship*(x), *meta-strong-relationship*(x, y, *true*, t), $t \neq 0$
meta-strong-relationship(x, z, *true*, w), $w \neq 0$
then *insert*($\delta_1(u, y, z)$, $C_{weak}(z)$), *insert*($\delta_1(v, z, y)$, $C_{weak}(y)$).

r_{12}: if *meta-strong-relationship*(x, y, *true*, z), $z \neq 0$, $\exists z$(*no-key*(x, w), $\neg$*covered*(x, w))
then *insert*($\delta_2(u, y, C(y))$, *Class*(x)), *insert*($\delta_3(u, y, C_{weak}(y), y)$, *Class*($x$)).

Rule r_{10} is used to cope with the case that only one of the considered pair of foreign keys corresponds to a weak entity relation schema, while r_{11} is used to handle the case that both the foreign keys come from weak entity relation schemas. In addition, by r_{10} and r_{11}, only pure relationship relation schemas are considered. On the contrary, r_{12} is used to construct semantic constraints for those classes which are created in terms of 'not-pure' relationship relation schemas which contain supporting keys as foreign keys. The ideas of these three rules are similar to r_7 and r_9 except that supporting keys are considered.

The fourth group of rules is used to generate objects in terms of weak relationship relation schemata.

- the fourth group of rules:

r_{13}: if *meta-weak-relationship*(x, y, _, _) then *generate-class*(x, y)

r_{14}: if *meta-weak-relationship*(x, y, *true*, _) then *insert*($\delta_2(u, y, C(y))$, *class*(x)).

As for weak entity relation schemas, a class will be created for each weak relationship relation schema, along with a δ_2 semantic constraint constructed to specify the corresponding semantic relationships. (For simplicity, here we do not give the rules for dealing with composite keys which appear in a weak relationship as foreign keys. The corresponding rules can be constructed in the same way as for strong relationship relation schemas.)

In the following, we give the fifth group of rules, the rules for establishing *is-a* and *agg* links. First, we define two important concepts (see [CBS94, MM90, VA95] for original definitions).

Definition 1. (*compatible*) Let $R_i(X_i)$ and $R_j(X_j)$ be two relation-schemata, where X_i and X_j denotes the attribute sets for R_i and R_j, respectively. We say two attributes x ($\in X_i$) and y ($\in X_j$) to be *compatible* if they are associated with the same domain. Two attribute subsets Y ($\subseteq X_i$) and Z ($\subseteq X_j$) are said to be compatible iff there exists a one-to-one correspondence of compatible attributes between Y and Z. □

Definition 2. (*inclusion dependency*) Let $R_i(X_i)$ and $R_j(X_j)$ be two relation-schemata associated with relations re_i and re_j, respectively. An *inclusion dependency* w.r.t. the current database state (a database state is a set of relations $\{re_1, re_2, ..., re_n\}$, where each re_k is a relation for R_k) is a statement of the form $V(Y) \subseteq V(Z)$, where Y and Z are compatible subsets of X_i and X_j, respectively; $V(Y) \subseteq V(Z)$ is satisfied by re_i and re_j iff $\pi_Y(re_i) \subseteq \pi_Z(re_j)$, where $\pi_X(re_k)$ denotes the projection of re_k on X. □

In the following group of rules, we denote two compatible attributes x and y by *compatible*(x, y). In addition, the number of tuples of a relation r is denoted *cardinality*(r).

- the fifth group of rules:

r_{15}: if *meta-strong-entity*(x_1, y_1, 1), *meta-strong-entity*(x_2, y_2, 1), *compatible*(y_1, y_2), *Values*(y_2) $\subseteq$ *Values*(y_1)
then *subclass*(*class*(x_2), *class*(x_1)).

This rule is used to establish class/subclass relationships based on the inclusion dependency.

r_{16}: if *meta-strong-entity*(x, y, z), $z > 1$ then *foreign-key*(x, y).

This rule is used to collect all the foreign keys appearing in strong entity relation schemas into a temporary relation '*foreign-key*(v, z)'.

r_{17}: if *meta-strong-entity*(x, y, z), $z > 1$, *cardinality*(*foreign-key*(x, _)) = 1,
then *insert*(δ_2(u, y, C(y)), *class*(x)).

If a strong entity relation schema contains only one foreign key, a δ_2 semantic constraint will be constructed (for the class created in terms of the relation schema) by this rule.

r_{18}: if *meta-strong-entity*(x, y, z), $z > 1$, *cardinality*(*foreign-key*(x, _)) > 1,
then *AggregationOf*(*class*(x), C(y)).

If a strong entity relation schema contains n ($n > 1$) foreign keys, then n part-of links will be established (with each link corresponding to a foreign key) by this rule

Trivially, the following rule will be constructed to represent the transitive property of *subclass* relationship.

r_{19}: if *is-a*(x, y), *is-a*(y, z) then *is-a*(x, z).

The sixth group of rules is used to identify own-attributes from instance-attributes for any generated class. To this end, three epistemologic principles are considered:

1) The specification of a concept into some subconcepts is based on the values of at least one attribute of it. All the instances of a subconcept possess the same values of the corresponding attributes. (Therefore, these values of the attributes become the character of the corresponding subclass, distinguishing it from others.)

2) The union of all instances of the subconcepts of a concept should be equal to the set of instances of the concept.

3) Two concepts with the compatible key attributes are in the same semantic category. That is, both of them should be involved in the same generalization-specification hierarchy.

The principles motivate the following definition.

Definition 3. Let R be a relation schema and *attr* be an attribute of it. If the number of values of *attr* (denoted *cardinality*(*attr*)) is less than some constant *Const*, then *attr* is called a *division* *attribute* w.r.t. *R*. Such a constant *Const* is chosen based upon the ration of the number of distinct values in an attribute over the number of the tuples of the corresponding relation and the largest fan-out of the local generalization/specification hierarchies. □

The observation leading to this definition is that it is possible to partition a class into several subclasses, in terms of some division attribute, with each possessing a value of the division attribute. For example, attribute *person*.sex in the sample database is a division attribute. Its value set is {man, woman}, in terms of which subclasses *man* and *woman* can be generated. This partition is important in the context of schema integration, since in this way we can eliminate the meta-data conflict, i.e., the conflict appearing when the same concepts are represented at the schema level in one database and at the instance level in another.

If such a partition has be done, each value of the division attribute corresponds to the name of a subclass generated as above. Therefore, the corresponding attribute should be handled as a partially own attribute of the original class. That is, this attribute should not be inherited by the subclasses generated by partitioning the corresponding attribute. For illustration, consider *person*.sex again. If the subclasses *man* and *woman* are generated by partitioning *person*.sex, then *man* and *woman* should not inherit the attribute "sex" from *person* for the redundancy reason. However, the other subclasses such as *professor, student* and *employee* should inherit it. In addition, if the cardinality of a division attribute is one, this attribute becomes trivially the own attribute of the corresponding class.

Based on the above discussion, we construct the following rules to distinguish among division attributes, (partially) own attributes and instance attributes.

- the sixth group of rules:

r_{20}: if *attr*(*x, y*), *cardinality*(*y*) = 1 then *own-attr*(*x, y*).

r_{21}: if *attr*(*x, y*), *cardinality*(*y*) $\leq$ *Const* then *division-attr*(*x, y*).

r_{22}: if *division-attr*(*x, y*), *partitioned*(*y*), $z \in$ *Values*(*y*) then *partially-own-attr*(*x, y, z*).

Rule r_{20} is used to define own attributes, in which predicate *attr*(*x, y*) holds iff *y* is an attribute of class *y*. Rule r_{21} is used to define division attributes in terms of Definition 3. Finally, r_{22} says that if *y* is a division attribute of *x* and partitioned, then *y* is a partially own attribute of *x* w.r.t. the subclasses created in terms of the values of *y*. By executing '*generate-subclass*(*x*, *y*, *z*)', a subclass corresponding to *z* (a value of attribute *y*) will be generated, possessing all attributes of *x* except *y*; and by executing '*partially-own-attr*(*x, y, z*)', *y* will be labeled as a partially own attribute of *x* w.r.t. *z*.

3.1.3 A Sample Transformation

To illustrate briefly the usage of the rule set given above, let us trace a sample transformation. The example is so constructed that major ideas can be presented in a simple way.

Example 4. Consider the relational schema given in Fig. 8.

In order to translate this relational schema into an object-oriented one, we perform the inference by executing the numbered (forward reasoning) rules in sequence.

university:	[*u-name, city, president*]
department:	[*d-name, u-name, director*]
courses:	[*course#, c-name*]
person:	[*ssn#, name, age, sex*]
professor:	[*p-ssn#, name, d-name, u-name*]
course-prof:	[*course#, ssn#*]
teaching-plan:	[*time, classroom, ssn#*]
project:	[*pro#, project-director, fountation*]
foundation:	[*foun#, foun-name, sponsor*]

(In the relational schema, all the keys are underlined.)

Fig. 8. A relational schema

First, based on the first four groups of rules, we transform the relation schemas into a series of classes, together with some semantic constraints inserted. (See Fig. 9(a) for illustration.) Then, by executing the fifth group of rules (from r_{15} to r_{18}), a set of links as shown in Fig. 9(b) will be established, leading to a complete object-oriented schema. Finally, by performing the sixth group of rules (from r_{21} to r_{22}), we can partition *person.sex* of *person* and generate two subclasses: *man* and *woman*.

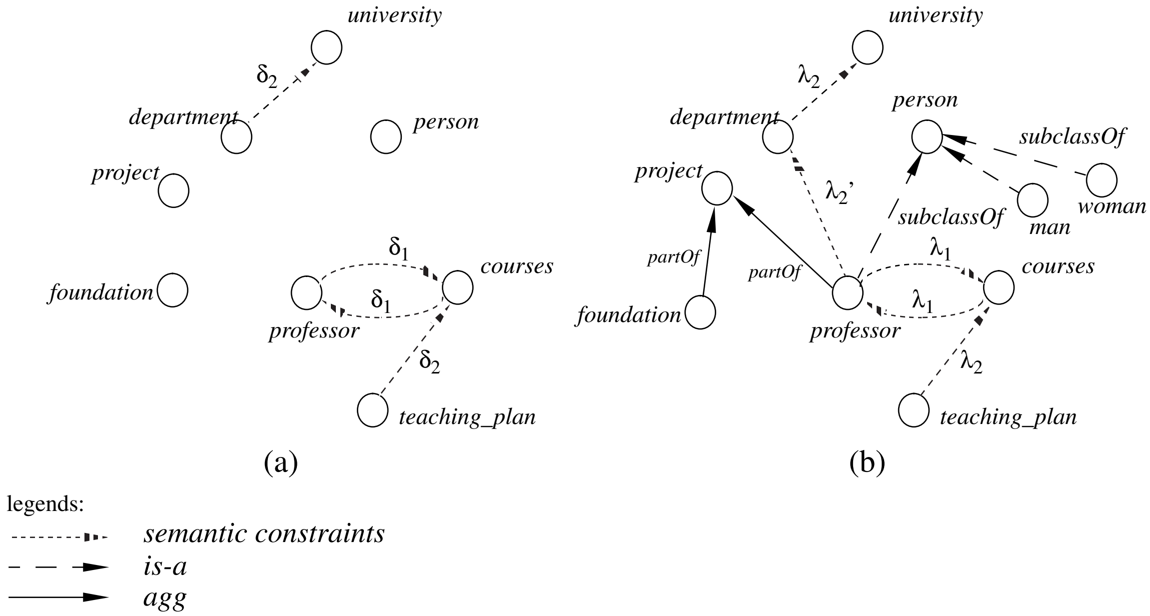

Fig. 9. The derived object-oriented schema

In the following, we explain the entire process step-by-step.

By executing r_1 repeatedly, the classes: *university, person, professor, courses, project* and *foundation,* together with the corresponding attributes, will be generated, each of which corresponds to a strong entity relation. By executing rules r_2, and r_3, class *department* will be produced with a semantic constraint of the form δ_2 inserted into it to specify the semantic relationship between it and its supporting entity. Since the relationship relation *course-prof* is a 'pure-relationship', no class is generated for it. (Neither rules from r_4 to r_5 nor rules from r_6 to r_7 will be performed because the condition-parts of them are not satisfied by the relational schema.) Instead, two semantic constraints of the form δ_1 will be inserted into *professor* and *cours-*

es, respectively, by executing r_8 and r_9. Further, by executing r_{13} and r_{14}, class *teaching-plan* will be generated with a semantic constraint of the form δ_2 inserted into it. (Note that rules from r_{10} to r_{12} will not be executed because the condition-parts of them are not satisfied.)

By performing r_{15}, the *is-a* relationship between *professor* and *person* will be built. Then, by executing rules from r_{16} to r_{18} repeatedly, another semantic constraint of the form δ_2 for *professor* (by r_{17}) and two *agg* links from *foundation* and *professor* to *project*, respectively, will be generated (see Fig. 9(b)). Finally, by executing r_{21} (r_{20} will not be executed because its condition-part is not satisfied by any class), attribute *person.sex* will be identified as a division attribute if the system constant *Const* is set to be a constant larger than two. Further, if the partition operation has been done, r_{22} will be preformed. In this case, two extra classes *man* and *woman* will be generated and *person.sex* will be labelled as a partially own attribute.

□

3.2 Schema Integration

The schema integration is a more difficult task than the schema transformation. Up to now, all the existing methods are based on the semantic equivalence built manually. In our research, we try to develop a semi-automatical process to establish semantic equivalences by utilizing subgraph isomorphisms. In the following, we present this methodology in Subsection 3.2.1. Then, in Subsection 3.2.2, we construct a set of rules to control the integration process.

3.2.1 Identification of Semantic Equivalence

Our methodology for identifying semantic equivalences can be depicted in Fig. 10.

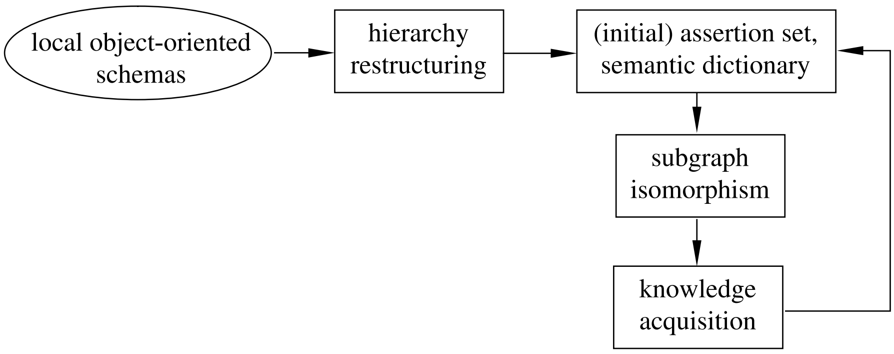

Fig. 10. Process for identifying semantic equivalences

It starts with a process for restructuring hierarchies for each component databases to remove structural and meta-data heterogeneities. We do this by resolving division attributes and decomposing composite attributes for all classes participating in integration. Then, in terms of an initial set of assertions built manually and a semantic dictionary built by semantic analyses, an algorithm for identifying subgraph isomorphism will be performed to determine the semantic equivalence relationships for each pair of component databases. The next step is to feed back the results obtained by performing the subgraph isomorphism algorithm to designers to check the validation. Once some results have been validated, they will then be added to the assertion set. The algorithm for identifying subgraph isomorphisms will be performed once again with the newly constructed assertion set. This process repeats until no new results can be obtained.

In the following, we illustrate this process using an example.

Example 5. Consider the following two local object-oriented schemata derived from two university's databases, respectively.

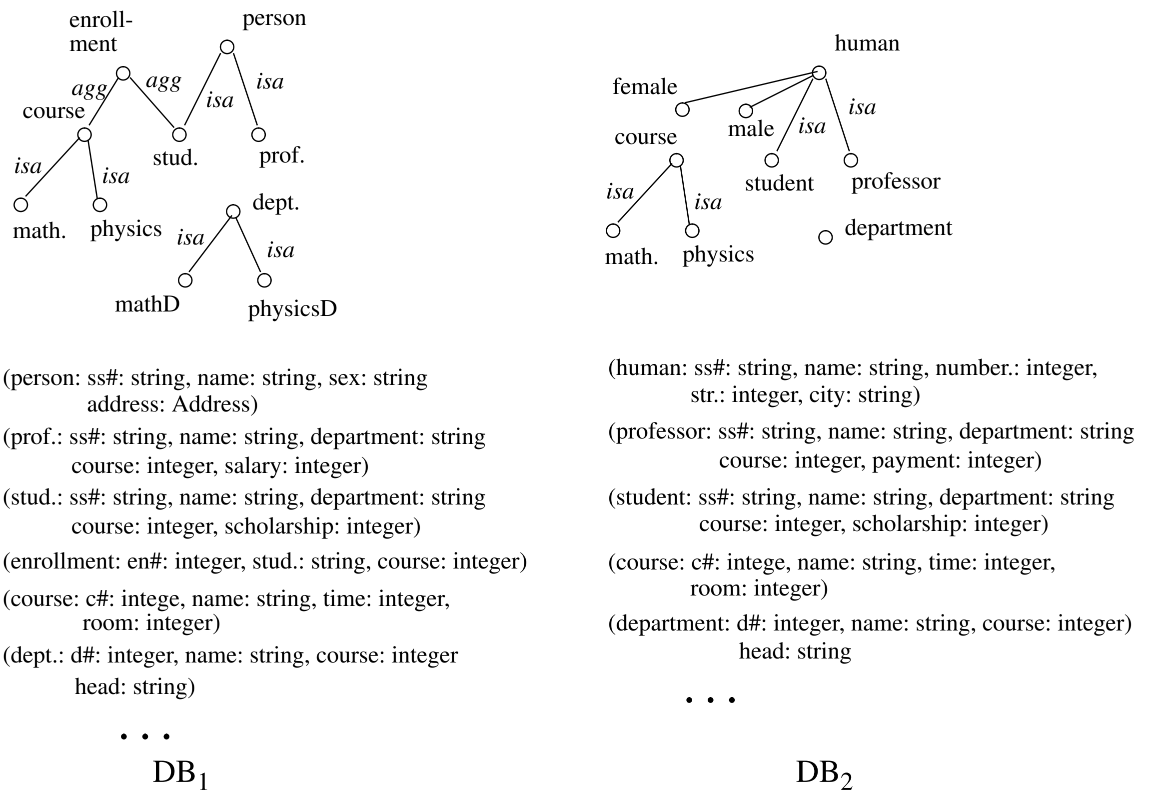

Fig. 11. Two local data schemas

The hierarchy restructuring process will first resolve division attributes, i.e. corresponding to each value of some division attribute, a new subclass will be generated. In this example, for instance, two subclasses "man" and "woman" will be created in terms of division attribute "sex" of *person*. Then, a direct comparison of correspondent concepts (in this case, *man* in DB_1 and *male* in DB_2 as well as *woman* in DB_1 and *female* in DB_2) is possible. The other task of the hierarchy restructuring process is to decompose composite attributes. For example, the composite attribute "address" (in DB_1) will be decomposed into three attributes: "no.", "street" and "city". Then, it is possible to make a comparison of "No." in DB_1 and "Number" in DB_2, "street" in DB_1 and "str." in DB_2 as well as "city" in DB_1 and "city" in DB_2. The goal of the hierarchy restructuring process is to remove the structural and meta-data conflicts such that the mechanism for identifying subgraph isomorphisms can be used. In other words, using the hierarchy restructuring process, a graph for each local schema will be obtained, in which each node corresponds to an "atomic" concept. Obviously, in this case, one may expect to identify semantic equivalences of some pairs of nodes by considering the corresponding relationships among concept nodes, i.e., by considering subgraph isomorphisms. We say that two graphs: *G* and *G'* are isomorphic if there exists a mapping from nodes of *G* to nodes of *G*' such that the corresponding nodes have the same out-degree and in-degree. For the example in question, each local schema (after the hierarchy restructuring process has been applied to it) can be represented using a directed graph (abbr. digraph) as shown in Fig. 12 (here some more information is considered for explanation purpose.) Then, by identifying the subgraph isomorphism, some semantically equivalent pairs can be determined automatically (see the portions marked greyly for illustration).

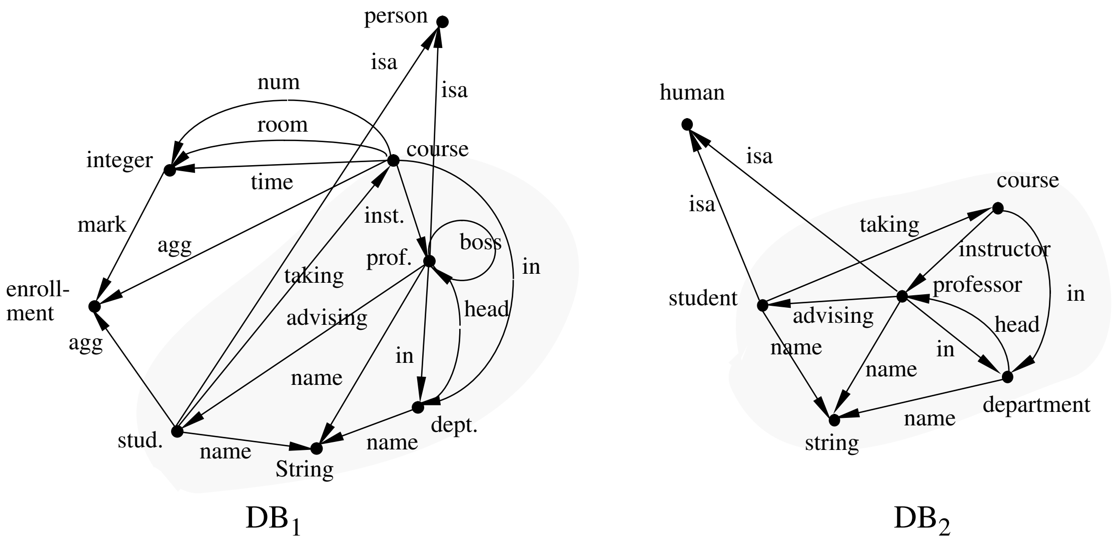

Fig. 12. Illustration for subgraph isomorphism

In general, for any two digraphs, there may be several subgraph isomorphisms if only the out-degree and in-degree are considered. Therefore, in order to confine the search to the relevant potions, some hints are needed. For example, the cardinality constraints on edges can be used to discard some undesired isomorphisms. Note that such constraints can be obtained by analyzing the value distribution of related attributes (normally, four constraints: 1:*n, n*:*m, n*;1 and 1:1 will be considered, corresponding to 1 to *n*, *n* to *m*, *n* to 1 and 1 to 1 constraints.) In addition, an initial assertion set should be provided at the beginning. For example, the assertion: *semantic-similar*(DB_1.Person, DB_2.Human) will confine the search for the equivalent counterparts of *stud.* in DB_1 to {*student, professor*} in DB_2, because only *student* and *professor* are the subclasses of *human* which is semantically similar to *person,* the superclass of *stud.*, while any other classes not belonging to the hierarchy rooted at *human* can not be equivalent to *stud.*. Further, a semantic dictionary will be helpful [BHP94]. For example, a tree structure (in a semantic dictionary) shown in Fig. 13 may exclude the pair (DB1.stud., DB2. professor) due to the fact that the semantic discrepancy between “scholarship” and “payment” is larger than that between “scholarship” and “allowance”.

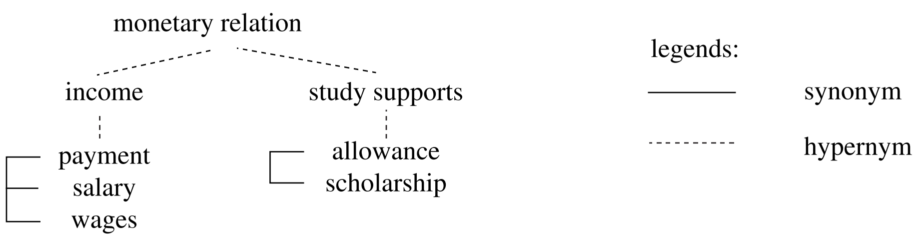

Fig. 13. A tree structure in a semantic dictionary

Of course, in terms of the similarity or the difference between only one pair of attributes one can not determine whether two concepts are semantically equivalent. A global consideration must be made with respect of all the attributes involved. In this case, a method for computing semantic distances between two concepts with different representations has to be developed based on the information available in the semantic dictionary. This is to be investigated in the forthcoming research.

After the subgraph isomorphism, we will get several groups of assertions with each group corresponding to a possible matching. But only one is desired.

Although the initial assertion set and semantic dictionary helps very much for identifying isomorphisms, we need a process to validate the obtained results since we can not provide a complete semantic dictionary to cover all the different representations of the same concepts, which may be employed by different designers. In the process of the knowledge acquisition, several operations are provided to help designers to validate the results obtained by subgraph isomorphism. Among others, the following operations are very important.

- operations for validating the matchings obtained by subgraph isomorphism:

  remove-a-matching(*AG*): remove an assertion group named *AG*.

  accept-a-matching(*AG*): accept an assertion group named *AG*.

  remove-a-assertion(*ass*): remove an assertion named *ass*.

  add-a-assertion(ass): insert an assertion named *ass*.

Note that using the techniques discussed above we can eliminate name conflicts, structural conflicts and meta-data conflicts automatically or semi-automatically. However, the domain conflicts can be removed only by building mapping manually. Here, we do not discuss this issue. The interested reader is referred to [BCG95] for a detailed description of relevant problems.

3.2.2 Rule-based Strategy for Schema Integration

In this subsection, we present two groups of rules. The first rule group is used to control the process for identifying subgraph isomorphisms. The second one is used to do schema integration. In the following discussion, two databases DB_i and DB_j are considered. The mapping M_{ij} between them will be constructed. The mapping M_{ij} is partitioned into two parts: M_{ij}^a and M_{ij}^b, where M_{ij}^a represents the part for concept equivalences while M_{ij}^b represents the equivalences over the elements of attribute domains. In fact, only M_{ij}^a can be built automatically or semi-automatically. M_{ij}^b has to be built manually.

- the first group of rules:

  r_{23}: if *semantic-similar*($\mathrm{DB}_i.x$, $\mathrm{DB}_j.y$) then *insert*((x, y), M_{ij}^a).

In the above rule, *semantic-similar*($\mathrm{DB}_i.x$, $\mathrm{DB}_j.y$) is an assertion used for designers to specify semantic equivalences.

  r_{24}: if *is-a*($class_i$, x), *is-a*($class_j$, y), M_{ij}^a($class_i$, $class_j$) then *relevant*(x, y)

(*Note that for a local database, a rule representing the transitive property has been established:

if *is-a*(x, y), *is-a*(y, z) then *is-a*(x, z)*)

In the following two rule, $AG(\mathrm{DB}_i, \mathrm{DB}_j) = \textit{sub-isomorphism}(\mathrm{DB}_i, \mathrm{DB}_j)$, represents the possible matchings between DB_i and DB_j found by performing the algorithm for identifying isomorphisms. Note that each matching is represented by a group of assertions.

r_{25}: if $GTS \in AG(\mathrm{DB}_i, \mathrm{DB}_j)$, $(x, y) \in GTS$, $\neg relevant(x, y)$ then $delete((x, y), AG)$.

r_{26}: if $GTS \in AG(\mathrm{DB}_i, \mathrm{DB}_j)$, $(x, y) \in GTS$, $relevant(x, y)$ then $insert((x, y), M_{ij}^{a})$.

- the second group of rules:

r_{27}: if $x = class_i.name$, $y = class_j.name$, $M_{ij}^{a}(x, y)$ then $integrate(class_i, class_j)$.

r_{28}: if $M_{ij}^{a}(x, y)$, $relevant(class_i, class_j)$, $\textit{own-attr}(class_i, x)$, $\neg\textit{own-attr}(class_j, y)$
then $subclass(class_i, class_j)$.

r_{29}: if $\textit{own-attr}(class_i, attr)$, $x \in Values(attr)$, $y = class_j.name$, $relevant(class_i, class_j)$, $M_{ij}^{b}(x, y)$
then $subclass(class_i, class_j)$

r_{30}: if $\neg(x = class_i.name, y = class_j.name, M_{ij}^{a}(x, y))$,
$\neg(M_{ij}^{a}(x, y), relevant(class_i, class_j), \textit{own-attr}(class_i, x), \neg\textit{own-attr}(class_j, y))$,
$\neg(\textit{own-attr}(class_i, attr), x \in Values(attr), y = class_j.name,$
$relevant(class_i, class_j), M_{ij}^{b}(x, y))$
then $edge(class_i, class_j)$.

The last rule states that if the conditions appearing in r_{27}, r_{28} and r_{29} are not satisfied, the relationship between two classes: $class_i$ and $class_j$ can not be determined. In this case, we draw an undirected edge from $class_i$ to $class_j$ to indicate that we do not know whether $class_i$ is a subclass of $class_j$ or *vice versa*.

4. Implementation Issues

Based upon the above discussion, we propose the following system architecture.

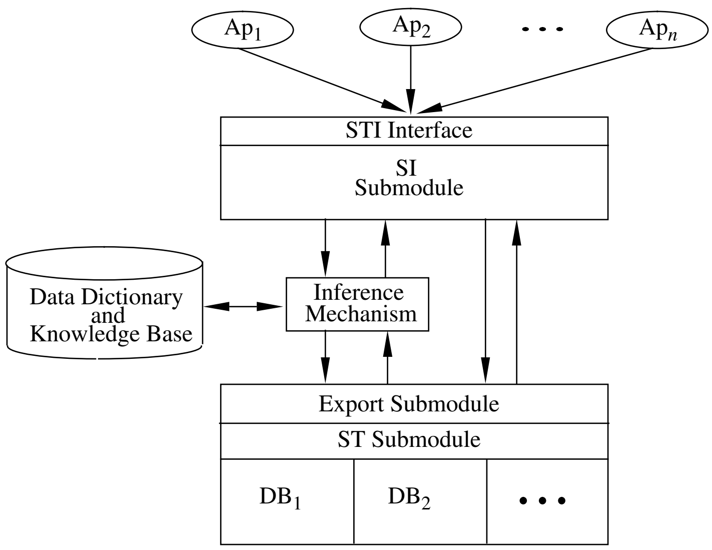

Fig. 14. System Architecture

This architecture consists of six main components:

- Schema Transformation (ST-Submodule),
- Export Control (Export-Submodule),
- Inference-Mechanism,
- Data Dictionary and Knowledge Base,
- Schema Integration (SI-Submodule) and
- Interface (STI-Interface),

which are briefly described below.

1) ST-Submodule

The ST-Submodule provide a set of functions to do local schema analyses such as those for building meta-data structures, those for managing local hierarchies and those for communicating with the inference engine.

2) Export-Submodule

To guarantee the semantic completeness of the integrated schema, two exportation principles will be enforced:

(i) For a local hierarchy, all superclasses of a class should be exported if the class is exported.

(ii) For a local hierarchy, if an object and one of those objects linked with it by aggregation links are exported, then all other objects correlated by the aggregation links have to be in the export schema.

As with the schema transformation and the schema integration, these two principles can be implemented by representing them using rules. In addition, the functions for managing exported schemata are required.

3) Inference-Mechanism

The inference mechanism consists of two parts: inference engine and rule management. The inference engine executes the rule sets for translating and integrating schemata as well as for exporting schemata. The rule management does the tasks such as the storage of rules, the modification of rules and etc.

4) Data Dictionary and Knowledge Base

The meta-data and the mappings from the local schemata to the single schema as well as the mappings among the local schemata are stored in the data dictionary, while the rules explained in the previous two sections are stored in the knowledge base.

5) SI-Submodule

The S-Submodule takes on the tasks for restructuring local hierarchies, for removing conflicts and for controlling the mapping consistence. In addition, as with the ST-Submodule, the communication with the inference engine will be performed.

6) STI-Interface

STI-Interface will provides a set of functions, which makes it possible for users to control the whole process of the schema transformation and the schema integration.

For the schema transformation, the following functions will be implemented:

- traversal of a local hierarchy,
- expansion of a node in a hierarchy,
- exportation of a subgraph and
- starting of the inference engine for transforming schemata.

For the schema integration, the following functions will be implemented:

- unfolding of an integrated class,
- traversal of an integrated hierarchy,
- expansion of a node in an integrated hierarchy,
- consistence checks of the mappings built manually,
- restructuring of a local hierarchy, and
- starting of the inference engine for integrating schemata.

5. Summary and Future Work

In this paper, a systematic method is presented for translating and integrating schemata. The main idea is to build a set of rules to support an automatical process of the schema transformation and a semi-automatical process of the schema integration. The rules for transforming schemata are constructed based on the schema analyses while those for integrating schemata are based on the subgraph isomorphism. In addition, a method for restructuring local schemata and a heuristics based on the linguistic analyses are discussed, which can be used to facilitate and to speed up the identification of subgraph isomorphisms

As the future work, the following issues are interesting:

- Developing a database logic, based on the framework proposed in [CRV93], to formalize relational schemata and database states as well as the concepts of relation classification, which helps to represent the schema transformation process more clearly.

- Building a graph semantics for object-oriented schemata to support the subgraph isomorphism algorithm so that a higher-automatical process of integrating schemata in a heterogeneous environment can be implemented.

- Constructing a powerful data structure to accommodate the semantic dictionary as well as a theory to quantify the semantic distinction of different representations of the same concepts.

- Developing a knowledge acquisition algorithm suitable for gathering knowledge about the semantic heterogeneities of component databases.

References

BCG95 W. Benn, Y. Chen and I. Gringer, “On the Mapping Consistence in a Heterogeneous Information Environment”, in preparation.

BHP94 M.W. Bright, A.R. Hurson and S. Pakzad, “Automated Resolution of Semantic Heterogeneity in Multidatabases”, *ACM Transaction on Database Systems*, Vol. 19, No.

2, June 1994, pp. 212 - 253.

BOT86 Y. Breitbart, P. Olson, and G. Thompsom, “Database integration in a distributed heterogeneous database system,” in *Proc. 2nd IEEE Conf. Data Eng.*, 1986, pp. 301 - 310.

CBS94 R.H.L. Chiang, T.M. Barron, and V.C. Storey, “Reverse engineering of relational databases: extraction of an EER model from a relational database,” *Data & knowledge Engineering*, vol. 12, 1994, pp. 107 - 142.

Ch76 P.P.S. Chen, “The Entity-Relationship Model: Toward a Unified View of Data”, *ACM TODS* 1, No. 1, March 1976.

CH81 J.K. Cheng and T.S. Huang, “A Subgraph Isomorphism Algorithm Using Resolution”, *Pattern Recognition*, Vol. 13. No. 5. pp. 372 - 379, 1981.

CRV93 J.A. Chudziak, H. Rybinsky and J. Vorbach, “Towards a Unifying Logic Formalism for Semantic Data Models”, *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 492 - 507.

CW93 S. Ceri and J. Widom, “Managing Semantic Heterogeneity with Production Rules and Persistent Queues”, in *Proc. 19th Int. VLDB Conference*, Dublin, Ireland, 1993, pp. 108 -119.

DH84 U. Dayal and H.-Y. Hwang, “View definition and generalization for database integration in a multidatabase system,” *IEEE Trans. Software Eng.*, vol. 10, no. 6. pp. 628 - 645, 1984.

DAT87 S.M. Deen, R.R. Amin, and M.C. Taylor, “Data integration in distributed databases,” *IEEE Trans. Software Eng.*, vol. 13, no. 7, 1987.

HLM94 G. Harhalakis, C.P. Lin, L. Mark and P.R. Muro-Medrano, “Implementation of Rule-based Information Systems for Integrated Manufacturing”, *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 892 - 908, Dec. 1994.

Jo93 P. Johannesson, “Using Conceptual Graph Theory to Support Schema Integration”, in *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 283 - 296.

KC93 J-L. Koh and Arbee L.P. Chen, “Integration of Heterogeneous Object schemas”, in *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 297 - 314.

LA86 W. Litwin and A. Abdellatif, “Multidatabase interoperability,” *IEEE Comput. mag.*, vol. 19, No. 12, pp. 10 - 18, 1986.

Ll87 J.W. Lloyd, “*Foundation of Logic Programming*”, Springer-Verlage, Berlin, 1987.

LNE89 J.A. Larson, S.B. Navathe, and R. Elmasri, “A theory of attribute equivalence in databases with application to schema integration,” *IEEE Trans. Software Eng.*, vol. 15, No. 4, pp. 449 - 463, 1989.

LR82 T. Landers and R. Rosenberg, “An overview of multibase,” in *Distributed Databases*, H.J. Schneider, Ed. Amsterdam: North-Holland, 1982.

Ma90a F. Manola, “Object-oriented knowledge bases, Part I,” AI Expert, vol. 10, pp. 26 - 536, Mar. 1990.

Ma90b F. Manola, “Object-oriented knowledge bases, Part II,” AI Expert, vol. 11, pp. 46 - 57, April 1990.

MM90 V.M. Markowitz and J.A. Makowsky, “Identifying extended entity-relationship object structures in relational schemas,” *IEEE Trans. Software Eng.*, vol. 16, No. 8, pp. 777 - 790, 1990.

RPRG94 M.P. Reddy, B.E. Prasad, P.G. Reddy, and A. Gupta, “A methodology for integration of heterogeneous databases,” *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 920 - 933, Dec. 1994.

SK92 W. Sull and R.L. Kashyap, “A Self-organizing knowledge representation schema for extensible heterogeneous information environment,” *IEEE Trans. on Knowledge and Data Engineering*, vol. 4, No. 2, 185 - 191, April 1992.

Te87 M. Templeton *et al.*, “Mermaid - A front end to distributed heterogeneous databases,” *Proc. IEEE*, vol. 75, no. 5, 1987.

TS93 C. Thieme and A. Siebes, “Schema Integration in Object-Oriented Databases”, in: *Proc. 5th. Int. Conf. on Advanced Information Systems Engineering*, Paris, France, June 1993, pp. 54 - 70.

VA95 M.W. Vermeer and P.M. Apers, “Object-oriented views of relational databases incorporating behavior,” in: *Proc. 4th Int. Conf. on Database Systems for Advanced Application (DASFAA’ 95)*, Singapore, April 1995, pp. 26 - 35.