JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 555-591 (2000)

Integrating Heterogeneous OO Schemas

YANGJUN CHEN
IPSI Institute, GMD GmbH
64293 Darmstadt, Germany
E-mail: yangjun@darmstadt.gmd.de

A key problem in providing enterprise-wide information is the integration of databases that have been independently developed. A major requirement is to accommodate heterogeneity and at the same time preserve the autonomy of component databases. This article addresses this problem and presents a strategy to integrate heterogeneous OO schemas. As compared to the existing methodologies, this approach integrates local schemas into a deduction-like global schema. In this way, more semantic relationships of component schemas can be captured, and more complete integration can be obtained. In addition, an efficient algorithm is proposed which can do the integration almost automatically, based on the correspondence assertions supplied by designers. This algorithm is efficient in the sense that the characteristics of assertions are utilized to avoid useless matchings.

Keywords: federated databases, correspondence assertion, schema integration, derivation assertion, integration algorithm

1. INTRODUCTION

With the advent of applications involving increased cooperation among systems, the development of methods for integrating the pre-existing databases has become important. The design of such global database systems must allow unified access to diverse and possibly heterogeneous database systems without subjecting them to conversion or major modifications [4, 7, 11, 18, 33]. One important step in integrating heterogeneous systems is to build a global schema from local ones, which is usually done in two phases: schema transformation and schema integration [1, 30]. By means of schema transformation, a local schema is transformed into an abstract one, e.g., an object-oriented schema [24, 25]. Then, all the local object-oriented schemas are integrated into a global one, thereby removing semantic conflicts caused by different perceptions of the same real world concepts.

To eliminate semantic conflicts among the component databases, a set of correspondence assertions for declaring their semantic relationships has to be constructed by DBAs or by users. Normally, four set relationships between object classes, equivalence, inclusion, intersection, and exclusion, are defined so as to provide knowledge about correspondences that exist among the local schemas [35].

In this article, we will introduce a new assertion, the so-called *derivation assertion*, to accommodate more heterogeneities, which can not be treated using the existing methodologies (see [2, 10, 13, 22, 27, 35]). As an example, consider two local object-oriented schemas, S_1 and S_2. Assume that S_1 contains two classes, *parent* and *brother*, and that S_2 contains one class, *uncle*. A derivation assertion of the form S_1(*parent, brother*) $\rightarrow S_2$ (*uncle*) can specify their corresponding semantic relationship clearly, which can not be

Received December 17, 1998; revised April 1, 1999; accepted June 5, 1999.
Communicated by Arbee L. P. Chen.

established otherwise. We claim that this kind of assertion is necessary for the following reason. Imagine a query concerning *uncle*, submitted to the integrated schema from S_1 and S_2. If the above assertion is not specified, the query evaluation will not take schema S_1 into account; thus, the answers to the query will not be correctly computed in the sense of cooperations. Some more complicated examples will be given later to show that derivation assertions can always be used to handle intricate semantic relationships.

Recently, the problem of schema integration has been addressed extensively. In [35], the four semantic assertions mentioned above were used to declare semantic conflicts between two heterogeneous schemas. In addition, attention was paid to the path correspondence problem there. However, no formal method has been developed for this situation. In [13], another kind of path correspondences was introduced, with which more difficult semantic conflicts (like S_1(*Salary.Person*) ~ S_2(*Salary.Salesmen.Person*), indicating that *Salary* of *Person* from S_1 and *Salary* of *Salesmen* from S_2 are identical) can be tackled. However, no derivation relationships can be dealt with in this way, either. In fact, such a path correspondence can be declared using a combination of equivalence and inclusion. That is, we can declare S_1(*Person*) $\supseteq$ S_2(*Salesmen*), thereby specifying that the attribute *Salary* of *Person* is equivalent to *Salary* of *Salesmen*. In [22], a hyperrelation approach was proposed, with which different attributes of relevant concepts (belonging to different local schemas) can be connected together. Similarly, no derivation problem is considered in this method. In [26], a formal method to describe schema equivalence was developed. It distinguishes between "unconditional" equivalence and the "conditional" equivalence, and also can not be used to declare the derivation relationship. A similar approach was described in [15], where the notion of a database "context" is used to specify conditionally equivalent concepts. In the other methods, such as those proposed in [2, 10, 27], derivation correspondence is not mentioned at all.

To integrate relevant concepts connected by derivation assertions, however, the deductive approach should be employed, and a mechanism to do inferences should be developed to support such more complete global schemas. For this purpose, we simplify the data model proposed in [16] by replacing the concept of "object constructors" with that of "aggregation functions", which is well-defined, extends predicate calculus and enriches the object-oriented model with deduction abilities. In this way, an object-oriented global schema can be equipped with an inference mechanism to capture more semantics. (More importantly, this concept can be easily implemented on the "Ontos" system [28] by using its aggregation functions. "Ontos" does not support the concept of object constructors; our system is built on "Ontos".) As we will see later, the path problem proposed in [34, 35] can also be handled formally in our framework. In addition, autonomy is not violated since the "virtual" inferences (more exactly, rule evaluation; see Appendix B) are performed only at an abstract level and no extra requirements are placed on the local databases.

On the other hand, the integration algorithm has not been studied extensively in previous work concerning federated databases. Although several approaches [33, 35] have been suggested, in terms of the given assertions, to integrate local schemas automatically, no effort has been made to optimize this process. That is, no analysis of correspondence assertions has been done to minimize redundant operations by using their characteristics. Furthermore, in [33, 35], only equivalence and inclusion assertions were considered by an integration process; ways to deal with the other kinds of assertions were not considered. In addition, approaches to integrating aggregation links as well as is-a links (paths) have not

been fully addressed up to now. In fact, these problems are not trivial, and some attention should be paid to them. To this end, we present a new algorithm for perfoming integration almost automatically while taking the assertion characteristics and link integrations into account to achieve high performance.

The remainder of this article is organized as follows. Section 2 introduces the data model used in our system. In section 3, our system architecture is briefly outlined. Section 4 discusses the assertion set, through which the semantic correspondences between local schemas can be defined. In section 5, we give our integration principles. Section 6 is devoted to a integration algorithm. Finally, conclusions are set forth in section 7.

2. OBJECT MODEL

As discussed in the introduction, we need a powerful data model to represent integrated schemas. One way to do this is to accommodate deduction with complex objects and object identities so that local databases can be integrated more fully into a deduction-like object-oriented schema. In the following, we will present an object model with well-defined semantics that equips complex objects with a deduction capability. This object model is used to represent the integrated information in our system.

In fact, our model is a modification of those proposed in [16, 23]. First, in our model, the object identifiers can be referenced only through aggregation functions, rather than as attribute values as suggested in [16]. In addition, we implement the concept of *object constructors* developed in [16] as a combination of aggregation functions in a natural way. Further, due to the above modification, a rule in our model is simply a clause of the first-order logic [20], and no extra complexity is assumed, as compared with the object constructors. On the other hand, the aggregation function is supported in the "Ontos" system [28], which is used as our platform.

In our model, a schema is defined as a set of classes $\boldsymbol{C}$. The type of a class C in $\boldsymbol{C}$, denoted by $type(C)$, is defined as:

$$type(C) = \langle a_1{:}type_1, ..., a_l{:}type_l, Agg_1 \text{ with } cc_1, ..., Agg_k \text{ with } cc_k \rangle,$$

where a_i represents an attribute name, $type_i \in$ {boolean, integer, real, character, string, date} $\cup$ $type(\boldsymbol{C})$ and Agg_j represents an aggregation function: $type(C) \rightarrow type(C')$ $(C, C' \in \boldsymbol{C})$. Further, each aggregation function may be associated with a cardinality constraint $cc_j \in$ {[1:1], [1:*n*], [*m*:1], [*m*:*n*]} (j = 1, ..., k). For instance, a class *Article* may be of the type: *type*(*Article*) = <*title*: string, *author_name*: string, *Published_in*: *Proceedings* with [*m*:1]>, where '*Published_in: Proceedings*' represents an aggregation function (aggregation relationship) *Article* $\rightarrow$ *Proceedings*, specifying the semantic relationship between *domain* class *Article* and *range* class *Proceedings*.

Accordingly, an object (instance) of C is represented as a term (called the *complex O-term*):

$$\langle o : C \mid a_1{:}v_1, ..., a_l{:}v_l, agg_1, ..., agg_k \rangle,$$

where o is the object identifier, C is its class, a_i's are attribute names, v_j's are the corresponding values and each agg_j represents an instance of Agg_j. For example, an instance of class *Article* may be of the form: <id_1: *Article* | *title*: 'improving path-consistence algorithm', *author_name*: 'John', *Published_in*(.) with [*m*:1]>, where *Published-in*(.) takes 'id_1' as the input and returns a value, say 'AI_Tool_91', an object identifier of the class called *Proceedings*, which can be used to visit the corresponding object. In addition, when we refer to an object without considering its attribute values, we simply write <o : C> instead of <o : C | a_1:v_1, ..., a_l:v_l, agg_1, ..., agg_k> for convenience.

The classes in an object database are organized into an inheritance hierarchy. We say that a class C is a subclass of another class C', denoted <C: C'> (or *is_a*(C, C')) and called the *typing O-term*, iff {<o : C>} $\subseteq$ {<o' : C'>}, where {<o : C>} ({<o' : C'>}) represents all the instances belonging to class C (C'). For example, <*student: person*> and <*faculty: employee*> are two types of O-terms.

As for the O-terms (complex O-terms and typing O-terms), we can define derivation relations in a standard way, as implicitly universally quantified statements of the form: γ_1 & γ_2 ...& $\gamma_l \Leftarrow \tau_1$ & τ_2 ...& τ_k, where both γ_i's and τ_k's are O-terms or normal predicates of the first-order logic. (Notice that an O-term can be regarded as a higher order predicate, in which variables for class names and attribute names are allowed). For example, the rule <o_1 : *Empl* | *e_name: x, work_in:* o_2 *Dept*> $\Leftarrow$ <o_2 : *Dept* | *d_name: y, manager*: o_1 *Empl*> states that department managers work in the department they manage. Here, *Empl* and *Dept* are classes, o_1 and o_2 are object variables and *work_in* and *manager* are two aggregation functions. Universal quantifiers over o_1 and o_2 are omitted. As another example, consider the so-called 'interesting pair' problem, which was first addressed in [23] and was further discussed in [16]. The problem is to find the pairs employee-manager such that the employee's department's manager's name coincides with the employee's name, which can be represented (using our method) as follows:

pair(o_1, *manager*(o_2)) $\Leftarrow$ <o_1 : *Empl* | *e_name: x, work_in*: o_2 *Dept*>, *manager*(o_2).
e_name = *x*.

As we can see, this rule is much simpler than that presented in [16], and the semantic ambiguity of [23] is also eliminated.

Alternatively, the first rule above may be written in the following form:

<o_1 : *Empl* | *e_name: x, work_in*: *y*> $\Leftarrow$ <o_2 : *Dept* | *d_name: y, manager: x*>,}

if *work_in* and *manager* are defined as attribute names.

In addition, in a derivation rule, we allow variables for object identifiers, class names, attribute names or aggregation function names appearing in an O-term. In this way, more complicated semantic relationships can be declared. In particular, for integrating schemas of heterogeneous local databases, such a rule can be used to specify complicated schematic discrepancies where an attribute value in one database appears as an attribute name or as a class name in another database (which will be discussed in sections 4 and 5.)

Lastly, we note that here the multi-valued attributes are not considered here for the sake of simplicity. However, it is not difficult to extend the model to accommodate the relevent concepts.

3. SYSTEM ARCHITECTURE

Before we present our principles for doing schema integration, we will present our system architecture, which consists of three-layers, FSM-client, FSM and FSM-agents, as shown in Fig. 1.

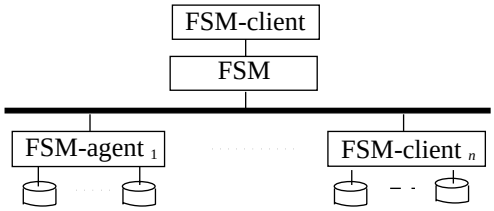

Fig. 1. System architecture.

Here, FSM represents the "Federated System Manager". The task of the FSM-client layer is application management, providing a suite of application tools which enable users and DBAs to access the system. The FSM layer is responsible for merging potentially conflicting local databases and defining global schemas. In addition, centralized management is supported in this layer. The FSM-agents layer corresponds to local system management and addresses all the issues w.r.t. schema translations and exports as well as local transaction and query processing.

With this architecture, each local schema is first transformed into an object-oriented one to remove model conflicts, so that a component database can be integrated into a cooperation more easily (see [6]). However, the data residing in a local database should not be translated, but rather be referenced. Therefore, a datum (in some local database) needs to be uniquely identified in a federated environment. In our system, if a relation is translated into a class, then each of its tuples (of some relation) will be assigned an OID so that the transformed schema will behave just like an object-oriented one. This assignment can be done as follows.

Each component database in our system is installed in some FSM-agent and must be registered in the FSM. Then, if we number the tuples of a relation in the normal way, the OIDs for tuples will be in the following form:

<FSM-agent name>.<database system name>.<database name>.<relation name>.<integer>,

where "." denotes string concatenation. For example, FSM-agent1.informix.PatientDB.patient-records.5 is a legal OID for the fifth tuple of the relation "patient-records" in a database called "PatientDB." Accordingly, each attribute value will be implicitly prefixed with a string of the form

<FSM-agent name>.<database system name>.<database name>.<relation name>.<attribute name>.

Based on such a mechanism, a series of data mappings for each one attribute A of the integrated schema can be constructed, denoted $F^{A}_{DB_i,B}$, (i = 1, 2), with each being used for value correspondences of attribute A and attribute B from the local database DB_i. An $F^{A}_{DB_i,B}$ may be a simple string "default", indicating that all actual values of B form a subset of A; a set of triples of the form (a, b; χ), representing that a of A corresponds to b of B to degree $\chi \in [0, 1]$ (where χ is used to support the fuzzy set concept; see [5] for a detailed discussion); or a simple function of the form $y = f(x)$ (such as $y = 2.54 \cdot x$), where y and x are variables ranging over the domains of A and B, respectively. In the following, for simplicity, we will not discuss data conflicts and will assume that for each pair of classes considered, the relevant data mappings will be established manually and can be accessed by the corresponding methods defined in the root-class (or called the meta-class, pre-defined in the system) for any integrated classes. Corresponding to the above three kinds of data mappings, three accessing methods will be implemented. In addition, additional methods may be associated with the integrated classes manually to establish special value correspondences. Such correspondences can not be made otherwise (see [19]).

In this article, we will discuss only the process of integrating two local object-oriented databases. For the integration of more than two local databases, we adopt a simple "accumulation" strategy as shown in Fig. 2(a), where each S_i stands for a local database schema and each IS_j for an integrated schema of its two child nodes. However, an integration process like that shown in Fig. 2(b) is allowed.

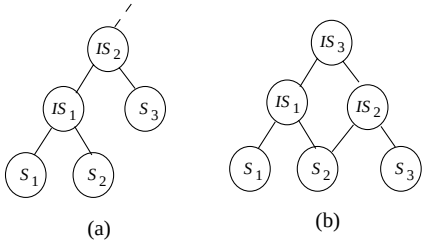

Fig. 2. Integration process.

Using the former strategy, we integrate a single schema into the existing integrated schema at each step. Using the latter, we first contruct a set of integrated schemas by integrating some pairs of local schemas such that all the participating local schemas are considered. Then, for the integrated schemas, we repeat this process until a global schema is generated.

4. ASSERTION SET FOR INTEGRATION

In this section, we will discuss assertions and their classifications.

4.1 Assertion Classification

Assertions for classes

[35] proposed simple and uniform correspondence assertions for the declaration of semantic, descriptive, structural, naming and data correspondences and conflicts. These assertions allow one to declare how the schemas are related but not how to integrate them.

Concretely, four semantic correspondences between two classes were defined in [35], based on the *real-world states* (*RWS*) of object classes. They are equivalence (≡), inclusion (⊇ or ⊆), disjunction (∅) and intersection (∩). Equivalence between two classes means that their extensions (populations) hold the same number of occurrences and that we should be able to relate these occurrences in some way (e.g., with their *object identifiers*). Borrowing terminology from [13], a correspondence assertion can be informally described as follows:

$S_1 \bullet A \equiv S_2 \bullet B$, iff $RWS(A) = RWS(B)$ always holds,
$S_1 \bullet A \subseteq S_2 \bullet B$, iff $RWS(A) \subseteq RWS(B)$ always holds,
$S_1 \bullet A \cap S_2 \bullet B$, iff $RWS(A) \cap RWS(B) \neq \phi$ holds sometimes,
$S_1 \bullet A \varnothing S_2 \bullet B$, iff $RWS(A) \cap RWS(B) = \phi$ always holds,

For example, assuming that *person, book, faculty* and *man* are four classes from S_1, and that *human, publication, student*, and *woman* are another four classes from S_2, the following four assertions can be established to declare their semantic correspondences, respectively: S_1•*person* ≡ S_2•*human*, S_1•*book* ⊆ S_2•*publication*, S_1•*faculty* ∩ S_2•*student*, S_1•*man* ∅ S_2•*woman*.

Observation reveals that the above four assertions are not powerful enough to specify all the semantic relationships of local databases. As mentioned in the introduction of this article, an extra assertion, derivation (→), has to be introduced to capture more semantic conflicts, which can be informally described as follows. Let $A_1, A_2, ..., A_n$ be class names from S_1 and let B be those from S_2. A derivation assertion has the following form:

$S_1(A_1, A_2, ..., A_n) \rightarrow S_2 \bullet B$, iff $P(RWS(A_1), RWS(A_2), ..., RWS(A_n)) \wedge$ "some constraints" holds $\Rightarrow RWS(B)$,

where P is a predicate representing that $RWS(A_1)$, $RWS(A_2)$, ..., and $RWS(A_n)$ exist simultaneously and "⇒" represents the logical implication. The intuition of this assertion is that each occurrence of B can be derived by some operations over a combination of occurrences of $A_1, A_2, ...,$ and A_n. Here, "some constraints" refers to such operations. At a very abstract level, such a constraint can not be specified. But with the help of the declaration of attribute relationships and aggregation function correspondence (see below), we can establish it exactly. For example, assume that *Book* is a class of the type: *type*(*Book*) = <*ISBN*: string, *title*: string, *author*: <*name*: string, *birthday*: date>> from S_1, and that *Author* is a class of the type: *type*(*Author*) = <*name*: string, *birthday*: date, *book*: <*ISBN*: string, *title*: string>> from S_2. We can construct their semantic correspondence using two derivation assertions: S_1•*Book* → S_2•*Author* and S_2•*Author* → S_1•*Book*. At the same time, through a further specification w.r.t. attributes or aggregation functions, these two derivation assertions can be declared more exactly using an attribute correspondence, like S_1•*Book*•*ISBN* ≡ S_2•*Author*•*book*•*ISBN*.

Assertions for attributes, aggregation functions and values

In our system, a second group of correspondence assertions is defined for attributes, such as composed-into ($\alpha(x)$), more-specific-than (β) and those used for classes (i.e., ≡, ⊇ or ⊆, ∩, and ∅). Here, $\alpha(x)$ indicates that the relevant attributes can be combined into a new attribute x, and β is used to declare that one of the two attributes provides more specific information than the other. For example, if *city* and *street-number* are two attributes

belonging, respectively, to two classes being integrated, then *city* α(address) *street-number* states that *city* and *street-number* can be combined into a new attribute called *address*. To explain 'more-specific-than', assume that *category* is an attribute of class *restaurant-1*, and that *cuisine* is an attribute of *restaurant-2*. Then, *cuisine* may contain more specific information than *category* (e.g., the value 'Milan' of *cuisine* is more specific than the value 'Italian' of *category*). Therefore, the *cuisine* β *category* shows this semantic relationship. In addition, we may associate a predicate of the form *att* τ *Cont* with an inclusion ($\supseteq$ or $\subseteq$) to provide more semantic information, where *att* stands for an attribute, *Cont* represents a constant and $\tau \in \{=, <, \leq, >, \geq, \neq\}$. For example, if *type*(*stock*) = <*time*: date, *stock-name*: string, *price*: integer> and *type*(*stock-in-March-April*) = <*stock-name*: string, *price-in-March*: integer, *price-in-Apri*l: integer> are two classes belonging to local databases S_1 and S_2, respectively, then we can use the following two assertions to specify the corresponding semantic relationships:

$S_1 \bullet$ *stock-in-March-April* $\bullet$ *price-in-March* $\subseteq S_2 \bullet$ *stock* $\bullet$ *price with time* = '*March*'
$S_1 \bullet$ *stock-in-March-April* $\bullet$ *price-in-April* $\subseteq S_2 \bullet$ *stock* $\bullet$ *price with time* = '*April*'.

Further, for two classes being integrated, the semantic correspondence among their aggregation functions also has to be specified. To this end, a third group of assertions is utilized, which contains reverse ($\aleph$), equivalence ($\equiv$), inclusion ($\supseteq$ or $\subseteq$), disjunction ($\oslash$) and intersection ($\cap$), where $f \aleph g$ represents that g is a reverse function of f. For example, the aggregation function *spouse* appearing in the class *man* is the reverse of the *spouse* function appearing in the class *woman*. The other assertions are used for the set relationships of the aggregation function's ranges.

In some cases, it is necessary to specify the value correspondence of attributes in the same database. For this purpose, we use '=' and '$\neq$' for single-valued attributes and '$\in$', '$\supseteq$', '$\cap$', '$\oslash$' and '=' for multi-valued attributes. For example, if S_1(*parent, brother*) $\rightarrow S_2$ (*uncle*) is declared as a class correspondence, then we need to further specify the relationship between attribute values to show how *parent* and *brother* can be connected together to form an *uncle* concept. In this case, if *Pssn#* and *brothers* are two attributes belonging to *parent* and *brother*, respectively, then *parent*$\bullet$*Pssn#* $\in$ *brother*$\bullet$*brothers* should be established in the complete description of the assertion w.r.t. *parent*, *brother* and *uncle*. (We will discuss it in the next subsection in more detail.)

Last, due to the fact that in a complex object class an attribute itself may have the type of some other class, the correspondences between elements in different levels of two classes have to be considered. For example, assume that *Book* is a class of the type: *type*(*Book*) = <*ISBN*: string, *title*: string, *author*: <*name*: string, *birthday*: date>> from S_1, and that *Author* is a class of the type: *type*(*Author*) = <*name*: string, *birthday*: date, *book*: <*ISBN*: string, *title*: string>> from S_2. Then, it is desirable to allow a correspondence of the form $S_1$$\bullet$*Book* $\equiv S_2$$\bullet$*Author*$\bullet$*book* (or $S_2$$\bullet$*Author* $\equiv S_1$$\bullet$*Book*$\bullet$*author*) to specify some of their semantic relationships exactly.

We summarize all the assertions in the following Table 1, Table 2 and Table 3.

4.2 Specifying an Assertion

The discussion given in the previous subsection motivates the following definition.

Table 1. Assertions for classer.

$\equiv$	equivalence
$\subseteq, \supseteq$	inclusion
$\cap$	intersection
$\varnothing$	exclusion
$\rightarrow$	derivation

Table 2. Assertion for attributes.

$\equiv$	equivalence
$\subseteq, \supseteq$	inclusion
$\cap$	intersection
$\varnothing$	exclusion
$\alpha(x)$	composed-into
β	more-specified-than

Table 3. Assertions for aggregation functions.

$\equiv$	equivalence
$\subseteq, \supseteq$	inclusion
$\cap$	intersection
$\varnothing$	exclusion
$\aleph$	reverse

Definition 4.1 A *path* w.r.t. a class C is a sequence of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} \ldots \bullet b$, where a_i is an attribute name of C, a_{ij} is an attribute name of $type(a_i)$ (if $type(a_i) \in type(\boldsymbol{C})$, i.e., a_i itself is a class), ..., $a_{ij...hl}$ is an attribute name of $type(a_{ij...h})$ (if $type(a_{ij...h}) \in type(\boldsymbol{C})$), ... and b is of the form $a_{ij...hl...s}$ or "$a_{ij...hl...s}$." If the path is of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} \ldots \bullet a_{ij...hl...s}$, then the attribute values (or the aggregation function's range) of $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} \ldots \bullet a_{ij...hl...s}$ are represented. Otherwise, the path is of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} \ldots \bullet$ "$a_{ij...hl...s}$," used to refer to the attribute name (or the aggregation function name) $a_{ij...hl...s}$ itself.

The following example helps to illustrate this concept.

Example 1. Consider the above *Book* (from S_1) and *Author* (from S_2) classes. The paths *Book•author•birthday* and *Author•book•*"*title*" refer to the attribute values of *birthday* (in the class *Book* from S_1) and the string "*title*" (in the class *Author* from S_2), respectively.

Based on the above discussion, a correspondence assertion ($\theta ::= \equiv \mid \supseteq \mid \varnothing \mid \cap \mid \rightarrow$) between a class A from schema S_1 and a class B from S_2 can be described as shown in Fig. 3, including four kinds of correspondences value correspondence of attributes in S_1, value correspondence of attributes in S_2, attribute correspondence between S_1 and S_2, and agg_function correspondence between S_1 and S_2.

Note that in Fig. 3, not only can the attribute and agg_function correspondences between two local schemas be declared (through "attribute correspondence" and "agg_function correspondence", respectively), but the value correspondences between two attributes in the same schema can also be specified (through "value correspondence of attributes in S_i", i = 1, 2; see also Example 3 for illustration).

$S_1(A_2, A_2, ..., A_n)\ \theta\ S_2 \bullet B$
value correspondence of attributes in S_1:
……
$path_{ik}\ \delta\ path_{jl}$
……
value correspondence of attributes in S_2:
……
attribute correspodence:
……
$S_1 \bullet path_{ms}\ \gamma\ S_2 \bullet path_{Bt}$ with $P_1, ..., P_g$
agg_function correspondence:
$S_1 \bullet path_{uv}\ \lambda\ S_2 \bullet path_{Bw}$

where $\delta ::= = \mid \neq \mid \in \mid \supseteq \mid \varnothing \mid \cap$,
$\gamma ::= \alpha \mid \beta \mid \equiv \mid \varnothing \mid \cap$,
$\lambda ::= \aleph \mid \equiv \mid \supseteq \mid \varnothing \mid \cap$ and
P_j (j=1,...,g) are the predicates of the form: $att\ \tau\ Cont.$

Fig. 3. Description of derivation assertion.

Example 2. In Fig. 4, four correspondence assertions are shown, each specifying a different semantic relationship. Such assertions may be given by users or by DBAs.

$S_1 \bullet person \equiv S_2 \bullet human$
attribute correspondence:
$S_1 \bullet person \bullet ssn\# \equiv S_2 \bullet human \bullet ssn\#$
$S_1 \bullet person \bullet full\_name \equiv S_2 \bullet human \bullet name$
$S_1 \bullet person \bullet city\ o(address)\ S_2 \bullet human \bullet street\text{-}number$
$S_1 \bullet person \bullet interests \supseteq S_2 \bullet human \bullet hobby$

(a)

$S_1 \bullet book \subseteq S_2 \bullet publication$
attribute correspondence:
$S_1 \bullet book \bullet ISBN \equiv S_2 \bullet publication \bullet ISBN$
$S_1 \bullet book \bullet title \equiv S_2 \bullet publication \bullet title$
$S_1 \bullet book \bullet auther \subseteq S_2 \bullet publication \bullet contributor$
agg_function correspondence:
$S_1 \bullet book \bullet published\_by \equiv S_2 \bullet publication \bullet published\_by$

(b)

$S_1 \bullet faculty \cap S_2 \bullet student$
attribute correspondence:
$S_1 \bullet faculty \bullet fssn\# \equiv S_2 \bullet student \bullet ssn\#$
$S_1 \bullet faculty \bullet name \equiv S_2 \bullet student \bullet name$
$S_1 \bullet faculty \bullet income \cap S_2 \bullet student \bullet study\_support$
agg_function correspondence:
$S_1 \bullet faculty \bullet work\_in \equiv S_2 \bullet student \bullet work\_in$

(c)

$S_1 \bullet man \varnothing S_2 \bullet woman$
attribute correspondence:
$S_1 \bullet man \bullet ssn\# \equiv S_2 \bullet woman \bullet ssn\#$
$S_1 \bullet man \bullet name \equiv S_2 \bullet woman \bullet name$
$S_1 \bullet man \bullet occupation \equiv S_2 \bullet woman \bullet occupation$
agg_function correspondence:
$S_1 \bullet man \bullet spouse\ \aleph\ S_2 \bullet woman \bullet spouse$

(d)

Fig. 4. Examples of correspondence assertions.

With the assertion show in Fig. 4(a), we can indicate that the class *person* from S_1 is equivalent to the class *human* from S_2. Three more assertions shown in Figs. 4(b), 4(c) and 4(d) help to explain the usage of "$\subseteq$", "$\cap$" and "$\varnothing$", respectively.

In the following, we will show three further examples to demonstrate how the derivation assertion can be used to specify complicated semantic relationships.

Example 3. Consider the following two schemas for genealogical applications.

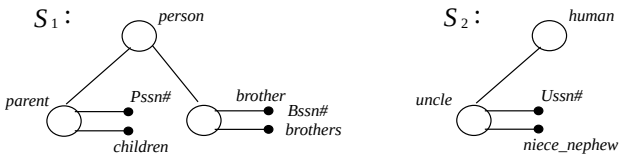

Fig. 5. Two simplified data schemas.

In the Fig. 5, *Pssn#*, *Bssn#* and *Ussn#* are attributes for social security numbers. The existing methodologies, such as those proposed in [9, 12, 13, 17, 19, 30, 35], fail to declare the semantic relationship among *parent*, *brother* (from S_1) and *uncle* (from S_2). Using the derivation assertion, however, we can specify the correspondence among them as follows:

$S_1(parent, brother) \rightarrow S_2 \bullet uncle$
value correspondence of attributes in S_1:
$parent \bullet Pssn\# \in brother \bullet brothers$
value correspondence of attributes in S_2: no constraints

attribute correspodence:

$S_1 \bullet brother \bullet Bssn\# \equiv S_2 \bullet uncle \bullet Ussn\#$
$S_1 \bullet parent \bullet children \supseteq S_2 \bullet uncle \bullet niece\_nephew$

Note that in the above assertion, more complicated semantics are associated with "S_1(*parent, brother*) $\rightarrow$ S_2(*uncle*)". That is, through the value correspondence of the attribute in S_1: *parent•Pssn#* $\in$ *brother•brothers* as well as the attribute correspondences S_1•*brother•Bssn#* $\equiv$ S_2•*uncle•Ussn#* and S_1•*parent•children* $\supseteq$ S_2•*uncle•niece_nephew*, the following semantic relationship can be declared:

parent(x, y), *brother*(z, y) $\rightarrow$ *uncle*(x, z),

where *parent*(x, y) indicates that y is the *parent* of x, *brother*(z, y) indicates that y is the *brother* of z and *uncle*(x, z) indicates that z is the *uncle* of x.

Example 4. As another example, we consider the *Book* and *Author* classes again. In [35], a path correspondence assertion as shown in Fig. 6(a) was be defined to specify the corresponding semantic relationship. Then, for integration purposes, [35] provided a "path integration rule" to integrate these two semantically equivalent paths. The shortcoming of this method is that no formal method can be developed to represent the integration results.

In contrast, using our method, this path equivalence can be represented as two derivation assertions as shown in Figs. 6(b) and (c). Then, based on these two assertions, two inference rules can be exactly constructed, each for an assertion as shown in section 5. In this way, the relevant semantics can be formally established. Furthermore, such rules can be created automatically using our integration principles and integration algorithms, which we will present in the next section.

S_1 (*Book*-author) $\equiv$ S_2(*Author*-book)
(a)

$S_1 \bullet Book \rightarrow S_2 \bullet Author$
value correspondence of attributes in S_1: no constraints
value correspondence of attributes in S_2: no constraints
attribute correspodence:
$S_1 \bullet Book \bullet$ ISBN $\equiv S_2 \bullet Author \bullet book \bullet ISBN$
$S_1 \bullet Book \bullet title \equiv S_2 \bullet Author \bullet book \bullet title$
(b)

$S_2 \bullet Author \rightarrow S_1 \bullet Book$
value correspondence of attributes in S_1: no constraints
value correspondence of attributes in S_2: no constraints
attribute correspodence:
$S_2 \bullet Author \bullet name \equiv S_1 \bullet Book \bullet author \bullet name$
$S_2 \bullet Author \bullet birthday \equiv S_1 \bullet Book \bullet author \bullet birthday$
(c)

Fig. 6. Path equivalence and the corresponding derivation assertions.

Example 5. As a third example of derivation assertion, we consider an extreme situation to show how derivation assertions can be used to specify the semantic correspondence when the so-called *schema conflict* exists [14]. Examine the following two local schemas:

S_1: *type* (car_1) = <*time:* string, *car-name:* string, *price:* integer>,
S_2: *type* (car_2) = <*time:* string, *car-name*$_1$: integer, ..., *car-namen:* integer>.

In S_1, there is a single class, with one instance per month and car, storing the actual price at that time. In S_2, there is a single class, with one instance per month, and one attribute per car, named using the car name and storing its price. The semantic correspondence between them can be established as shown in Fig. 7. (Note that the goal of this example is to show an extreme case in a spectrum of semantic conflicts, which can be declared by establishing derivation rules.)

$S_1 \bullet car_1 \rightarrow S_2 \bullet car_2$

value correspondence of attributes in S_1: no constraints

value correspondence of attributes in S_2: no constraints

attribute correspodence:

$S_1 \bullet car_1 \bullet time \equiv S_2 \bullet car_2 \bullet time$

$S_1 \bullet car_1 \bullet car\text{-}name \cap S_2 \bullet car_2 \bullet \{`car\text{-}name_1', ..., `car\text{-}name_n'\}$

$S_1 \bullet car_1 \bullet price \cap \bigcup_{i=1}^{n} (S_2 \bullet car_2 \bullet car\text{-}name_i)$

(a)

$S_2 \bullet car_2 \rightarrow S_1 \bullet car_1$

value correspondence of attributes in S_2: no constraints

value correspondence of attributes in S_1: no constraints

attribute correspodence:

$S_2 \bullet car_2 \bullet time \equiv S_1 \bullet car_1 \bullet time$

$S_2 \bullet car_2 \bullet car\text{-}name_1 \subseteq S_1 \bullet car_1 \bullet price$

with $S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_1$

... ...

$S_2 \bullet car_2 \bullet car\text{-}name_n \subseteq S_1 \bullet car_1 \bullet price$

with $S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_n$

(b)

Fig. 7. Derivation assertions.

5. INTEGRATION STRATEGIES

Using our object model, we can integrate any two local (object-oriented) databases based on an assertion set given manually. As we will see in Section 6, this integration process can be done almost automatically by executing our integration algorithm with all types of assertions involved.

Once the correspondence assertions between local databases have been stated, integration can be done based on a series of integration principles. In this section, we will discuss these principles in detail.

To simplify our exposition, we assume two default strategies. The first one is that, if for a class, no equivalence assertion is defined, we make a copy of it in the integrated schema, and the relationships with the other integrated classes are built in terms of the corresponding local ones. The second default strategy is that, if no assertion for a pair of attributes is specified, we regard them as being semantically disjointed. Such attributes should be simply accumulated into the corresponding integrated class. Furthermore, we use $IS(S_i \bullet A)$ (resp., $IS(S_i \bullet B \bullet a)$) to denote the integrated version of class A (resp., attribute a of some class B) of the local schema S_i. Thus, if $S_1 \bullet A \equiv S_2 \bullet B$, then $IS(S_1 \bullet A) = IS(S_2 \bullet B)$ = equivalence-class ($S_1 \bullet A$, $S_2 \bullet B$), denoted as IS_{AB}. Similarly, for any two equivalent attributes (aggregation functions) $IS(S_1 \bullet A \bullet a)$ ($IS(S_1 \bullet A \bullet f)$) and $IS(S_2 \bullet B \bullet b)$ ($IS(S_2 \bullet B \bullet g)$), we use IS_{ab} (IS_{fg}) to denote their integrated version. Further, we use S to represent the integrated schema and *value_set*(*att*) for the largest non-null subset of the domain of the attribute *att* w.r.t. the current database state. Finally, "/" represents the set difference operation.

(1) Integration principle for equivalence assertions.

Based on the above notations, the first principle can be defined as follows.

if $S_1 \bullet A \equiv S_2 \bullet B$ **then**
{*insert* (IS_{AB}, S);
for each attribute pair (a, b) with a in A and b in B **do**
{**switch** $a\theta b$ {
case $a\omega b$ with $\omega \in \{\equiv, \supseteq, \subseteq\}$: *insert* ($IS_{ab}$, IS_{AB});
value_set (IS_{ab}): = *value_set* ($S_1 \bullet A \bullet a$) $\cup$ *value_set* ($S_2 \bullet B \bullet b$);
break;
case $a \cap b$: *insert* ($a\_$, IS_{AB}); *value_set* ($a\_$): = *value_set* ($S_1 \bullet A \bullet a$)/
value_set ($S_2 \bullet B \bullet b$); } (*"/" represents set difference*)
insert ($b\_$, IS_{AB}); *value_set*($b\_$): = *value_set* ($S_2 \bullet B \bullet b$)/*value_set*
($S_1 \bullet A \bullet a$);};
insert (a_b, IS_{AB}); *value_set*(a_b): = *value_set* ($S_1 \bullet A \bullet a$)$\cap$*value_set*
($S_2 \bullet B \bullet b$);};
break;
case $a \varnothing b$: *insert* (*IS*($S_1 \bullet A \bullet a$), IS_{AB}); *insert* (*IS* ($S_2 \bullet B \bullet b$), IS_{AB});
value_set (*IS*($S_1 \bullet A \bullet a$)): = *value_set* ($S_1 \bullet A \bullet a$);
value_set (*IS*($S_2 \bullet B \bullet b$)): = *value_set* ($S_2 \bullet B \bullet b$);
break;
case $a\ \alpha\ (z) b$: *insert* (z, IS_{AB}); *value_set*(z): = *cancatenation* ($A \bullet a$, $B \bullet b$);
break;
case $a\ \beta\ b$: *insert* (*IS*($S_1 \bullet A \bullet a$), IS_{AB}); *value_set*(*IS* ($S_1 \bullet A \bullet a$)): =
value_set (a);
break;}}
for each aggregation function pair (f, g) with f in A and g in B **do**
{**switch** $f\ \theta\ g$ {
case $f\ \aleph\ g$: *insert* (*IS*($S_1 \bullet A \bullet f$), IS_{AB}) with the corresponding local *cc*'s;
insert (*IS*($S_2 \bullet B \bullet g$), IS_{AB}) with the corresponding local *cc*'s;
break;
case $f\ \omega\ g$ with $\omega \in \{\equiv, \supseteq, \cap\}$: let C be the range class of $A \bullet f$; let D be
the range class of $B \bullet g$; if $C \equiv D$ or $C \cap D$ then insert (IS_{fg}, IS_{AB}) and
construct its cardinality constraint (*cc*) based on the integration prin-
ciple for is-a and aggregation links (see principle 6);
break;
case $f \varnothing g$: *insert* (*IS*($S_1 \bullet A \bullet f$), IS_{AB}) with the corresponding local *cc*'s;
insert (*IS*($S_2 \bullet B \bullet g$), *ISAB*) with the corresponding local *cc*'s;
break;}}}

where $a\_$, $b\_$, and a_b represent three newly created attributes for the integrated class, and where cancatenation($A \bullet a$, $B \bullet b$) is a function defined as follows:

$$cancatenation(x, y) = \begin{cases} x \cdot y & \text{if there exist } oi_1 \in A \text{ and } oi_2 \in B \text{ such that } oi_1 = oi_2 \\ & \text{(in terms of data mapping), } x = oi_1 \cdot a \text{ and } y = oi_2 \cdot b; \\ Null & \text{otherwise.} \end{cases}$$

In addition, in the description for the aggregation function integration, the corresponding strategies are not specified in detail, but will be treated using Principle 6.

Using this statement, we can integrate two equivalent classes into one. In this way, all the relevant attributes and aggregation functions can be handled in terms of the "attribute and aggregation function correspondence" given in the corresponding assertion.

Example 6. Consider the assertions shown in Fig. 4(a). Let "*person*" be chosen to stand for $IS_{person,human}$, "*ssn#*" for $IS_{pssn\#,hssn\#}$, "*name*" for $IS_{full\_name,name}$ and "*interests*" for $IS_{interests,hobby}$, respectively. Then, the integrated version of S_1(*person*) and S_2(*human*) should be of the following type:

> *type*(*person*) = <*ssn#*: string, *name*: string, *interests*: {string}, address: strings connected with ·>,

where *interests* is a multi-valued attribute, {string} represents a set of strings and "address" is a new attribute name constructed in terms of "S_1•*person*•*city* α(*address*) S_2•*human*•*street-number*" given in assertion Fig. 4(a).

(2) Integration principle for inclusion assertions.

For the inclusion relationship, a simple integration principle can be defined as follows:

if S_1•$A \subseteq S_2$•B **then** insert *is_a*(*IS*(*A*), *IS*(*B*)) into *S*.

Additionally, in order to avoid any redundantly generated i*s_a* link, we extend the above principle to obtain a more general one:

if S_1•$A \subseteq S_2$•B_1, S_1•$A \subseteq S_2$•B_2, ..., S_1•$A \subseteq S_2$•B_n, <B_2: B_1>, ..., <B_n: B_{n-1}> **then** insert *is_a*(*IS*(*A*), *IS*(B_n)) into *S*.

This generalized principle can be pictorially illustrated as shown in Fig. 8.

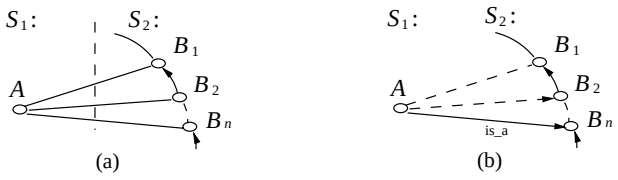

Fig. 8. Illustration for *is_a* link integration.

From this diagram, we can see that for the case shown in Fig. 8(a), only one is-a link "*is_a*(*IS*(*A*), *IS*(B_n))" is generated in terms of the above principle(as shown in Fig. 8(b)), instead of a set of *is_a* links, each one for an inclusion assertion. Note that it is possible that, for some B_i ($1 < i < n$), no assertion between S_1•A and S_2•B_i will be specified for some reason. In this case, a more complex control mechanism is needed to implement the above principle. We will addressed this problem in Subsection 6.1, where an efficient integration algorithm will be discussed in detail.

Example 7. Let *professor* be a class from S_1. Let *human* and *employee* be two classes from S_2. Assume that $S_1 \bullet professor \subseteq S_2 \bullet human$ and $S_1 \bullet professor \subseteq S_2 \bullet employee$ are declared. Then, based on the above principle, only one is-a link "*is_a*(*IS*(*professor*), *IS*(*employee*))" will be generated for the integrated schema if $S_2 \bullet employee \subseteq S_2 \bullet human$ is locally (in S_2) specified.

(3) Integration principle for intersection assertions.

The third integration principle deals with the intersection relationship. For them, we construct rules to do the corresponding integration tasks. First, for each pair of attributes whose semantic relationship is specified using the intersection assertion, we provide a so-called *attribute integration function* (AIF) for the purpose of attribute value conflict resolution. For instance, for assertion shown in fig. 4(c), if $S_1 \bullet faculty \bullet income$ and $S_2 \bullet student \bullet study\_support$ are integrated into an attribute named *income_study_support*, we can compute its attribute values based on the following function:

$$AIF_{i\_s\_s}(x,y) = \begin{cases} \dfrac{x+y}{2} & \text{if there exist } oi_1 \in faculty \text{ and } oi_2 \in student \text{ such that } oi_1 = oi_2 \\ & \text{(in terms of data mapping), } x = oi_1 \cdot income \text{ and } y \\ & = oi_2 \cdot study\_support; \\ Null & \text{otherwise.} \end{cases}$$

Further, for an integrated attribute IS_{attr}, we define a function $re(S_i, IS_{attr})$ used to find its corresponding local version in S_i. (Note that such functions have to be provided by users or DBAs since their semantics entirely depend on individual instants.) Then, the integration principle can be described as follows:

if $S_1 \bullet A \cap S_2 \bullet B$ **then**
{insert $IS(S_1 \bullet A)$ into S;
insert $IS(S_2 \bullet B)$ into S;
insert IS_{AB} into S;
construct {$<x: IS_{AB}> \Leftarrow <x: IS(S_1 \bullet A)>$, $<y: IS(S_2 \bullet B)>$, $y = x$;
$<x: IS_{A\_}> \Leftarrow <x: IS(S_1 \bullet A)>$, $\neg <x: IS_{AB}>$;
$<x: IS_{B\_}> \Leftarrow <x: IS(S_2 \bullet B)>$, $\neg <x: IS_{AB}>$},
for each attribute pair (a, b) with a in A and b in B **do**
{**switch** $a\ \theta\ b$ {
case $a\ \omega\ b$ with $\omega \in \{\equiv, \supseteq, \subseteq\}$:
construct {$insert(IS_{ab}, IS_{AB}) \Leftarrow <\_:IS_{AB}>$; (*Here "_"means "do not care". *)
$x \in value\_set(IS_{ab}) \Leftarrow x \in value\_set\ (re(S_1, IS_{ab})) \vee x \in value\_set\ (re(S_2, IS_{ab}))$};
break;
case $a \cap b$: construct {$insert\ (IS_{ab}, IS_{AB}) \Leftarrow <\_:IS_{AB}>$;
$AIF_{a\_b}(x,y) \in value\_set(IS_{ab}) \Leftarrow x \in value\_set\ (re(S_1, IS_{ab})), y \in value\_set\ (re(S_2, IS_{ab}))$};
break;
case $a \varnothing b$: construct {$insert\ (IS(S_1 \bullet A \bullet a), IS_{AB}) \Leftarrow <\_:IS_{AB}>$;
$insert\ (IS(S_1 \bullet A \bullet b), IS_{AB}) \Leftarrow <\_: IS_{AB}>$;
break;
case $a\alpha\ (z)\ b$: construct {$insert\ (z, IS_{AB}) \Leftarrow <\_:IS_{AB}>$;
$cancatenation(x, y) \in value\_set(z) \Leftarrow x \in value\_set\ (re(S_1, IS_{ab})), y \in value\_set\ (re(S_2, IS_{ab}))$};

break;
case $a\ \beta\ b$: construct {*insert* ($IS(S_1 \bullet A \bullet a)$, IS_{AB}) $\Leftarrow$ <_: IS_{AB}>;
$x \in$ *value_set* ($IS(S_1 \bullet A \bullet a)$) $\Leftarrow$ <_: IS_{AB}>, $x \in$ *value_set* ($re(S_1, IS(S_1 \bullet A \bullet a))$;}
break;}}
for each aggregation function pair (f, g) with f in A and g in B **do**
{**switch** $f\ \theta g$ {
case $f\ \aleph\ g$: report an error; break;
case $f\ \omega\ g$ with $\omega \in \{\equiv, \supseteq, \cap\}$: let C be the range class of $A \bullet f$; let D be the range calss of $B \bullet g$; if $C \equiv D$ or $C \cap D$ then construct
insert (IS_{fg} with *cc*, IS_{AB}) $\Leftarrow$ <_: IS_{AB}>; (**cc* represents its cardinality constraint generated based on Principle 6.*)
break;
case $f\ \varnothing\ g$: construct {*insert* ($IS(S_1 \bullet A \bullet f)$ with *cc'*, IS_{AB}) $\Leftarrow$ <_:IS_{AB}>;
insert ($IS(S_1 \bullet A \bullet g)$ with *cc* ", IS_{AB}) $\Leftarrow$ <_: IS_{AB}>; (*where *cc* ' and *cc*" represents the local cardinality constraints for f and g, respectively.*)
break;}}}}.

In the above description, IS_{AB}, $IS_{A\text{-}}$, and $IS_{B\text{-}}$ represent, respectively, the intersection part of $S_1 \bullet A$ and $S_2 \bullet B$, the part of $S_1 \bullet A$ which does not belong to $S_2 \bullet B$, and the part of $S_2 \bullet B$ which does not belong to $S_1 \bullet A$. Such symbols can be thought of as virtually defined classes since their objects can be referenced only by computing the body classes of rules defining them.

Example 8. Consider the assertion $S_1 \bullet faculty \cap S_2 \bullet student$, for which three rules are generated to define the virtual classes:

$<x: IS_{faculty,\ student}> \Leftarrow <x: IS(S_1 \bullet faculty)>, <y: IS(S_2 \bullet student)>, y = x,$
$<x: IS_{faculty\text{-}}> \Leftarrow <x: IS(S_1 \bullet faculty)>, \neg <x: IS_{faculty,\ student}>,$
$<x: IS_{student\text{-}}> \Leftarrow <x: IS(S_2 \bullet student)>, \neg <x: IS_{faculty,\ student}>.$

In addition, the following rules are established to define virtual attributes and virtual aggregation functions for $IS_{faculty,\ student}$:

insert ($IS_{fssn\#,\ ssn\#}$, $IS_{faculty,\ student}$) $\Leftarrow$ <_: $IS_{faculty,\ student}$>,
$x \in$ *value_set* ($IS_{fssn\#,ssn\#}$) $\Leftarrow x \in$ *value_set*($re(S_1, IS_{fssn\#,ssn\#})$) $\vee\ x \in$ *value_set* ($re(S_2, IS_{fssn\#,ssn\#})$),
insert($IS_{name,name}$, $IS_{faculty,\ student}$) $\Leftarrow$ <_:$IS_{faculty,student}$>,
$x \in$ *value_set* ($IS_{name,name}$) $\Leftarrow x \in$ *value_set* ($re(S_1, IS_{name,name})$) $\vee\ x \in$ *value_set* ($re(S_2, IS_{name,name})$),
insert($IS_{income,\ study\_support}$, $IS_{faculty,\ student}$) $\Leftarrow$ <_:$IS_{faculty,\ student}$>,
$AIF_{i\_s\_s}(x, y) \in$ *value_set* ($IS_{income,\ study\_support}$) $\Leftarrow$
$x \in$ *value_set* ($re(S_1, IS_{income,\ study\_support})$), $y \in$ *value_set* ($re(S_2, IS_{income,\ study\_support})$) .

Note that we do not establish rules for attributes appearing in $IS_{faculty\text{-}}$ and $IS_{student\text{-}}$ since, for them, no integration happens at all.

(4) Integration principle for disjoint assertions.

First, we note that an assertion of the form $S_1 \bullet A\ \varnothing\ S_2 \bullet B$ is meaningful only in the case where there are two object classes A' and B' such that $S_1 \bullet A' \equiv S_2 \bullet B'$ and <*A: A'*> and <*B: B'*> hold. Accordingly, the integration principle for disjoint assertions can be defined as follows:

if $IS(S_1 \bullet A') \equiv IS(S_2 \bullet B')$, $S_1 \bullet A' \supseteq S_1 \bullet A$, $S_2 \bullet B' \supseteq S_2 \bullet B$, $S_1 \bullet A \varnothing S_2 \bullet B$ **then** construct $<x: IS(S_2 \bullet B)> \Leftarrow <x: IS(S_1 \bullet A')>, \neg <x: IS(S_1 \bullet A)>$.

In general, if we have a set of disjoint assertions, $S_1 \bullet A_i \varnothing S_2 \bullet B_j$ ($i = 1, ..., n$; $j = 1, ..., m$) with $<A_i: A>$, $<B_j: B>$ for each i and j, and if $IS(S_1 \bullet A) \equiv IS(S_2 \bullet B)$, then we can establish the following rule to integrate the relevant concepts:

$<x: IS(S_2 \bullet B_1)> \vee ... \vee <x: IS(S_2 \bullet B_m)> \Leftarrow <x: IS(S_1 \bullet A)>, \neg <x: IS(S_1 \bullet A_1)>, ..., \neg <x: IS(S_1 \bullet A_n)>$.

Alternatively, if there exists a specification about reverse aggregation functions, we can rewrite this principle in the following way:

if $S_1 \bullet A \varnothing S_2 \bullet B$ **then** {
  if there exists $S_1 \bullet A \bullet agg_A \aleph S_2 \bullet B \bullet agg_B$ **then**
    construct $<x: IS(S_2 \bullet B)|... IS_{agg_A, agg_B}: y ...> \Leftarrow <y: IS(S_1 \bullet A)|...IS_{agg_A, agg_B}: x...>\}$ and
      $<y: IS(S_1 \bullet A)|...IS_{agg_A, agg_B}: x...>\} \Leftarrow <x: IS(S_2 \bullet B)|...IS_{agg_A, agg_B}: y...>$,
where IS_{agg_A, agg_B} is defined as follows:

$$IS_{agg_A, agg_B}(x) = \begin{cases} agg_A(x) & x \in IS(S_1 \bullet A); \\ agg_B(x) & x \in IS(S_2 \bullet B). \end{cases}$$

(5) Integration principle for derivation assertions.

As with the intersection assertion, for a derivation assertion, several virtual rules are constructed but in a more complex form. To this end, we first partition (manually) one derivation assertion into several smaller ones such that neither the attribute name nor the aggregation function appears more than once in an attribute correspondence or in an aggregation function correspondence. For example, assertions shown in Figs. 7(a) and (b) can be decomposed into the forms shown in Figs. 9 and 10, respectively.

$S_1 \bullet car_1 \rightarrow S_2 \bullet car_2$
- value correspondence of attributes in S_1: no constrains
- value correspondence of attributes in S_2: no constraints
- attribute correspodence:
  - $S_1 \bullet car_1 \bullet time \equiv S_2 \bullet car_2 \bullet time$
  - $S_1 \bullet car_1 \bullet car\text{-}name \cap S_2 \bullet car_2 \bullet \{$'$car\text{-}name_1$'$\}$
  - $S_1 \bullet car_1 \bullet price \cap S_2 \bullet car_2 \bullet car\text{-}name_1$

• • •

$S_1 \bullet car_1 \rightarrow S_2 \bullet car_2$
- value correspondence of attributes in S_1: no constraints
- value correspondence of attributes in S_2: no constraints
- attribute correspodence:
  - $S_1 \bullet car_1 \bullet time \equiv S_2 \bullet car_2 \bullet time$
  - $S_1 \bullet car_1 \bullet car\text{-}name \cap S_2 \bullet car_2 \bullet \{$'$car\text{-}name_n$'$\}$
  - $S_1 \bullet car_1 \bullet price \cap S_2 \bullet car_2 \bullet car\text{-}name_n$

Fig. 9. Decomposed derivation assertions for $S_1 \bullet car_1 \rightarrow S_2 \bullet car_2$.

$S_2 \bullet car_2 \rightarrow S_1 \bullet car_1$
- value correspondence of attributes in S_2: no constrains
- value correspondence of attributes in S_1: no constrains
- attribute correspodence:
  - $S_2 \bullet car_2 \bullet time \equiv S_1 \bullet car_1 \bullet time$
  - $S_2 \bullet car_2 \bullet car\text{-}name_1 \subseteq S_1 \bullet car_1 \bullet price$
    with $S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_1$

(a)

• • •

$S_2 \bullet car_2 \rightarrow S_1 \bullet car_1$
- value correspondence of attributes in S_2: no constraints
- value correspondence of attributes in S_1: no constraints
- attribute correspodence:
  - $S_2 \bullet car_2 \bullet time \equiv S_1 \bullet car_1 \bullet time$
  - $S_2 \bullet car_2 \bullet car\text{-}name_n \subseteq S_1 \bullet car_1 \bullet price$
    with $S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_n$

(b)

Fig. 10. Decomposed derivation assertions for $S_2 \bullet car_2 \rightarrow S_1 \bullet car_1$.

Then, we construct a graph G (called *assertion graph*) for each decomposed derivation assertion of the form $S_1(A_1, A_2, ..., A_n) \rightarrow S_2 \bullet B$. In the graph, there is a node for each "path" refering to an element in some class (see Definition 3.1) and an edge between nodes $path_a$ and $path_b$ iff $path_a$ *rel* $path_b$ with *rel* $\in \{=, \in, \subseteq\}$ is specified. For instance, for the assertion shown in Example 3, we can construct the graph shown in Fig. 11(a).

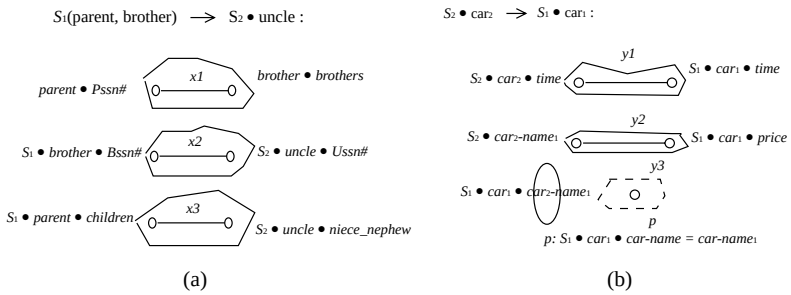

Fig. 11. Assertion graph and hyperedges.

The key step in constructing a virtual rule is to establish the relationships among the O-terms of the rule to be constructed through *variables* as in Artificial Intelligence. (More exactly, specify variables as in *parent*(x, y), *brother*(z, y) $\rightarrow$ *uncle*(x, z).) For this purpose, we mark the nodes of G in the following way:

(1) Each connected subgraph of G is marked using a different variable as shown in Fig. 11(a). For example, the connected subgraph consisting of only one edge (*parent*•*Pssn#*, *brother*•*brothers*) is marked x_1.
(2) For each predicate $p(path_1, ..., path_m)$ appearing in the assertion, we construct a hyperedge $he(p)$, representing the set containing nodes $path_1$, ..., and $path_m$. For example, in the graph associated with the assertion shown in Fig. 10(a), we have a hyperedge for the predicate $S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_1$ (see Fig. 11(b) for an example, in which the hyperedge is marked p.) Note that, here, $car\text{-}name_1$ is a constant, and '= $car\text{-}name_1$' can be considered to be a predicate name. Then, '$S_1 \bullet car_1 \bullet car\text{-}name = car\text{-}name_1$' is a unary predicate.

(We also note that the isolated node "$S_1 \bullet car_1 \bullet car\text{-}name$" is considered to be a connected subgraph, which is marked y_3.)

The goal of the assertion graph is to facilitate the generation of derivation rules. Given a derivation assertion of the form $S_1(A_1, A_2, ..., A_n) \rightarrow S_2 \bullet B$, what we want is to establish a rule of the form $B' \Leftarrow A_1', A_2', ..., A_n', p_1, ..., p_l$, where B' and A_n' ($i = 1, ..., n$) are O-terms and p_j ($j = 1, ..., l$) are normal predicates. Intuitively, B' corresponds to B, A_i' to A_i and p_j to the predicates appearing in the assertion. They are logically linked together through shared attribute variables and object variables.

We first define the following concepts.

Definition 5.1 A *reverse substitution* θ is a finite set of the form $\{c_1/x_1, ..., c_n/x_n\}$, where each x_i is a variable, each c_i is a constant or a variable and $c_1, ..., c_n$ are distinct. Each element c_i/x_i is called a *binding* for c_i.

This concept can be thought of as a reverse operation of the *substitution* concept defined in [29]. In fact, we are performing a process which is just the reverse of rule evaluation in logic programming, by means of which variables are instantiated. But in a reverse substitution, a constant (or a variable) will be replaced with a variable.

Definition 5.2 Let $\theta = \{c_1/x_1, ..., c_n/x_n\}$ be a reverse substitution, and let A be an O-term, a constant or a variable. Then, $A\theta$ is an O-term (or a variable) obtained from A by simultaneously replacing each occurrence of c_i in A with the variable x_i ($i = 1, ..., n$).

As an example, consider O-term $B = <o1: IS(S_2 \bullet uncle) \mid Ussn\#: x, niece\_nephew: y>$. Let $\theta = \{x/x_2, y/x_3\}$. Then, $B\theta = <o1: IS(S_2 \bullet uncle) \mid Ussn\#: x_2, niece\_nephew: x_3>$. Note that in this example, both x and y are typed variables for strings.

If $S = \{A_1, A_2, ..., A_n\}$ is a finite set of O-terms, constants and variables, and if θ is a reverse substitution, then $S\theta$ denotes the set $\{A_1\theta, A_2\theta, ..., A_n\theta\}$.

Definition 5.3 Let $\theta = \{c_1/x_1, ..., c_n/x_n\}$ and $\delta = \{d_1/y_1, ..., d_m/y_m\}$ be reverse substitutions. Then, the composition $\theta\delta$ of θ and δ is the reverse substitution obtained from the set.

$$\{c_1/x_1\delta, ..., c_n/x_n\delta, d_1/y_1, ..., d_m/y_m\}$$

by deleting any binding $c_i/x_i\delta$ for which $c_i = x_i\delta$ and deleting any binding d_j/y_j for which $d_j \in \{c_1, ..., c_n\}$.

For a given derivation assertion, the relevant reverse substitutions can be produced in terms of its assertion graph G. According to the two different variable-marking approaches, we have two methods we can use to produce reverse substitutions:

(i) Consider a connected subgraph G_s of G. Assume that G_s is marked using x_s. Let $\{v_1, ..., v_t\}$ be the node set of G_s. Then, each v_q may be of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet "a_{ij...hl...s}"$ or of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet a_{ij...hl...s}$ (see Definition 4.1). If v_q is of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet "a_{ij...hl...s}"$, then construct a binding $b_q = C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet "a_{ij...hl...s}"/x_s$. If v_q is of the form $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet a_{ij...hl...s}$, then construct a binding $b_q = x/x_s$, where $C \bullet a_i \bullet a_{ij} \bullet a_{ijk} ... \bullet a_{ij...hl...s}: x$ is an attribute descriptor in the corresponding O-term. In this way, we can produce a reverse substitution $\theta_s = \{b_1, ..., b_q, ..., b_t\}$ for G_s.

(ii) Let $he(p)$ be a hyperedge containing nodes $u_1, ..., u_m$, where p is a predicate appearing in the assertion. Let $b_q = c/x$ be the binding generated as described above for u_q ($q = 1, ..., m$). If c is a variable, let b_q' be a/x, where $a:c$ is an attribute descriptor in the corresponding O-term. Otherwise, let b_q' be b_q. Then, the reverse substitution for the predicate p is the composition $\{b_1'\}...\{b_q'\}...\{b_m'\}$. (Note that each $\{b_q'\}$ is a reverse substitution.)

Based on the above discussion, the integration principle for derivation assertion can be summarized as follows:

if $S_1 (A_1, A_2, ..., A_n) \rightarrow S_2 \bullet B$ **then**
{construct an assertion graph G for it;
mark each connected subgraph G_j of G using x_j;

construct a hyperedge for each predicate p_i appearing in the assertion;
for each G_j **do**
generate reverse substitution θ_j;
for each hyperedge $he(p_i)$ **do**
generate reverse substitution δ_i;
generate a derivation rule of the form:
$B\theta_1...\theta_j... \Leftarrow \{A_1, A_2,...,A_n\}\theta_1...\theta_j, \{p_1,...,p_i...\}\ \delta_1...\ \delta_i...$
}

Example 9. Consider the derivation assertion given in Example 3. Its assertion graph can be constructed as shown in Fig. 11(a). Assume that the O-terms of the three classes are B = <*o1*: *IS*(S_2•*uncle*) | *Ussn#: x, niece_nephew: y*>, A_1 = <*o2*: *IS*(S_1•*parent*) | *Pssn#: z, children: u*> and A_2 = <*o3*: *IS*(S_1•*brother*) | *Bssn#: v, brothers: w*>. Then, from that assertion graph, three reverse substitutions can be produced: $\theta_1 = \{z/x_1, w/x_1\}$, $\theta_2 = \{v/x_2, x/x_2\}$ and $\theta_3 = \{u/x_3, y/x_3\}$.

According to the above principle, an inference rule of the following form will be constructed:

$B\theta_1\theta_2\theta_3 \Leftarrow \{A_1, A_2\}\ \theta_1\theta_2\theta_3$
$\Downarrow$
<*o*1: *IS*(S_2 • *uncle*) | *Ussn#*: x_2, *niece_nephew*: x_3> ⇐
<*o*2: *IS*(S_1 • *parent*) | *Pssn#*: x_1, *children*: x_3>, <*o*3: *IS*(S_1 • *brother*) | *Bssn#*: x_2, *brothers*: x_1>.

Example 10. Applying this principle to the decomposed assertions shown in Fig. 10, we can establish a set of rules as follows:

<*o*1: *IS*(S_1 • *car*$_1$) | time: y_1, *car-name*: y_2, *price*: y_3> ⇐ <*o*2: *IS*(S_2 • *car*$_2$) | time: y_1, *car-name*$_1$: y_3>, y_2 = *car-name*$_1$
...
<*o*1: *IS*(S_1 • *car*$_1$) | time: y_1, *car-name*: y_2, *price*: y_3> ⇐ <*o*2: *IS*(S_2 • *car*$_2$) | time: y_1, *car-name*$_n$: y_3>, y_2 = *car-name*$_n$.

We need only to consider assertion shown in Fig. 10(a), for which a graph as shown in Fig. 11(b) can be constructed. Assume that the O-terms of class *car*$_1$ and *car*$_2$ are of the forms B = <*o1*: *IS*(S_1•*car*$_1$) | *time: x, car-name: y, price: z*>, and A = <*o2*: *IS*(S_2•*car*$_2$) | *time: u, car-name*$_1$*: v, ...*>, respectively. Then, from that assertion graph, four reverse substitutions can be generated: $\theta_1 = \{x/y_1, u/y_1\}$, $\theta_2 = \{v/y_2, z/y_2\}$, $\theta_3 = \{y/y_3\}$ and $\delta = \{car\text{-}name/y_3\}$. The first three are produced as its connected subgraphs while the last one is established according to the hyperedge in it.

Let p denote the predicate S_1•*car*$_1$•*car-name* = *car-name*$_1$. Then, the first rule of the above set can be built as follows:

$B\theta_1\theta_2\theta_3 \Leftarrow A\theta_1\theta_2\theta_3, p\delta$
$\Downarrow$
<*o*1: *IS*(S_1 • *car*$_1$)|time: y_1, *car-name*: y_2, *price*: y_3> ⇐ <*o*2: *IS*(S_2 • *car*$_2$)|time: y_1, *car-name*$_1$: y_3>, y_2 = *car-name*$_1$.

Example 11. Given assertions shown in Figs. 6(b) and (c), the following two inference rules can be constructed based on the above principle:

<y: $IS(S_2 \bullet Author)$|*name*: string, *birthday*: date, *book*: <*ISBN*: y_1, *title*: y_2>> ⇐ <x: $IS(S_1 \bullet Book)$| *ISBN*: y_1, *title*: y_2,...>,
<y: $IS(S_1 \bullet Book)$ |..., *author*: <*name*: y_1, *birthday*: y_2>> ⇐ <x: $IS(S_2 \bullet Author)$ |*name*: y_1, *birthday*: y_2, *book*: <*name*: string, *birthday*: date>>.

If the attribute *book* in the class *Author* and the attribute *author* in the class *Book* are defined as aggregation functions, then we can generate the following two simpler rules using the above principle with a bit of modification:

<x: $IS(S_1 \bullet Book)$> ⇐ <y: $IS(S_2 \bullet Author)$ |*book*: x, ...>
<y: $IS(S_2 \bullet Author)$> ⇐ <x: $IS(S_1 \bullet Book)$ |*author*: y,...>.

As in a deductive database, the generated rules should be checked to see whether they are *well-defined*, *safe*, or *domain independent* and *allowed* in the presence of negated body predicates [8].

(6) Integration principle for is-a and aggregation links.

As for an is-a link *is_a*(A, A') in a local schema S_1, let us consider the pair of the form (B, B') in another schema S_2, which satisfies the following conditions:

- B and B' are connected with an is-a path, i.e., a path of the form: $B \leftarrow ... \leftarrow B'$;
- $IS(S_1 \bullet A') \equiv IS(S_2 \bullet B')$ and $IS(S_1 \bullet A) \equiv IS(S_2 \bullet B)$.

Then, if we insert any local is-a link into the integrated schema S (in fact, using our integration algorithm, this simple strategy can be employed), we will have some subgraphs of the forms as shown in Figs. 12(a) and (b) in the integrated schema.

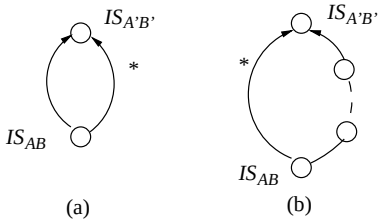

Fig. 12. Redundant is-a links which may exist in an integrated schema.

Therefore, for the subgraph shown in Fig. 12(a), only one of the two links should be inserted while for the subgraph shown in Fig. 12(b), the link indicated by * should not be inserted. This principle can be formally represented as follows:

if $IS(S_1 \bullet A') \equiv IS(S_2 \bullet B')$, $IS(S_1 \bullet A) \equiv IS(S_2 \bullet B)$, (*is_a*($A$, A') ∨ *is_a*(B, B')) **then** insert *is_a*(IS_{AB}, $IS_{A'B'}$) into S.

if $IS(S_1 \bullet A') \equiv IS(S_2 \bullet B')$, $IS(S_1 \bullet A) \equiv IS(S_2 \bullet B)$, $is\_a(A, A')$, $is\_a(B, B_1)$, $is\_a(B_1, B_2)$,..., $is\_a(B_n, B')$ **then**

insert $is\_a(IS_{AB}, IS(B_1))$, $is\_a(IS(B_1), IS(B_2))$,..., $is\_a(IS(B_n), IS_{A'B'})$ into S.

For the aggregation links, we consider only links of the form $agg(B, B')$ and $agg(A, A')$ with $IS(S_1 \bullet A')\ \delta\ IS(S_2 \bullet B')$ and $IS(S_1 \bullet A)\ \delta\ IS(S_2 \bullet B)$ ($\delta \in \{\equiv, \cap\}$). In these cases, the cardinality constraints associated with them should be integrated. To this end, consider the simple constraint lattice shown in Fig. 13(a).

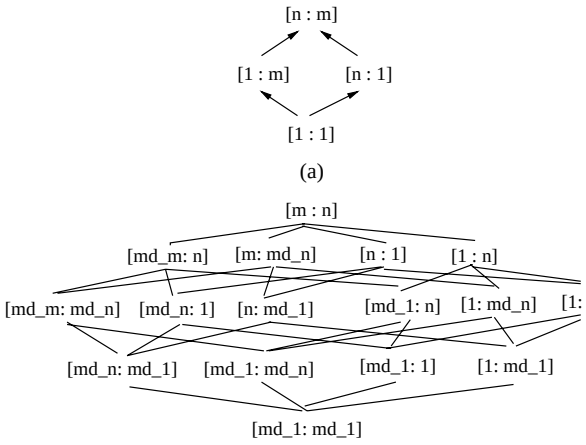

Fig. 13. Constraint lattices.

Based on this constraint lattice, the principle for resolving the constraint conflicts can be described as follows:

if $Agg(A', A)$ with cc_1, $Agg(B', B)$ with cc_2 **then** {insert $Agg(IS_{A'B'}, IS_{A'B'})$ with $lcs(cc_1, cc_2)$ into S}, where $lcs(cc_1, cc_2)$ represents the 'least common super-node' of cc_1 and cc_2.

For example, $[n : m]$ is $lcs([1: m], [n : 1])$ while $[n : 1]$ is $lcs([1: 1], [n : 1])$. In addition, a node is considered to be the least common super-node of itself. This idea can be generalized for more complicated cases. Consider, for example, the cardinality constraint of the so-called "mandatory n to 1", denoted [md_n : 1], which can be used to specify the situation where the participation is total (mandatory) and the mapping is "n to 1". If some constraints like this are involved, we can establish a constraint lattice as shown in Fig. 13 (b) to make the above principle implementable. This lattice reflects a relaxation strategy of the cardinality constraints. If a constraint conflict is encountered, we can resolve it by loosening the local constraints along the lattice from bottom-up, which is least loosened.

6. CONTROLLING THE INTEGRATION PROCESS

In this section, we will discuss our integration algorithm. This algorithm generates almost automatically an integrated schema from two local object-oriented ones based on their correspondence assertions declared by users. Only in very difficult situations is human interference needed. (See the discussion in 6.1.) The algorithm is efficient compared to that proposed in [33] since the semantics of local schemas are used to avoid the need to check useless pairs of concepts. More importantly, a semantically clear integrated schema can be generated by avoiding redundant is-a links. In subsection 6.1, we specify the main part of our algorithm in detail. In subsection 6.2, we discuss how to integrate links. Finally, the correctness and time complexity of the algorithm are considered in subsection 6.3.

6.1 Integration Algorithm

Notice that in our methodology, any local schema will be transformed into an object-oriented one before the integration process is performed. Consequently, a local schema can be viewed as a graph consisting of a set of object classes connected by is-a links, aggregation links or semantic constraints. Accordingly, the input of the following algorithm for controlling the integration process consists of two graphs (with each representing a local schema) and a set of assertions.

In the following, we will not consider the principles for integrating links. We will postpone the relevant discussion concerning link integration to subsection 6.2. In addition, to simplify the explanation, we will assume that each graph has a start node and will be

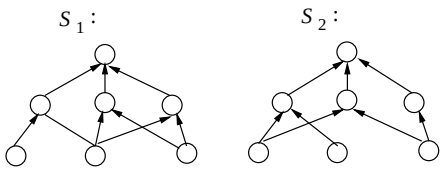

Fig. 14. Illustration of input graphs.

traversed only along is-a links. If an input graph does not have such a node, we construct a virtual one for it, and for each of those nodes which have no parent nodes in the original graph, we draw a meaningless edge from it to the virtual start node. Then, the input graphs of the algorithm can be illustrated as shown in Fig. 14, where each node corresponds to a class and each arc corresponds to an is-a or an aggregation link.

To perform the integration, a naive algorithm will check, for example, all nodes in S_2 for each node in S_1 to see whether some integration should be done on the corresponding concepts. If each local schema contains O(n) nodes, then, including the time spent on the integration operations, the time complexity of a naive algorithm will be larger than O(n^2). Below is a naive algorithm which uses a breadth-first-search but works in a different way from that proposed in [33].

Algorithm *naive_schema_integration*

input: S_1, s_1; S_2, s_2; (*s_1 and s_2 represent the start nodes of S_1 and S_2, respectively.*)
output: S (*S represents the integrated schema.*)

begin
1 Q: = (s_1, s_2); (*Q is a queue structure used to control breadth-first search*)
2 while Q is not empty **do**
3 { (N_1, N_2):= pop(Q); (*take the top element of Q*)
4 let N_{11},...,N_{1k} be child nodes of N_1;
5 let N_{21},...,N_{2m} be child nodes of N_2;
6 put all the pairs of the form (N_{1i}, N_{2j}), (N_1, N_{2j}) or (N_{1i}, N_2) (i=1,..., k; j=1,..., m) into Q;
7 do the integration according to the assertion between N_1 and N_2;
}
end

In this algorithm, a queue structure Q is used to control a breadth-first search of two input graphs. Each element in Q is a pair (N, N'), where N is a node in S_1 and N' a node in S_2. In each iteration, the top pair (a, b) of Q will be checked, and the corresponding integration operation will be performed as described in section 5. Simultaneously, all the pairs of the form (a', b') are put into Q, and are checked in the subsequent iterations, where a' is a or an a's child node and b' is b or a b's child node. We note that this control mechanism is quite different from that proposed in [33]. There, traversal of the two input graphs is completely separated. That is, traversal is performed in one of the two input graphs, say S_1. Then, for each node in S_1, the entire S_2 is searched. In contrast, in the above algorithm, these two processes are integrated together, based on which a lot of optimization (which will be discussed below) can be realized without difficulty. More importantly, the integration principle for inclusion of assertions (for which links will be generated) can be elegantly handled by integrating a depth-first traversal into this algorithm.

First, we will consider the possibility of optimization by analyzing the characteristics of correspondence assertions.

The following observations are important.

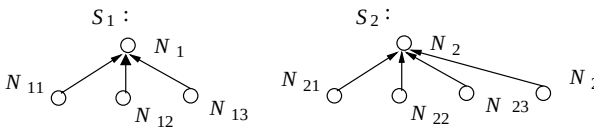

Fig. 15. Two simple input graphs.

1. Consider the two simple graphs shown in Fig. 15.
   If they are the input graphs and $N_1 \equiv N_2$ is an assertion in the assertion set, then the pairs pa_1 = {(N_1, N_{21}), (N_1, N_{22}), (N_1, N_{23}), (N_1, N_{24})} and pa_2 = {(N_{11}, N_2), (N_{12}, N_2), (N_{13}, N_2)} needn't be checked for the following reason. Consider any $N \in \{N_{21}, N_{22}, N_{23}, N_{24}\}$. Then, from $N_1 \equiv N_2$ and *is_a*(N, N_2), we know immediately that *is_a*(*IS*(N), *IS*(N_1)) holds. That is, the semantic correspondences between each pair of pa_1 can be derived. The same analysis applies to all pairs of pa_2.
2. Consider Fig. 15 again. If $N_1 \subseteq N_2$ is specified, then all the pairs of pa_2 needn't be checked either since for any $N \in \{N_{11}, N_{12}, N_{13}\}$ *is_a*(*IS*(N), *IS*(N_2)) can be decided from the relationships $N_1 \subseteq N_2$ and *is_a*(N, N_1) without doing any checking. However, all the pairs of pa_1 have to be checked since in this case, nothing can be inferred from $N_1 \subseteq N_2$

and the other known is-a relationships. Similarly, if $N_1 \supseteq N_2$ is specified, then pa_1 should be but pa_2 needn't be checked. In addition, this observation can be generalized to 'is-a' paths. That is, if $B_1 \leftarrow B_2 \leftarrow ... \leftarrow B_k$ is a path in S_2, $S_1(A) \subseteq S_2(B_k)$ holds and A_1 is a subclass of A, then the relationships $S_1(A_1) \subseteq S_2(B_k)$, ..., $S_1(A_1) \subseteq S_2(B_1)$ can be inferred. Thus, the corresponding pairs needn't be checked.

3. If $N_1 \varnothing N_2$ or a derivation assertion involving N_1, and if N_2 is specified, then we will check neither pa_1 nor pa_2. This is because, in this case, no clear semantic relationships between each pair of pa_1 and pa_2 can be defined. For example, if S_1(man) $\varnothing$ S_2(woman) is defined and S_1(man_student) is declared to be a subclass of S_1(man), then one will tend not to give an assertion to specify the semantic relationship between S_1(man_student) and S_2 (woman). As another example, consider the assertion S_2(*parent, brother*) $\rightarrow$ S_1(*uncle*). One might not specify an assertion between S_2(*old-brother*) and S_1(*uncle*), either. Therefore, ignoring such pairs is reasonable. Of course, it is possible that some assertions between such pairs may be specified. But we believe that ignoring such assertions will not damage the semantics of the integrated schema - a theme in forthcoming research. Alternatively, if such pairs exist, we may, for the purpose of safety, inform the user that something is strange, and ask her or him whether the assertion is correct or a mistake. (This is the only case where user interference is required.)
4. If $N_1 \cap N_2$ is specified, then both pa_1 and pa_2 should be checked. This is because, from $N_1 \cap N_2$, we can not derive anything about the relationship among N_1_(part of N_1 which does not belong to N_2) and the child nodes of N_2. (The same analysis applies to N_2_ and the child nodes of N_1.) Further, in the case where no assertion between N_1 and N_2 is given, the pairs of pa_1 and pa_2 should be checked completely.

Obviously, in order to speed up the above algorithm, lines 6 and 7 in the algorithm have to be changed so that only some of pairs are put into the queue Q based on different correspondence assertions. In addition, a depth-first-search should be integrated to search is-a paths so that the integration principles for inclusions can be fully implemented in a convenient way. Further, to avoid any redundant operations during graph traversal and to implement the integration principle for inclusion assertions uniformly, we introduce a labelling technique and a mechanism for label inheritance used to mark the corresponding pairs, which needn't be checked during graph traversal. See the following Fig. 16.

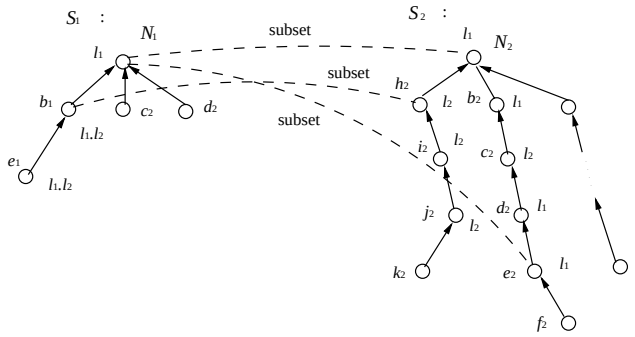

Fig. 16. Illustration of node labelling.

In this diagram, if $N_1 \subseteq N_2$ is specified, we will use a depth-first search to traverse the subgraph rooted at N_2, thereby labelling any path of the form $P = N_2 \leftarrow ... \leftarrow N$ (starting at N_2) with the following properties:

(i) for any node v ($\neq N$) on P, $N_1 \subseteq v$ is specified or no assertion between N_1 and v is defined;
(ii) $N_1 \equiv N$, or $N_1 \subseteq N$ but for any descendent node N_c of N, neither $N_1 \equiv N_c$ nor $N_1 \subseteq N_c$ is defined.

Assume that the path $N_2 \leftarrow b_2 \leftarrow c_2 \leftarrow d_2 \leftarrow e_2$ in the diagram shown in Fig. 16 is one such path. Then, during depth-first traversal, we will mark all the nodes on the path with a label, say l_1, to indicate that these nodes should not be checked against any node in the subgraph rooted at N_1. To do this, N_1 will also be labelled with l_1, and it will further be inherited by all N_1's child nodes. Then, in the subsequent traversal, we can use l_1 to avoid checks of b_1, c_1 and d_1 against any node labelled with l_1 in S_2. Similarly, e_1 will not be checked against these nodes (in S_2) either, in terms of label l_1 inherited from b_1. During graph traversal, some other paths will be marked (with different labels) for the same reason. For example, the path $h_2 \leftarrow i_2 \leftarrow j_2$ shown in the above diagram may be labelled with l_2 if $b_1 \subseteq h_2$, $b_1 \subseteq i_2$, $b_1 \subseteq c_2$ but $b_1 \subseteq k_2$, $\theta \in \{\rightarrow, \varnothing, \cap\}$. Then, the label for b_1 will be changed to $l_1 \cdot l_2$ to indicate that all the nodes in the subgraph rooted at b_1 will not be checked against any node labelled with l_1 or l_2 in S_2. (According to the inheritance mechanism, all the child nodes of b_1 will also possess $l_1 \cdot l_2$.) In general, if a node in S_1 is labelled with $l_1 \cdot l_2 \cdot ... \cdot l_k$, it should not be checked against any node labelled with l_1 or l_2 or ... or l_k in S_2. Similarly, if a node in S_2 is labelled with $l_1' \cdot l_2' \cdot ... \cdot l_j'$ besides any label of the form l_i, it should not be checked against any node labelled with l_1' or l_2' or ... or l_j' in S_1. Below, we will show that each node in S_1 and S_2 will be labelled with a pair of label sequences to implement this duality.

The following algorithm (named *schema_integration*) is mainly based on a combination of breadth-first and depth-first search. By means of breadth-first search, a control similar to *naive_schema_integration* is performed, but with label inheritance and some optimization. By mean of depth-first search, the above labelling technique is implemented. To control the breadth-first search, a queue structure S_b is used to record node pairs of the form (N_1, N_2), whose semantic relationship is to be checked. Further, a stack structure S_d is utilized to control the local depth-first search (in S_1 or in S_2) when a pair with assertion $\subseteq$ (or $\supseteq$) is encountered during breadth-first traversal. (See lines 11 and 18; depth-first search is done by calling *path_labelling*; see below.)

Essentially, the main control is done in lines 3-6 of the following algorithm. Let (s_1, s_2) be the pair being considered. Assume that s_1 has child nodes N_{11}, ..., N_{1k}, and that s_2 has child nodes N_{21}, ..., N_{2m}. Then, all the pairs of the form (N_{1i}, N_{2j}) (i = 1, ..., k; j = 1, ..., m) will be put into S_b for subsequent checks. Further, in terms of the assertion between s_1 and s_2, pairs of the form (s_1, N_{2j}) or (N_{1i}, s_2) (i = 1, ..., k; j = 1, ..., m) may not be put into S_b since their semantic relationships may be derived. (See the discussion above again to understand lines 16, 23, 31, and 33-35.)

Additionally, slightly deviating from the labelling technique discussed above, each node N of S_1 and S_2 is dynamically associated with a pair of label sequences, $<l_1 \cdot ... \cdot l_n, l_1' \cdot ... \cdot l_m'>$, instead of only a label sequence. (We discuss this technique in this way so that the

main idea behind the mechanism can be well understood.) Here, $l_1 \cdot ... \cdot l_n$ are called the labels of N, denoted *labels*(N), representing labels obtained during depth-first search while $l_1' \cdot ... \cdot l_m'$ are the labels obtained through inheritance, called the inherited labels of N and denoted *inherited-labels*(N). Therefore, for a current node N_i, *inherited-labels*(N_i) = $l_1' \cdot ... \cdot l_m'$ indicates that if a node N_j (in another graph) possesses a label pair with *labels*(N_j) = $l_1'' \cdot ... \cdot l_k''$ such that $\{l_1'', ..., l_k''\} \cap \{l_1', ..., l_m'\}$ is not empty, then N_j should not be checked against N_i (see line 7.).

When node N_1 in S_1 meets N_2 in S_2 the first time, and when for them $N_1 \subseteq N_2$ is specified, depth-first-search will be executed to traverse the subgraph rooted at N_2. By this process, any node N with the properties i) and ii) shown above will be labelled, and some integration operations over N_1 and the nodes encountered during traversal will be performed.

Finally, we assume that each node of S_1 and S_2 is initially associated with an empty label pair: < , >.

Algorithm *schema_integration* ((* breadth-first search*)
input: S_1, s_1; S_2, s_2; (*s_1 and s_2 represent the start nodes of S_1 and S_2, respectively.*)
output: S (*S represents the integrated schema.*)
begin
1 l: =0; (*l is used to label paths during depth-first search.*)
2 S_b: = (s_1, s_2);
3 while S_b is not empty **do**
4 {(N_1, N_2): =pop(S_b);
5 let N_{11}, ..., N_{1k} be child nodes of N_1; let N_{21}, ..., N_{2m} be child nodes of N_2;
6 put all the pairs of the form (N_{1i}, N_{2j}) (i=1, ..., k; j=1, ...,m) into S_b;
7 **if** *inherited-labels* (N_1) $\cap$ *labels* (N_2) = $\phi \wedge$ *labels* (N_1) $\cap$ *inherited-labels* (N_2) = ϕ **then**
8 {**switch** (N_1 θ N_2) {
9 **case** $N_1 \equiv N_2$: put N = merging (N_1, N_2) into S;
10 let M_{11}, ..., M_{1i} be brother nodes of N_1;
let M_{21},..., M_{2j} be brother nodes of N_2;
remove the pairs of the form (N_1, M_{2j}) or (M_{1i}, N_2) from S_b;
break;
11 **case** $N_1 \subseteq N_2$: call *path_labelling* (N_1, S_2, N_2, l); (*Algorithm *path_labelling* is given below.*)
12 let l' be the returned label of *path_labelling*;
13 *inherited-labels* (N_1): = *labels* (N_1). l';
14 **for** each child node N_{1i} of N_1 **do**
15 *inherited-labels* (N_{1i}):= *inherited-labels* (N_1);
16 put all the pairs of the form (N_1, N_{2i}) into S_b;
17 break;
18 **case** $N_1 \supseteq N_2$: call *path_labelling* (N_2, S_1, N_1, l);
19 let l' be the returned label of *path_labelling*;
20 *inherited-labels* (N_2):= *inherited-labels* (N_2). l';
21 **for** each child node N_{2i} of N_1 **do**
22 *inherited-labels* (N_{2j}):= *inherited-labels* (N_2);
23 put all the pairs of the form (N_{1i}, N_2) into S_b;

```
24              break;
25          case N1 ∅ N2: construct the corresponding rules in terms of Principle 4;
26          case N1 and N2 involved in a derivation assertion:
27              construct the corresponding rules in terms of Principle 5;
28              break;
29          case N1 ∩ N2: insert IS(N1) and IS(N2) into S;
30              construct rules defining IS(N1_), IS(N2_) and IS_{N1N2} based Principle 3;
31              put all the pairs of the form (N1, N2j) and (N1i, N2) into Sb;
32              break
33          default: put all the pairs of the form (N1, N2j) and (N1i, N2) into Sb;}}
34      else if inherited-labels(N1) ∩ lables (N2) ≠ ϕ
                                    then put all the pairs of the form (N1, N2j) into Sb;
35                                  else put all the pairs of the form (N1i, N2) into Sb;
    }
end
```

In the above algorithm, one of seven cases, $N_1 \equiv N_2$, $N_1 \subseteq N_2$, $N_2 \subseteq N_1$, $N_1 \varnothing N_2$, "N_1 and N_2 involved in a derivation assertion", $N_1 \cap N_2$, and "no assertion specified between N_1 and N_2" is handled in each step of breadth-first traversal. If $N_1 \equiv N_2$, then only the pairs of the form (N_{1i}, N_{2j}) are put into S_b. In addition, all the pairs of the form (N_1, M_{2j}) or (M_{1i}, N_2) (where M_{1i} and M_{2j} represent the brother nodes of N_1 and N_2, respectively) should be removed from S_b since the relationship between N_1 (N_2) and M_{2j} (M_{1i}) is the same as the local relationship between N_2 (N_1) and M_{2j} (M_{1i}). If $N_1 \subseteq N_2$, then depth-first search (by calling *path_labelling*; see below) will be performed over a subgraph of S_2 (rooted at N_2), in which the label sequence of each node P will be lengthened with a new label (see line 1 of *path_labelling*) if it is reached by mean of depth-first traversal and satisfies $N_1 \subseteq P$. Furthermore, the corresponding integration operation will be performed whenever the appropriate node of S_2 is encountered during depth-first traversal. (See lines 10-12, 13-17, 19-25 of *path_labelling*.) The new label will be returned from *path_labelling*, and N_1's inherited label will also be lengthened with it. Then, this new inherited label will be transferred to all its child nodes to execute label inheritance.

A similar description applies for the case $N_2 \subseteq N_1$. In the fourth, fifth and sixth cases, the corresponding integration operations will be performed without any optimization. If no assertion is specified for N_1 and N_2, nothing will be done; traversal continues. Finally, we note that the labels are checked in lines 7 and 34 to avoid any useless matching.

As mentioned earlier, depth-first-search should be employed to tackle the integration principle for the inclusion assertion, by means of which the is-a paths have to be searched ahead of the breadth-first-search process. On the one hand, some integrated is-a links should be created according to this principle. On the other hand, we should avoid any redundant traversal caused by the combination of these two orthogonal search strategies. To this end, we label the paths with the properties discussed above during depth-first search. Then, this label will be returned to the breadth-first-search process to avoid useless checks. In addition, to cope with cases where no assertions are defined at all for some nodes (classes), we denote these nodes using a special symbol, e.g., '$*$', during local depth-first-search. Then, based on a backtracking mechanism, the integration principle for the inclusion assertion can be implemented as discussed in the previous section.

Algorithm *path_labelling* (*depth-first search*)
input: N_1, T, N_2, l
output: l (*l will be changed during the algorithm and used as the otput.*)
begin
1 l:= l+1;
2 S_d:= N_2;
3 while S_d is not empty **do**
4 {V:= pop(S_d);
5 **switch** (N_1 θ V) {
6 **case** $N_1 \subseteq V$: *labels* (V): = *labels* (V). l;
7 let V_1, ..., V_k be child nodes of V;
8 put all the nodes V_i (i = 1,..., k) into S_d; (*go deeper into the graph*)
9 break;
10 **case** $N_1 \equiv V$: *labels* (V):= *lables* (V).l;
11 put N = merging (N_1, V) into S;
12 break; (* the remaining part of the current path will no longer be searched.*)
13 **case** θ ∈ {→, ∅, ⊇}: let $U_k \leftarrow U_{k-1}$*← U_{k-2}*...← U_1* ← V) be an is-a path in T such that
14 all nodes but U_k and V on it are denoted by*;
15 **for** all U_j* (j = 1,..., k-1) **do**
16 *labels* (U_j*): = *labels* (U_j*)/l; (*undo the invalid labels; i.e., in the subsequent traversal, such nodes should not be prevented from being checked against any node in the subgrah rooted at N_1.*)
17 insert *is_a* (*IS*(N_1), *IS*(U_k)) into S; (*$N_1 \subseteq U_k$ must be specified; based on Fig. 8(b), an is-a link should be generated.*)
18 break;
19 **default**: mark V with*;
20 **if** V has child nodes, then put all child nodes into S_d (*go deeper into the graph*)
21 **else** {let $U_k \leftarrow U_{k-1}$* ←U_{k-2}*...←U_1*←V) be an is-a path in T such that all
22 nodes but U_k and V on it are denoted by*;
23 **for** all U_j* (j = 1,..., k-1) **do**
24 *lables* (U_j*): = *lables* (U_j*)/l; (*undo the invalid labels*)
25 insert *is_a* (*IS*(N_1), *IS*(U_k)) into S;}}
}
end

In the above algorithm, any path rooted at N_2 is traversed until a node N with one of the following properties is encountered: (1) $N_1 \equiv N$; (2) N_1 θ N with θ ∈ {→, ∅, ∩, ⊇}; (3) N is an end node. In the first case (see lines 10-12), we generate an integrated class for N_1 and N and the remainder of the corresponding path will not be searched. In the second and third cases (see lines 13-18 and lines 19-25, respectively), we backtrack (along the corresponding path) to the first node N' which is not denoted by *. In terms of the characteristics of the algorithm, we know that $N_1 \subseteq N$' must be specified. Then, an is-a link will be created just as Principle 2. In Appendix A, we trace a sample integration process to demonstrate how the algorithms work.

6.2 More About Link Integration

In the algorithm presented above, link integration is not considered at all. In our implementation, each local link is implicitly taken as a link in the integrated schema in this procedure. Consequently, in an integrated schema generated by the above algorithms, some subgraphs as shown in Fig. 12 may exist. Based on Principle 6, the edge denoted by * in such graphs should be removed. Further, for the aggregation links, we should replace the corresponding semantic constraints with their 'least common super-constraint' w.r.t. the constraint lattice shown in Fig. 13.

To address these problems, we modify the above algorithms a bit as follows.

First, we mark each node *A* (using a special symbol), for which an equivalence assertion is specified. Then, we check its father node *B* to see whether it has already been marked. If so, we further check *B*'s counterpart *C* to see whether *A*'s counterpart *D* is a child node of *C*. (See Fig. 17(a) for an illustration.) In this way, we can easily identify the subgraphs of the form shown in Fig. 12(a).

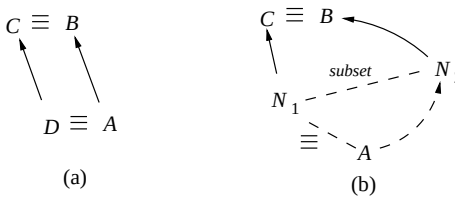

Fig. 17. Illustration of link integration.

In the case of is-a links, we remove one of the two edges. Otherwise, we integrate the two aggregation links into one link with the semantic constraint being its *lcs*. Of course, the aggregation links to be integrated must have similar meaning, and their relationship must have been declared. (See Principle 6.)

The same technique applies to subgraphs of the form shown in Fig. 12(b). Consider algorithm *path_labelling* once again. If the returned value of this algorithm is not a label, but rather a pair consisting of a label and a node *A*, for which an equivalence assertion is specified (see lines 9-11 in *path_labelling*), and if the current call of *path_labelling* is of the form *path_labelling* (N_1, S_2, N_2, l), then we need to check N_2's father node *B* to see whether it has been marked. If it has been marked, then we check *B*'s counterpart *C* to see whether N_1 is a child node of *C*. (See Fig. 17(b) for an illustration). In this way, this kind of subgraph can also be identified without difficulty.

6.3 Correctness and Time Complexity of the Refined Integration Algorithm

The correctness of the refined integration algorithm can be directly derived from the discussion given just before the description of that algorithm in subsection 6.2. This is because the difference between the algorithms *naive-schema-integration* and *schema-integration* lies only in the fact that by the latter some pairs are not checked. However, based

on that discussion, the removed pairs really do not need to be considered, and their semantic relationships are included eventually in the integrated schema; or all the pairs are checked explicitly or implicitly using *schema-integration*.

We distinguish among three kinds of pairs (N_1, N_2), where $N_1 \in S_1$ and $N_2 \in S_2$:

(1) those pairs that are really checked during the execution of *schema-integration*;
(2) those pairs of the form (N_1, N_2) with the following properties:
    (i) there exists an $N \in S_1$ ($N \in S_2$) such that N is an ancestor of N_1 (N_2) and
    (ii) $N \equiv N_2$ ($N \equiv N_1$) holds;
(3) those pairs (N_1, N_2) with N_1 being labelled using $<l_1 \cdot ... \cdot l_n, l_1' \cdot ... \cdot l_m'>$ and N_2 being labelled using $<l_1'' \cdot ... \cdot l_i'', l_1''' \cdot ... \cdot l_k'''>$ such that $\{l_1, ..., l_n\} \cap \{l_1''', ..., l_k'''\} = \phi$ or $\{l_1' \cdot ... \cdot l_m'\} \cap \{l_1'' \cdot ... \cdot l_i''\} = \phi$.

From lines 9-10 of *schema-integration*, we can see that the second kind of pair will not be checked. However, it really needn't be checked since the semantic relationship between each pair of this kind can be directly derived. From line 7, if $\{l_1, ..., l_n\} \cap \{l_1''', ..., l_k'''\}$ is not empty or $\{l_1' \cdot ... \cdot l_m'\} \cap \{l_1'' \cdot ... \cdot l_i''\}$ is not empty, then the corresponding pair (N_1, N_2) will not be checked, either. This is because their semantic relationship can also be inferred according to the integration principle for inclusion and the inclusion relationship between some of their ancestors. From *schema-integration*, we can see that no other pairs are ignored. Therefore, we claim that all the pairs are checked explicitly or implicitly, which guarantees the correctness of the algorithm.

To simplify the time complexity analysis, we assume a simple setting where both S_1 and S_2 have tree structures and each concept from S_1 has exactly one equivalent counterpart from S_2. Further, we assume that both S_1 and S_2 have the same height h. We denote the average number of pairs (N_1, N_2) ($N_1 \in S_1$, $N_2 \in S_2$) checked during the process as Ω_h. Let d be the average degree (the number of edges incident to a node) of the tree corresponding to S_1. Then, we have the following recurrence relations:

$$\Omega_h = \frac{1}{2} \cdot (1 + d \cdot \Omega_{h-1} + \frac{n}{d^0}),$$
$$\Omega_{h-1} = \frac{1}{2} \cdot (1 + d \cdot \Omega_{h-2} + \frac{n}{d^1}),$$
……
$$\Omega_{h-i} = \frac{1}{2} \cdot (1 + d \cdot \Omega_{h-i} + \frac{n}{d^i}),$$
……

The first recurrence relation is obtained as follows. We will consider two "extreme" cases. The first case is where the roots of S_1 and S_2 match. In this case, the average number of the pairs to be checked should be $1 + d \cdot \Omega_{h-1}$. The second case is where the root of S_1 matches some leaf node of S_2. In this case, since the rest of the nodes of S_1 needn't be checked against S_2, the total number of pairs checked is on the order O(n). Therefore, the average number of the pairs to be checked is $\frac{1}{2} \cdot (1 + d \cdot \Omega_{h-1} + \frac{n}{d^0})$. Expanding the above recurrence relations, we can derive Ω_h = O(n).

7. CONCLUSIONS

In this paper, a new strategy is presented for integrating local OO schemas into a deduction-like one. On the one hand, a new correspondence assertion (derivation assertion) has been introduced to accommodate more heterogeneities which can not be handled by any existing method. On the other hand, a more powerful object model has been discussed, which enriches the object-oriented data model with deductive abilities. In this way, not only can the derivation assertion be tackled without difficulty, but some other complex semantic relationships, such as the 'path' concept proposed in [35] and role-consistency addressed in [32], can be treated uniformly in the same framework. Further, an efficient algorithm has been developed and analyzed in detail, which can be used to perform integration almost automatically if the correspondence assertions between local schemas are given. Using this algorithm, the characteristics of each assertion can be utilized to speed-up the computation, and the is-a paths can be taken into account to generate a semantically clearer global schema.

At present, the behaviors of some classes can not be inferred when their parent classes are declared with exclusion or derivation assertion. This is not only related to optimization of the integration algorithm, but also to semantic analysis. Investigation into this issue may lead to the discovery of new correspondence assertions. In addition, the efficient evaluation of rules defined across several databases is another interesting topic which is somewhat different from the strategies developed for the rules in a single one. Such rules may also be used to support automatic decomposition and translation of queries submitted to an integrated schema.

REFERENCES

1. W. Benn, Y. Chen, and I. Gringer, "A rule-based strategy for schema integration in a heterogeneous information environment," Internal Report, CSR-96-1, the Computer Science Department, Technic University of Chemnitz-Zwickau, Germany, 1996.
2. E. Bertino, "The integration of heterogeneous data management systems: approaches based on the object-oriented paradigm," in [3], Chapter 7, pp. 251-269.
3. O. Bukhres and A. K. Elmagarmid (eds): *Object-Oriented Multidatabase Systems: a Solution for Advanced Applications*, Prentice-Hall, 1996.
4. Y. Breitbart, P. Olson, and G. Thompsom, "Database integration in a distributed heterogeneous database system," in *Proceedings 2nd IEEE Conference Data Engineering*, 1986, pp. 301-310.
5. Y. Chen and W. Benn, "On the query transalation in federated relation databases," in *Proceedings of the 7th International DEXA Conference and Workshop on Database and Expert Systems Application*, 1996, pp. 491-498.
6. Y. Chen and W. Benn, "A rule-based strategy for transforming relational schemas into OO schemas," in *Proceedings of the First International Cooperative Database Systems for Advanced Applications*, 1996, pp. 85-88.
7. S. Ceri and J. Widom, "Managing semantic heterogeneity with production rules and persistent queues," in *Proceedings 19th International Conference on Very Large Data Base*, 1993, pp. 108-119.
8. S. K. Das, *Deductive Databases and Logic Programming*, Addison-Wesley, New York,

1992.
9. Y. Dupont, "Resolving fragmentation conflicts schema integration," in *Proceedings 13th International Conference on the Entity-Relationship Approach*, 1994, pp. 513-532.
10. M. Garcia-Solaco, F. Saltor, and M. Castellanos, "Semantic heterogeneity in multidatabase systems," in [3], Chapter 5, pp. 129-202.
11. G. Harhalakis, C. P. Lin, L. Mark, and P. R. Muro-Medrano, "Implementation of rule-based information systems for integrated manufacturing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, , 1994, 892-908.
12. J-L. Koh and Arbee L. P. Chen, "Integration of heterogeneous object schemas," *in Proceedings 12th International Conference on the Entity-Relationship Approach*, 1993, pp. 297-314.
13. W. Klas, P. Fankhauser, P. Muth, T. Rakow, and E. J. Neuhold, "Database integration using the open object-oriented database system VODAK," in [3], Chapter 14, 1996, pp. 472-532.
14. R. Krishnamurthy, W. Litwin and W. kent, "Language features for interoperability of databases with schematic discrepancies," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1991, pp. 40-49.
15. V. Kashyap and A. Sheth, "Semantic and semantic similarities between database objects: a context-based approach," *VLDB Journal*, Vol. 5, No. 4, 1996, pp. 276-304.
16. M. Kifer and J. Wu, "A logic for programming with complex objects," *Interational Journal of Computer and System Sciences*, Vol. 47, No. 1, 1993, pp.77-120.
17. P. Johannesson, "Using conceptual graph theory to support schema integration," in *Proceedings 12th Interational Conference on the Entity-Relationship Approach*, 1993, pp. 283-296.
18. W. Litwin and A. Abdellatif, "Multidatabase interoperability," *IEEE Computer Maggazine*, Vol. 19, No. 12, 1986, pp. 10-18.
19. E. Lim, S. Hwang, J. Srivastava, D. Clements, and M. Ganesh, "Myriad: design and implementation of a federated database prototype," *Software-Practice and Experience*, Vol. 25, No. 5, 1995, pp. 533-562.
20. J. W. Lloyd, *Foundation of Logic Programming*, Springer-Verlage, Berlin, 1987.
21. J. A. Larson, S.B. Navathe, and R. Elmasri, "A theory of attribute equivalence in databases with application to schema integration," *IEEE Transactions Software Engineering*, Vol. 15, No. 4, 1989, pp. 449-463.
22. C. Lee and M. Wu, "A hyperrelational approach to integration and manipulation of data in multidatabase systems," *Interational Journal of Cooperative Information Systems*, Vol. 5. No. 4, 1996, pp. 251-269.
23. D. Maier, "A logic for objects,"' in *Proceedings Workshop on Foundations of Deductive Databases and Logic Programmin*g, 1986, pp. 424-433.
24. F. Manola, "Object-oriented knowledge bases, Part I," *AI Expert*, Vol. 10, 1990, pp. 26-36.
25. F. Manola, "Object-oriented knowledge bases, Part II," *AI Expert*, Vol. 11, 1990, pp. 46-57.
26. P. McBrien and A. Poulovassilis, "A formalisation of semantic schema integration," *Information Systems*, Vol. 23, No. 5, 1998, pp. 307-334.

27. S. Navathe and A. Savasere, "A schema integration facility using object-oriented data model," in [3], Chapter 4, pp. 105-128.
28. ONTOS DB 3.0 Introduction to ONTOS DB 3.0 ONT-30-SUN-IODB-2.1 1989-1994 by ONTOS Inc., 1994.
29. J. A. Robinson, *Logic: Form and Function*, Edinburgh University Press, 1979.
30. M. P. Reddy, B. E. Prasad, P. G. Reddy, and A. Gupta, "A methodology for integration of heterogeneous databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, 1994, pp. 920-933.
31. M. Papazoglou, Z. Tari, and N. Russell, "Object-oriented technology for interschema and language mappings," in [3], Chapter 6, pp. 203-250.
32. P. Scheuermann and E. I. Chong, "Role-based query processing in multidatabase systems," in *Proceedings of 4th Interational Conference on Extending Database Technology*, 1994, pp. 95-108.
33. W. Sull and R. L. Kashyap, "A self-organizing knowledge representation schema for extensible heterogeneous information environment," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 2, 1992, pp. 185-191.
34. S. Spaccapietra, P. Parent, and Y. Dupont, "Model independent assertions for integration of heterogeneous schemas," *VLDB Journal*, Vol. 1, No. 1, 1992, pp. 81-126.
35. S. Spaccapietra and P. Parent, "View integration: a step forward in solving structural conflicts," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 2, 1994, pp. 258-274.

APPENDIX A: SAMPLE INTEGRATION

To briefly illustrate the behavior of the above algorithms, let us trace a sample integration. The example is constructed in such a way that major ideas can be presented in a simple way.

Example 12. Consider two simple local (object-oriented) schemas given in Fig. 18(a).

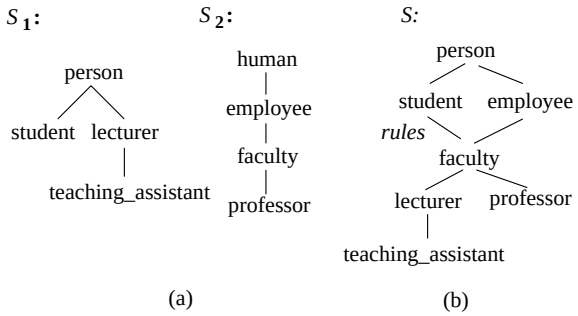

Fig. 18. Two simple local schemas and the assertion set defined for them.

If the assertion set between them is defined as shown in Fig. 18(b), then an integrated schema as shown in Fig. 18(c) will be generated automatically by executing *schema_integration*.

In the following, we will explain the entire process step-by-step. By a computation step, we mean an iteration in *schema_integration* or an iteration in *path_labelling*, represented as a pair: the operations executed in this step and the resulting state of S_b or S_d. Note that the state of S_b is changed by executing lines 5-6, 16, 23, 31 and lines 33-35 in *schema_integration*.

schema_integration (S_1, person; S_2, human):
initial step: source pairs into S_b; S_b state: [(person, human)]
step$_1$: pop and check of the pair on the top of S_b:
person ≡ human;
generation of an integrated class: person;
some pairs into S_b; S_b state: [(student, employee), (lecturer, employee)]
step$_2$: pop and check of the pair on the top of S_b:
no assertion between student and employee;
all the relevant pairs into S_b; S_b state: [(lecturer, employee), (student, faculty)]
step$_3$: pop and check of the pair on the top of S_b:
lecturer ⊆ employee;
call of *path_labelling* (lecturer, S_2, employee, *l*):
initial step: node employee into S_d; S_d state: [employee]
step$_{31}$: pop of the top element of S_d: employee;
check: lecturer ⊆ employee:
labelling: employee $^{<l,>}$;
child nodes of employee
into S_d; S_d state: [faculty]
step$_{32}$: pop of the top elemet of S_d: faculty;
check: lecturer ⊆ faculty:
labelling: faculty $^{<l,>}$;
child nodes of faculty
into S_d: S_d state: [professor]
step$_{33}$: pop of the top element of S_d: professor;
check: no assertion between lecturer and professor
marking professor with *;
faculty ← professor *;
(*since no other nodes appear on the path connecting faculty and professor,
no undoing operation will be done.*)
generation of *is_a* (lecturer, faculty);
professon has no child nodes and
therefore S_d becomes empty; S_d state: []
labelling: lecturer $^{<,l>}$;
label inheritance for child nodes of lecturer: teaching_assistant $^{<,l>}$;
some pairs into S_b: S_b state: [(student, faculty), (teaching_assistant, faculty)]
step$_4$: pop and check of the pair on the top of S_b:
student ∩ faculty;
the following rules will be generated:
$<x: IS_{faculty,\ student}> \Leftarrow <x: IS(S_1 \bullet faculty)>, <y: IS(S_2 \bullet student)>, y = x,$
$<x: IS_{faculty\text{-}}> \Leftarrow <x: IS(S_1 \bullet faculty)>, \neg <x: IS_{faculty,\ student}>,$
$<x: IS_{student\text{-}}> \Leftarrow <x: IS(S_2 \bullet student)>, \neg <x: IS_{faculty,\ student}>.$

some rules for integrated attributes will also be created (see Example 8);
no new pairs into S_b; S_b state: [(teaching_assistant $^{<,I>}$, faculty)]
step$_5$: no checking will be done for
the pair on the top of S_b; (*in terms of the relationship of labels and inherited-labels*)
no new pairs into S_b and
therefore S_b becomes empty; S_b state: [].

In the above execution, we see that an integrated version for S_1(person) and S_2(human), a new link connecting faculty and student, and several rules for declaring the semantic relationships among the integrated concepts (derived in terms of local classes: S_1(student) and S_2(faculty)) are created. Therefore, with our default strategies combined together, *schema_integration* and *path_labelling* will produce the integrated schema shown in Fig. 18(c).

In addition, the following three features of the algorithms can be observed:

1. In each iteration step of *schema_integration*, not all relevant pairs are put into S_b. Therefore, the optimization discussed in the previous subsection is implemented. For example, after S_1(person) ≡ S_2(human) is checked, only (student, employee) and (lecturer, employee) are put into S_b for subsequent checks. (In contrast, in the naive algorithm, pairs such as (student, human), (lecturer, human) and (person, employee) will be put into S_b.)
2. The integration principle for handling is-a paths is correctly realized. For example, only one is-a link between S_1(lecturer) ≡ S_2(faculty) is created, and all other is-a links such as *is_a*(*IS*(lecturer), *IS*(employee)), *is_a*(*IS*(teaching-assistant), *IS* (employee)) and *is_a*(*IS*(teaching-assistant), *IS*(faculty)) are not considered; they will be redundantly generated if an algorithm like that proposed in [33] is used.
3. By utilizing labels, repetitive checking of "⊆" (or "⊇") assertions is avoided. For example, pairs such as (teaching_assistant, employee) and (teaching_assistant, faculty) (for which the inclusion assertion is declared) need not be checked. More importantly, the corresponding depth-first searches are also avoided in this way.

APPENDIX B: EVALUATING VIRTUAL RULES

In this appendix, we will discuss our strategy for evaluating "virtual" rules to show that our method will not damage the autonomy.

Assume that S_1 contains two concepts *mother* and *father* while S_2 contains two other concepts *parent* and *brother*. Then, two rules of the following form will be generated in IS_1:

(1) *parent*(*x*, *y*) ← *mother*(*x*, *y*),
(2) *parent*(*x*, *y*) ← *father*(*x*, *y*).

If *uncle* is a concept of S_2, then the following rule will be generated in IS_2:

(3) *uncle*(*x*, *y*) ← *parent*(*x*, *z*), *brother*(*z*, *y*).

Given a query of the form ?-*uncle*(John, *y*) against IS_2, rule (3) will be evaluated. As in a normal deductive database, rules (1) and (2) will be invoked when *parent* is encountered. But the backward inference process is a bit different. Here, we associate each head predi-

cate q with a set of schema names $\boldsymbol{S}$ with each one containing q as a concept, and each body predicate p with a set of rules $\boldsymbol{R}$ with each one having p as its head. In this way, the above rules can be rewritten as follows:

(1) $parent^{\{S_2\}}(x, y) \leftarrow mother^{\{\}}(x, y)$,
(2) $parent^{\{S_2\}}(x, y) \leftarrow father^{\{\}}(x, y)$,
(3) $uncle^{\{S_3\}}(x, y) \leftarrow parentr^{\{1, 2\}}(x, z), brotherr^{\{\}}(z, y)$,
(4) $mother^{\{S_1\}}(x, y) \leftarrow$,
(5) $father^{\{S_1\}}(x, y) \leftarrow$,
(6) $brother^{\{S_2\}}(x, y) \leftarrow$.

Note that in the above rules, each basic predicate is represented as a rule with an empty body.

Based on the above labeling mechanism, the algorithm for evaluating rules can described as shown below. In the algorithm, q represents a query and Q is a set of rules whose head predicate matches q.

Algorithm *evaluation*(q, Q)
begin
for each rule of the form: $q^{\{\boldsymbol{S}\}} \leftarrow p_1^{\{\boldsymbol{R}_1\}}, ..., p_n^{\{\boldsymbol{R}_n\}} \in Q$ **do**
{temp := $\emptyset$;
for each $s \in \boldsymbol{S}$ **do** (*s repersents a schema name.*)
temp := temp $\cup$ results of evaluating q against s;
for each i = 1 to n **do**
$temp_i := evaluation(p_i, R_i)$; (*recursive call*)
temp' := $temp_1 \bowtie ... \bowtie temp_n$;
result := temp $\cup$ temp';
}
end

The above algorithm is just a naive version used to present our idea clearly. As in deductive databases, the constants appearing in the query and the constant propagation can be used to optimize the evaluation process.

Yangjun Chen ( ) received his BS degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and PhD degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as an assistant professor at the Technical University of Chemnitz-Zwickau, Germany. Dr. Chen is currently a senior engineer at the German National Research Center of Information Technology. His research interests include deductive databases, federated databases, multimedia databases, the constraint satisfaction problem, graph theory and combinatorics. He has published more than 50 works in these areas.