Transformation-based Approach for Integrating Semistructured Data

Sasivimol Kittivoravitkul

June 2nd, 2003

1 Introduction

Semistructured data (SSD) representations such as XML, HTML, and a variety of flatfile formats are becoming increasingly widespread in applications such as genome databases, digital libraries and electronic commerce *etc*. Unlike conventional databases where a predefined and fixed structure is required, semistructured data representations allow the structure of the data to be flexible in the sense that elements in the structure can be absent, can contain composite and multivalued data and can contain other elements. In addition, semistructured data sources are often used to store data whose schema is expected to change rapidly because of the flexibility of the structure.

As a result, semistructured data integration is more challenging than traditional databases integration. The integration must account for the heterogeneities and autonomies among data sources. Additionally, it must be able to handle the flexibility, and importantly the evolution of such sources.

The aim of this report is to propose a schema transformation-based approach to semistructured data integration. The approach is based on the use of reversible sequences of primitive transformations, called **both as view (BAV)** [MP03]. In the BAV approach, local schemas are incrementally transformed into a global schema by applying to them a sequence of primitive transformations. The BAV approach has been applied to the integration of traditional databases such as the relational model, the ER model, UML. Here it is applied to the integration of semistructured data sources.

To integrate semistructured data, first of all, it is essential to have a data model which can be naturally represented semistructured data. In this report, a new modelling language to handle semistructured data called **YATTA**, which is a variation of the **YAT model** by [CDSS98], is introduced. A set of primitive transformation rules to manipulate this model is also defined. The idea is to combine semistructured data sources represented in the YATTA model, transform them into equivalent form, and merge them into a global view/schema.

This report is structured as follows. In Section 2, a running example for this report is described. Then the YATTA model is proposed in Section 3. In Section 4, the primitive transformations rules for YATTA are defined. The methodology for defining a sequence of transformation is also given and illustrated using the running example. Furthermore, how our

approach handles queries and evolution of sources is discussed in this section. In Section 5, the transformation between the YATTA and relational models is presented. This shows how a schema in the relational model can be derived from YATTA. Finally, a concluding remark and future work are given in Section 6.

2 Running Examples

Figure 1 illustrates examples of semistructured data of two sources, *S1* and *S2*, that will be used as a running example in this report. The examples contain a portion of student data. The data is represented in human-readable form with its structure included. Informally, the structure can be described as containing student records in *S1* 1(a) and course records in *S2* 1(b).

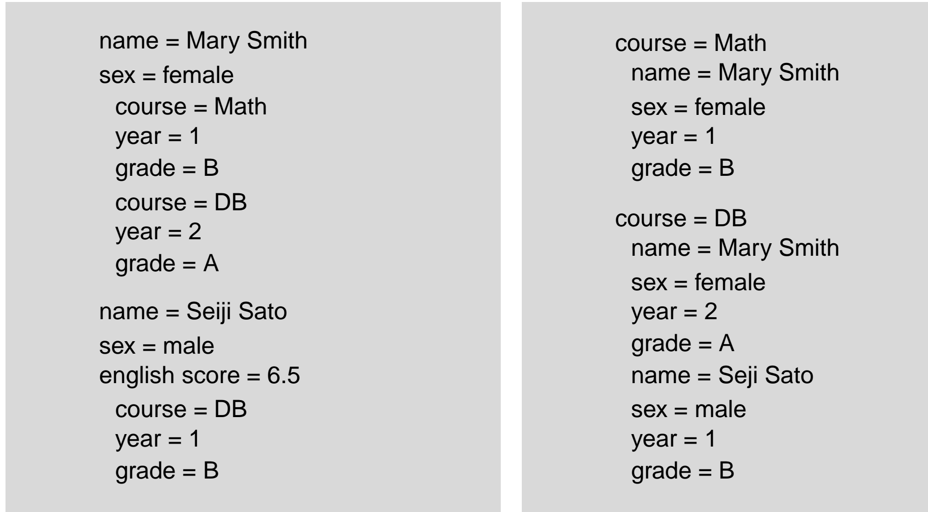
In *S1*, the structure of student records can vary according to students nationality. For home students, each record consists of elements which are name, sex and subrecords with elements course, year and grade, describing course they attended. For overseas students, an extra element *i.e.* english score is added. Every record in $S1$ consists of a collection of related data values and each value corresponds to a particular element of the record. For example, the first eight lines in Figure 1(a) are a record for the student Mary Smith, the next six lines are a record for the student Seiji Sato. The records are made up of elements that describe the student with subrecords. The first eight lines in the example say that the student record, Mary Smith, has two elements which are name and sex, and two subrecords which describe the courses she attended, the years she took the courses, and the grades obtained.

In *S2*, each record consists of elements which are course and subrecords with elements name, sex, year and grade, describing student attended the course. The first four lines in Figure 1(b), a record for the Math course, say that the Math course has an element course and a subrecord of a student Mary Smith who attended such course in her first year and got a grade "B".

3 Model of Semistructured Data

Various models such as **Object Exchange Model (OEM)** [PGMW95], **Edge-Labelled tree** [Pet96], **graph schema** [BDFS97, SBS00], **DataGuides** [GW97], and the **YAT model** [CDSS98] *etc.* have been proposed to represent semistructured data. These models were compared and contrasted in [Kit02], where it was established that the YAT model is the most promising for the transformation-based approach to semistructured data integration. The advantages of YAT over other models are that it represents order and cardinality constraints, and it can represent data in different levels of abstraction.

However, in our approach, the YAT model has two disadvantages. First, the need to specify the order of nodes in the YAT model may limit the flexibility in representing semistructured data. It is often the case that some semistructured data may not need to be ordered. Second, the YAT model allows data to be represented in various levels of details. In data integration, it is important that a modelling language gives a precise definition of a database schema.



(a) *S1.* (b) *S2.*

Figure 1: Semistructured data containing student records S1 and course records S2.

A variation of the YAT model, called **YATTA** (YAT for Tranformation-based Approach), is therefore proposed. YATTA extends the YAT model to distinguish between ordered and unordered data. Also, YATTA restricts the YAT model to just two levels of abstraction: the **schema level** where a structure of data is represented and the **data level** where real data is represented.

3.1 The YATTA Model

The **YATTA model** consists of a **YATTA schema** and **YATTA data**, representing the structure and content of semistructured data respectively. The models are rooted trees with labelled nodes. Each node is described by a tuple ⟨*label, type*⟩ in a YATTA schema and a triple ⟨*label, type, value*⟩ in a YATTA model, where *label* is a string describing what a node represents, *type* represents data type of a node, and *value* represents the value associated with the node. The *type* can be atomic *e.g.* string, integer *etc.* or compound *i.e.* set (marked '{}'), list (marked '[]') or bag (marked '⟨⟩'). In a YATTA data, if a node is of compound type, *value* is an integer identifer *id*. The outgoing edges of list nodes are ordered from left to right whereas those of set and bag nodes are unordered. The edges of a YATTA schema are labelled with cardinality constraints, where '∗' indicates zero or more occurrences, '+' indicates one or more occurrences, and '?' indicates zero or one occurrence. No label and '-' indicate exactly one occurrence. '-' is also used to identify a key node in a YATTA schema. In a YATTA data the edge is repeated the number of times it occurs, and data values are shown under leaf nodes.

An example of the YATTA schema for the student data in Figure 1(a) is shown in Figure 2. In the example, each student record is uniquely identified by the name of the student , and thus in Figure 2 the edge of name is labelled with '-'. Since the details about sex and english_score are optional, their edges are labelled with '?'. The fact that each student must have attended at least one course is represented by '+' on the edge. Information about courses is held as

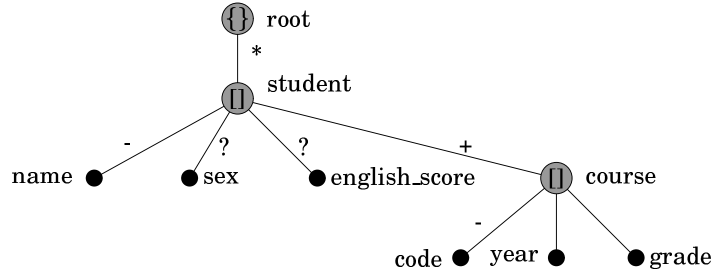


Figure 2: YATTA schema for S_1

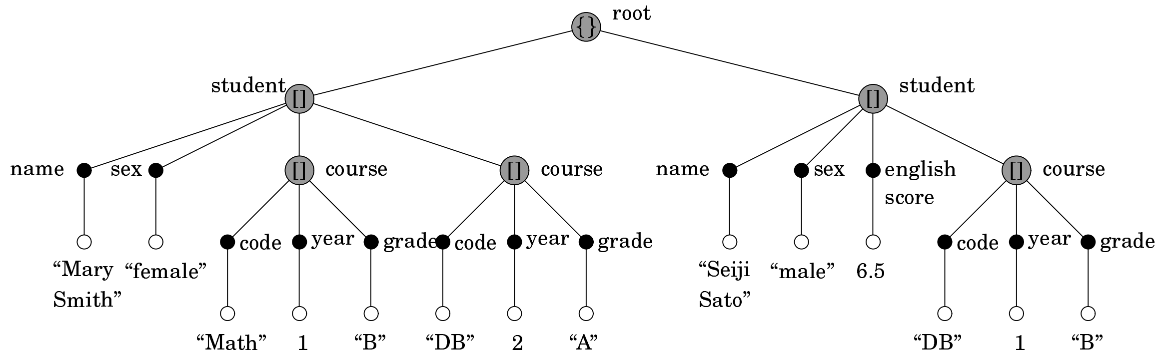


Figure 3: A YATTA data for S_1

subrecords of student, where code uniquely identifies the course each student attended, year indicates the year the student attended the course, and grade indicates the grade obtained. Figure 3 illustrates the YATTA data that conforms to the YATTA schema in Figure 2.

The syntax of YATTA is similar to that of YAT, except that data type is not represented as a node in YATTA. Also, a node of compound type is denoted by its labelled followed by its children between {}, [] or ⟨⟩ if the node is of set type, list type or bag type respectively. If a compound node is followed by its children in {}, the children are unordered and each child can only occur once. If it is followed by its children in [], the children are ordered from the leftmost to the rightmost. If it is followed by its children in ⟨⟩, the children are unordered and each child can occur more than once. The YATTA syntax for the YATTA schema in Figure 2 is illustrated below.

root $\rightarrow^{*}$ {student $\rightarrow$ [$\rightarrow^{-}$ name $\rightarrow$ String,
$\rightarrow^{?}$ sex $\rightarrow$ String,
$\rightarrow^{?}$ english_score $\rightarrow$ Integer,
$\rightarrow^{+}$ course $\rightarrow$ [$\rightarrow^{-}$ code $\rightarrow$ String,
$\rightarrow$ year $\rightarrow$ Integer,
$\rightarrow$ grade $\rightarrow$ String]] }

In the YATTA model, all nodes, except the root node, can only exist when its parent node exists. We therefore define a YATTA node n as an only construct in the YATTA model. Each YATTA node is identified by its **scheme**. The scheme of a YATTA node n is $\langle\langle a, n, t, c\rangle\rangle$, where a is the sequence of ancestor nodes of a node n denoted $a_1.a_2.a_3...a_p$. For example, a of name in Figure 2 is "root.student". The value of a is the value associated with the last node a_p in the sequence, which is the parent of n. The t is the data type of n and the c is a cardinality constraint on an edge between a_p and n. All the constructs of the YATTA schema in Figure 2 and their instances are shown below:

⟨⟨root, student, list, ∗⟩⟩ = {⟨&0, &1⟩, ⟨&0, &2⟩}
⟨⟨root.student, name, string, -⟩⟩ = [⟨&1, "Mary Smith"⟩, ⟨&2, "Seiji Sato"⟩]
⟨⟨root.student, sex, string, ?⟩⟩ = [⟨&1, "female"⟩, ⟨&2, "male"⟩]
⟨⟨root.student, english_score, real, ?⟩⟩ = [⟨&2, 6.5⟩]
⟨⟨root.student, course, list, +⟩⟩ = [⟨&1, &11⟩, ⟨&1, &12⟩, ⟨&2, &13⟩]
⟨⟨root.student.course, code, string, ⟩⟩ = [⟨&11, "Math"⟩, ⟨&12, "DB"⟩, ⟨&13, "DB"⟩]
⟨⟨root.student.course, year, integer, ⟩⟩ = [⟨&11, 1⟩, ⟨&12, 2⟩, ⟨&13, 1⟩]
⟨⟨root.student.course, grade, string, ⟩⟩ = [⟨&11, "B"⟩, ⟨&12, "A"⟩, ⟨&13, "B"⟩]

Here the "&" followed by a number represents the node identifier *id* which is the value of the complex node. For instance, &0 is a root node identifer.

3.2 Representing Semistructured Data in YATTA

In the previous section, the YATTA model has been described. This section discusses how to represent semistructured data in the YATTA model.

Semistructured data typically contains a sequence of records. Each record consists of a collection of related data values, where each value corresponds to a particular element of the record. Recall the example of semistructured data Figure 1(a), the example contains a record for the student Mary Smith followed by a record for the student Seiji Sato. Each student record consists of elements the student's name (name), the student's sex (sex), the student's English score (english_score), and the courses student attended which are described in the subrecord (course). The course subrecord contains elements the course's code (code), the years the student took the courses (year) and the grades obtained (grade).

To represent a schema embedded in semistructured data in a YATTA schema, a top-down methodology is applied and is defined as follows:

1. Define records in semistructured data as compound nodes and associate them with the root node. In the example, the student records are defined as compound nodes labelled with student and linked to the root node as shown in Figure 2.

   For each compound node, if the order of its elements is not important and each element can not be repeated, its type is set type. If the order of its elements is important, its type is list type. If there is the repetition of elements and elements and the order of the elements is not important, its type is bag type. In the example, a student node can be described as a tuple ⟨student,list⟩. The node is of type list since the order of elements in the semistructured data is important.

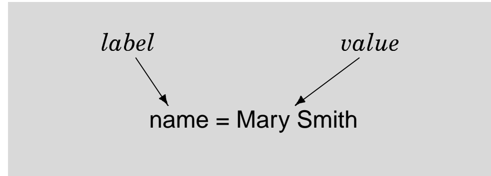


Figure 4: A line of semistructured data for student records.

2. Represent each element of a record as a child node. The child node is of atomic type if it is not a subrecord *e.g.* name, sex and english_score. Otherwise, the child node is defined as a compound node as in step 1 with their elements as child nodes. For example, course, a subrecord of student, is a compoud node with child nodes code, year and grade

   Each node is labelled with information to describe the element. In the example, *label* describing the elements can be derived from the data source. Each line in the data source can be expressed as a pattern "*label* = *value*", Figure 4, where name is a label for the node and Mary Smith is a value of such node.

The cardinality constraint of each node depends on occurences of records in a semistructured data source if the node is of compound type, and depends on occurences of elements in a record if the node is of atomic type. If records or elements occur exactly once, the cardinality constraint of the node is indicated by nothing but if the element is uniquely identified the record, the cardinality constraint is indicated by '-'. If there are one or more occurences, the cardinality constraint is indicated by "+". If there is zero or one occurence, the cardinality constraint is indicated by "?". If there are zero or more occurences, the cardinality constraint is indicated by "$*$".

A YATTA data is an instance of a YATTA schema for a particular semistructured data source. Each object in a YATTA data is associated with the object inside a YATTA schema, where the values of the node in a YATTA data are data values in the source if the node is of atomic type and the values are integer identifiers if the node is of compound type. As illustrated in the previous section, the YATTA data Figure 3 conforms to the YATTA schema Figure 2. but contains all the data values in the semistructured data source Figure 1(a).

4 Transformation-based Approach

The objective of data integration is to provide an integrated global schema to the local schemas which refer to existing data sources. Data integration therefore involves the process of transforming data from the various formats and structures in which it is represented in the local schemas into a form of compatible with the global schema. This process is called **transformation** process.

Our approach is based on the BAV approach which supports transformation process in data integration by defining associations between constructs in different schemas using transformation rules. The approach has been used on a number of structured data models such as the relational model and the ER model as described in [MP99b]. Here the BAV approach is applied on semistructured data model, YATTA.

This section presents primitive transformation rules on YATTA and illustrates how the rules can be applied to transform YATTA schemas of local sources into the global schema.

4.1 Primitive Transformation rules on the YATTA model

The primitive transformation rules on the YATTA model are functions that take a source YATTA schema, perform a transformation and return a new YATTA schema.

The primitive transformation rules on the YATTA model are defined as follows:

- renameYattaNode($\langle\langle a, n_1, t, c\rangle\rangle, \langle\langle a, n_2, t, c\rangle\rangle$) renames a node n_1 to have a new name n_2.
- addYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) adds a new node n linked to an existing node a_p, where a_p is the last node in a. The value of a is the value of a_p. The query q specifies the extent of the binary relationship between a_p and n.

  If a_p is of type t = list, then to add the node n there must be a position query p stating the position n relative to other sibling nodes of a_p. This position query may either be before(n'), after(n'), or any, where n' is some other node linked to a_p and any means in any place relative to other siblings.

  If a_p is of type $t \neq$ list, then there must be no p query field.
- deleteYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) deletes an existing node n and its relationship with a_p where a_p is the last node in a. The query q tells how the extent of the node can be restored from the remaining constructs. The position query p must be provided if a_p is of type t = list. A proviso associated with the transformation is that the node n does not have any child nodes.

In data integration, it is often the case that source schemas may not be precisely equivalent. Two more transformation rules which allow transformations between overlapping YATTA schema are provided. These rules can be defined in terms of the add and delete rule above where the query q can not return all values associated with the constructs.

- extendYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) adds a new node n. The extent of n is only partially defined by the transformation. If no values can be determined by any query, then the special value void may be used for q, or the query field omitted completely. If a_p, the last node in a, is of type t = list, the position query p must be provided.
- contractYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) deletes an existing node n. Only some values of n may be restored by the transformation. The value void may be used where no values can be determined, or the query field omitted. The position query p must be provided if a_p, the last node in a, is of type t = list. A proviso associated with the transformation is that the n does not have any child nodes.

According to BAV approach, there exists a **reverse primitive transformation** for every primitive transformation [MP99a]. The same concept is applied here: Table 1 shows the reverse primitive transformation for each YATTA primitive transformation.

Transformation	Reverse Transformation
renameYattaNode($\langle\langle a, n_1, t, c\rangle\rangle, \langle\langle a, n_2, t, c\rangle\rangle$)	renameYattaNode($\langle\langle a, n_2, t, c\rangle\rangle, \langle\langle a, n_1, t, c\rangle\rangle$)
addYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)	deleteYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)
deleteYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)	addYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)
extendYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)	contractYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)
contractYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)	extendYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$)

Table 1: Reverse transformation

4.2 A methodology for the YATTA Model transformations

This section describes a methodology for determining a sequence of primitive tranformations, called a **pathway**.

Given two models M and M', the methodology to define a pathway from one model M to another model M' is as follows.

1. Conduct a breadth first iteration over the nodes n' of M', and for each node n' not found in M:
   (a) If n' is of complex type, determine if there is a query q on M such that there is a one to one mapping between values returned by q and values associated to n'. If there is, then a new node n' is added into M by applying a rule addYattaNode, with the special function generateId used on the values returned by q to generate the identifiers of the complex node. This function acts as a kind of hashing function, which always returns the *same* identifier for the same input values, and always returns *distinct* identifiers for distinct input values.
   (b) If n' is of simple type, determine if there is a query q on M such that the values returned by q are equal to the values associated with n'. If there is, then a new node n' is added into M by applying a rule addYattaNode, with q placed as the query part of the transformation. If the query only returns some of the values of n' then instead use extendYattaNode.
   (c) If no query can be determined, then use extendYattaNode with the value void, which indicates that there is no method to determine anything about the instances of n' in M' from the information in M.
2. Once all the constructs of M' have been added into the model M, all redundant constructs of M that do not appear in M' are removed. This entails a breadth first search over the nodes n of M, and for each node n which do not appear in M':
   (a) If n is of complex type, determine if there is a query q on the constructs of M' in M such that there is a one to one mapping between values returned by q and values associated to n. If there is, then the node n is deleted by applying a rule deleteYattaNode, with the function generateId used on the values returned by q to restore the values of n.
   (b) If n is of simple type, determine if there is a query q on the constructs of M' in M such that the values returned by q are equal to the values associated with n. If

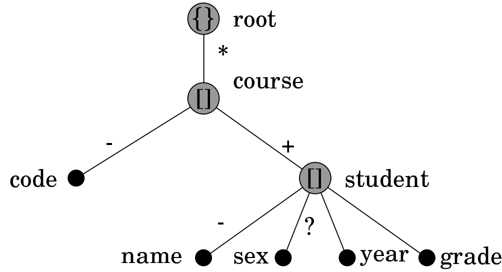


Figure 5: A YATTA schema for S_2

there is, then the node n is deleted from M by applying a rule deleteYattaNode, with q to restore the values of n. If the query only restores some of the values of n then contractYattaNode is used instead.

(c) If no query can be determined, then use contractYattaNode with the value void, which indicates that there is no method to restore the instances of n in M after the deletion.

According to the methodology, the pathway consists of a growing phase where the new constructs are added to the model in step 1, followed by shrinking phase where the redundant constructs are removed in step 2.

4.3 Example of Transformations

Suppose we want to integrate the data sources S_1 Figure 1(a) with S_2 Figure 1(b). First of all, S_1 and S_2 are represented in the YATTA model as shown in Figure 2 and 3 for S_1. To represent S_2 in the YATTA model, the course entity is defined as a compound node course associated with a root node. The course code is defined as an atomic node code whereas the student record is described as a compound node student. Both code and student are represented as child nodes of course. Since the value of code is uniquely identity a course record in S_2, code is designated as a primary key node in the YATTA schema of S_2. The attributes of the student entity, which are the student name, the student sex, the year student attended the course and the grade their obtained, are definded as atomic nodes name, sex, year and grade under student.

Figure 5 and 6 shows the YATTA schema and the YATTA data for S_2. As can be seen, the YATTA model for S_1 and S_2 are semantically heterogeneous because the same concept is represented using different modelling constructs. To illustrate the use of the primitive transformations, we give here a transformation pathway that transform the YATTA schema Y_1 of S_1 to the YATTA schema Y_2 of S_2.

Using our methodology, we must first identify how to add nodes to Y_1 that appear in Y_2. For instance, in Y_2 there are code nodes that corresponds to instances of code nodes in Y_1.

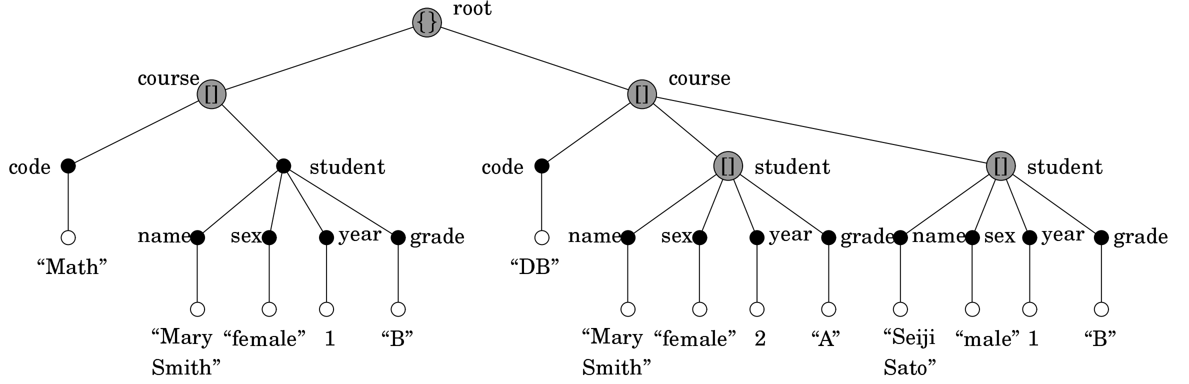


Figure 6: A YATTA data for S_2

To simplify the presentation of rules, we will use the functions below to represent queries over the YATTA models. For example, a function f tells how to get a code of the course student attended in Y_1. A function h tells how to get a name of student attended the course in Y_2. Note that when schemes of the constructs appear in a query, we may omit types and cardinality constraints, and use $\langle\langle a, n\rangle\rangle$ instead of $\langle\langle a, n, t, c\rangle\rangle$ to represent a YATTA node n.

$$
\begin{aligned}
\mathbf{Y_1}: \quad f &= \{\langle x, y\rangle \mid \langle r, x\rangle \in \langle\langle \text{root, student}\rangle\rangle, \\
&\quad \langle x, w\rangle \in \langle\langle \text{root.student, course}\rangle\rangle, \\
&\quad \langle w, y\rangle \in \langle\langle \text{root.student.course, code}\rangle\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
g &= \{\langle x, y, v, w, z\rangle \mid \langle r, x\rangle \in \langle\langle \text{root, student}\rangle\rangle, \\
&\quad \langle x, y\rangle \in \langle\langle \text{root.student, name}\rangle\rangle, \\
&\quad \langle x, v\rangle \in \langle\langle \text{root.student, sex}\rangle\rangle, \\
&\quad \langle x, u\rangle \in \langle\langle \text{root.student, course}\rangle\rangle, \\
&\quad \langle u, w\rangle \in \langle\langle \text{root.student.course, year}\rangle\rangle, \\
&\quad \langle u, z\rangle \in \langle\langle \text{root.student.course, grade}\rangle\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{Y_2}: \quad h &= \{\langle x, y\rangle \mid \langle r, x\rangle \in \langle\langle \text{root, course}\rangle\rangle, \\
&\quad \langle x, z\rangle \in \langle\langle \text{root.course, student}\rangle\rangle, \\
&\quad \langle z, y\rangle \in \langle\langle \text{root.course.student, name}\rangle\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
l &= \{\langle x, y, v, w, z\rangle \mid \langle r, x\rangle \in \langle\langle \text{root, course}\rangle\rangle, \\
&\quad \langle x, y\rangle \in \langle\langle \text{root.course, code}\rangle\rangle \\
&\quad \langle x, u\rangle \in \langle\langle \text{root.course, student}\rangle\rangle \\
&\quad \langle u, v\rangle \in \langle\langle \text{root.course.student, sex}\rangle\rangle, \\
&\quad \langle u, w\rangle \in \langle\langle \text{root.course.student, year}\rangle\rangle, \\
&\quad \langle u, z\rangle \in \langle\langle \text{root.course.student, grade}\rangle\rangle\}
\end{aligned}
$$

Below is a pathway of transformations on the YATTA model Y_1 to Y_2.

Pathway $Y_1 \rightarrow Y_2$

① addYattaNode($\langle\langle$root, course, list,$*\rangle\rangle, \{\langle \&0, x\rangle \mid \langle y, z\rangle \in f \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$)

② addYattaNode($\langle\langle$root.course, code, string,-$\rangle\rangle, \{\langle x, z\rangle \mid \langle y, z\rangle \in f \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$, any)

③ addYattaNode($\langle\langle$root.course, student, list,+$\rangle\rangle, \{\langle x, y\rangle \mid \langle w, z\rangle \in f \wedge x = \text{generateId}(\langle \&0, z\rangle) \wedge \langle w, u, _, _, _\rangle \in g \wedge y = \text{generateId}(\langle z, u\rangle)\}$, after($\langle\langle$root.course, code$\rangle\rangle$))

④ addYattaNode($\langle\langle$root.course.student, name, string,-$\rangle\rangle, \{\langle x, y\rangle \mid \langle u, y, _, _, _\rangle \in g \wedge \langle u, z\rangle \in f \wedge x = \text{generateId}(\langle z, y\rangle)\}$, any)

⑤ addYattaNode($\langle\langle$root.course.student, sex, string,?$\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, y, _, _\rangle \in g \wedge \langle u, z\rangle \in f \wedge x = \text{generateId}(\langle z, y\rangle)\}$, any)

⑥ addYattaNode($\langle\langle$root.course.student, year, integer, $\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, _, y, _\rangle \in g \wedge \langle u, z\rangle \in f \wedge x = \text{generateId}(\langle z, y\rangle)\}$, after($\langle\langle$root.course.student, sex$\rangle\rangle$))

⑦ addYattaNode($\langle\langle$root.course.student, grade, string, $\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, _, _, y\rangle \in g \wedge \langle u, z\rangle \in f \wedge x = \text{generateId}(\langle z, y\rangle)\}$, after($\langle\langle$root.course.student, year$\rangle\rangle$))

⑧ deleteYattaNode($\langle\langle$root.student.course, grade, string, $\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, _, _, y\rangle \in l \wedge \langle u, z\rangle \in h \wedge x = \text{generateId}(\langle z, y\rangle)\}$, after($\langle\langle$root.student.course, year$\rangle\rangle$))

⑨ deleteYattaNode($\langle\langle$root.student.course, year, integer, $\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, _, y, _\rangle \in l \wedge \langle u, z\rangle \in h \wedge x = \text{generateId}(\langle z, y\rangle)\}$, after($\langle\langle$root.student.course, code$\rangle\rangle$))

⑩ deleteYattaNode($\langle\langle$root.student.course, code, string,-$\rangle\rangle, \{\langle x, y\rangle \mid \langle u, y, _, _, _\rangle \in l \wedge \langle u, z\rangle \in h \wedge x = \text{generateId}(\langle z, y\rangle)\}$, any))

⑪ deleteYattaNode($\langle\langle$root.student, course, list,+$\rangle\rangle, \{\langle x, y\rangle \mid \langle w, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle) \wedge \langle w, u, _, _\rangle \in l \wedge y = \text{generateId}(\langle z, u\rangle)\}$, after($\langle\langle$root.student, english_score$\rangle\rangle$))

⑫ contractYattaNode($\langle\langle$root.student, english_score,?$\rangle\rangle$, void, after($\langle\langle$root.student, sex$\rangle\rangle$))

⑬ deleteYattaNode($\langle\langle$root.student, sex,?$\rangle\rangle, \{\langle x, y\rangle \mid \langle u, _, y, _, _\rangle \in l \wedge x = \text{generateId}(\langle \&0, y\rangle)\}$, after($\langle\langle$root.student, name$\rangle\rangle$))

⑭ deleteYattaNode($\langle\langle$root.student, name, string,-$\rangle\rangle, \{\langle x, z\rangle \mid \langle y, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$, any)

⑮ deleteYattaNode($\langle\langle$root, student, list,$*\rangle\rangle, \{\langle \&0, x\rangle \mid \langle y, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$)

In transformation ①, a new course node in Y_2 is added into Y_1. Since course is of complex type, its extents, which are integer identifers, depend on distinct course entry in Y_1. In order to generate the identifiers of course for every distinct course entry in Y_1, the function generateId is then used. In the transformation, no position query is required since root is of set type.

Transformation ② performs a similar process to define a code node linked to course. The extent of code is the values returned by f for the code in Y_1, associated to the same identifer that we generated in ①. Since code is the first sibling of course which is of type list, its order is set to any.

Transformation ③ states that a new student node is added for every appropriate student entry in Y_1, associated to the course node generated in ①. The new nodes are put after code.

In transformation ④-⑦, new nodes name, sex, year and grade are added into Y_1. The values of the new nodes name, year and grade are equal to those of existing nodes in Y_2, name, sex, year and grade respectively.

Once all new nodes of Y_2 are all added. This means the growing phase is completed and the shrinking phase is started. Now the nodes of Y_1 which is redundant and do not appear in Y_2 can be removed. We note that there are instances of a node grade of Y_1 for every new node grade of Y_2. The path to such grade nodes in Y_2 is identified by the query function l above.

Transformation ⑧ uses the query function l to remove grade node of Y_1, using information that will be present in Y_2 to restores its values.

In transformation ⑨ and ⑩, a similar process to transformation ⑧ is performed to remove year and code of Y_1. The values of the deleted nodes can be restored using the query specified in the transformations.

Before defining transformation ⑪ to delete a complex node course, every child node of course should be removed first. The values of course can be restored using the function generateId which returns the identifiers of the distinct values of code and name.

Since the instances of the english_score node in Y_1 cannot be restored from the information in Y_2, an contractYattaNode transformation with no query field is used in transformation ⑫.

Transformation ⑬, ⑭ and ⑮ remove sex, name and student node in Y_1 respectively. The values of the deleted nodes can be restored from the new nodes sex, name and student of Y_2.

Due to the reversibility of transformations, transformation pathways are **bi-directional**. Thus the transformation pathway for the mapping from Y_2 back to Y_1 can be automatically derived. Below gives the reverse transformation of $Y_1 \rightarrow Y_2$.

Pathway $Y_2 \rightarrow Y_1$

⓯ addYattaNode(⟨⟨root, student, list,∗⟩⟩, $\{\langle \&0, x\rangle \mid \langle y, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$)

⓮ addYattaNode(⟨⟨root.student, name, string,-⟩⟩, $\{\langle x, z\rangle \mid \langle y, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle)\}$, any)

⓭ addYattaNode(⟨⟨root.student, sex,?⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, y, _, _\rangle \in l$ $\wedge\, x = \text{generateId}(\langle \&0, y\rangle)\}$, after(⟨⟨root.student, name⟩⟩))

⓬ extendYattaNode(⟨⟨root.student, english_score,?⟩⟩, void, after(⟨⟨root.student, sex⟩⟩))

⓫ addYattaNode(⟨⟨root.student, course, list,+⟩⟩, $\{\langle x, y\rangle \mid \langle w, z\rangle \in h \wedge x = \text{generateId}(\langle \&0, z\rangle)$ $\wedge\, \langle w, u, _, _\rangle \in l \wedge y = \text{generateId}(\langle z, u\rangle)\}$, after(⟨⟨root.student, english_score⟩⟩))

❿ addYattaNode(⟨⟨root.student.course, code, string,-⟩⟩, $\{\langle x, y\rangle \mid \langle u, y, _, _, _\rangle \in l \wedge \langle u, z\rangle \in h$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, any))

❾ addYattaNode(⟨⟨root.student.course, year, integer, ⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, _, y, _\rangle \in l \wedge \langle u, z\rangle \in h$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, after(⟨⟨root.student.course, code⟩⟩))

❽ addYattaNode(⟨⟨root.student.course, grade, string, ⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, _, _, y\rangle \in l \wedge \langle u, z\rangle \in h$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, after(⟨⟨root.student.course, year⟩⟩))

❼ deleteYattaNode(⟨⟨root.course.student, grade, string, ⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, _, _, y\rangle \in g \wedge \langle u, z\rangle \in f$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, after(⟨⟨root.course.student, year⟩⟩))

❻ deleteYattaNode(⟨⟨root.course.student, year, integer, ⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, _, y, _\rangle \in g \wedge \langle u, z\rangle \in f$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, after(⟨⟨root.course.student, sex⟩⟩))

❺ deleteYattaNode(⟨⟨root.course.student, sex, string,?⟩⟩, $\{\langle x, y\rangle \mid \langle u, _, y, _, _\rangle \in g \wedge \langle u, z\rangle \in f$ $\wedge\, x = \text{generateId}(\langle z, y\rangle)\}$, any)

❹ deleteYattaNode(⟨⟨root.course.student, name, string,-⟩⟩, $\{\langle x,y\rangle \mid \langle u,y,_,_,_\rangle \in g \wedge \langle u,z\rangle \in f$ $\wedge\ x = \text{generateId}(\langle z,y\rangle)\}, \text{any})$

❸ deleteYattaNode(⟨⟨root.course, student, list,+⟩⟩, $\{\langle x,y\rangle \mid \langle w,z\rangle \in f \wedge x = \text{generateId}(\langle \&0,z\rangle)$ $\wedge\ \langle w,u,_,_,_\rangle \in g\ \wedge\ y = \text{generateId}(\langle z,u\rangle)\}$, after(⟨⟨root.course, code⟩⟩))

❷ deleteYattaNode(⟨⟨root.course, code, string,-⟩⟩, $\{\langle x,z\rangle \mid \langle y,z\rangle \in f \wedge x = \text{generateId}(\langle \&0,z\rangle)\}$, any)

❶ deleteYattaNode(⟨⟨root, course, list,∗⟩⟩, $\{\langle \&0,x\rangle \mid \langle y,z\rangle \in f \wedge x = \text{generateId}(\langle \&0,z\rangle)\})$

4.4 Query Translations

From the pathway, queries posed on Y_1 can be automatically translated into queries on Y_2, and *vice versa*. For example, a simple query Q_{Y_1} on Y_1 asks for all student names in S_1.

$$Q_{Y_1} = \{x \mid \langle y,z\rangle \in \langle\langle \text{root, student}\rangle\rangle \wedge \langle z,x\rangle \in \langle\langle \text{root.student, name}\rangle\rangle\}$$

The query Q_{Y_1} can be translated into the equivalent query Q_{Y_2} on Y_2 by substituting deleted constructs appearing in Q_{Y_1}. The ⟨⟨root, student⟩⟩ and ⟨⟨student, name⟩⟩ are replaced by their restoring query in the transformations ⑮ and ⑭ and performed some logical simplification to give:

$$Q_{Y_2} = \{x \mid \langle y,z\rangle \in \langle\langle \text{root, course}\rangle\rangle \wedge \langle y,w\rangle \in \langle\langle \text{root.course, student}\rangle\rangle \wedge \langle w,x\rangle \in \langle\langle \text{root.course.student, name}\rangle\rangle \wedge w = \text{generateId}(\langle \&0,x\rangle)\}$$

4.5 Handling Evolution in YATTA Transformations

According to the BAV approach [MP03], the evolution of semistructured data sources can be handled by incrementally extended and modified pathways. Suppose information about tutors is added into the source S_1, a YATTA schema Y_1', which includes new information tutor node, is shown in Figure 7.

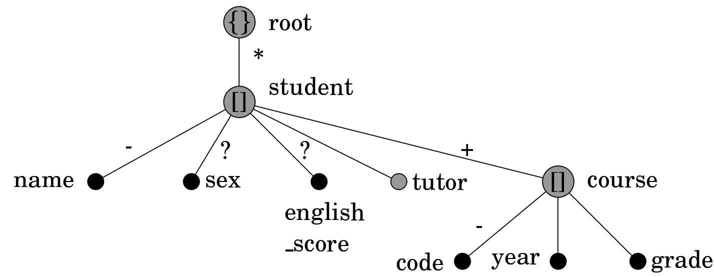


Figure 7: An evolved YATTA schema for student records

The evolution of Y_1 to Y_1' can be handled as a sequence of transformations T, $Y_1 \rightarrow Y_1'$ as below:

Pathway $Y_1 \rightarrow Y_1'$

⑯ extendYattaNode(⟨⟨root.student, tutor, string, ⟩⟩, *void*, after(⟨⟨root.student, english_score⟩⟩))

The extend transformation is used in ⑯, since the values of the node tutor are not derivable from any constructs in Y_2.

The reverse transformations pathway of T denoted T' is automatically derived and is illustrated as follows:

Pathway $Y_1' \rightarrow Y_1$

⓰ contractYattaNode(⟪root.student, tutor, string, ⟫, $void$, after(⟪root.student, english_score⟫))

The global view Y_2 can be repaired in order to reflect the change in Y_1 by a new pathway $Y_1' \rightarrow Y_2$ denoted T_{new}. The new pathway can be obtained by prefixing a pathway T' to the old pathway T_{old}, Y_1 to Y_2.

$$T_{new} = T';T_{old}$$

To ensure transformation preservation, the transformation t in T is checked and handled according to the operation (add, delete, rename, extract and contract) of the transformation. As described in [MP02], each transformation t in the pathway $S_2 \rightarrow S_2'$ needs to be handled according to the following cases.

- If t is add, delete or rename transformation, the new pathway $S_2' \rightarrow S_g$ will be equivalent to the old one $S_2 \rightarrow S_g$. No change is required for the global schema. All information in the global schema that was derivable from S_2 may now be derived from S_2'.
- If t is of form contract, the global schema S_g may contain constructs that are no longer available in the source model S_2'. Such constructs can be automatically removed from the global schema.
- If t is extend, the extended constructs may or may not be represented in the global schema. The constructs in the global schema needs to be investigated.

In the example, the transformation t is extendYattaNode. According to the rules above, the constructs of Y_2 need to be examined. Since the new node tutor can not be derived from any constructs of Y_2, nothing has to be done to the new pathway, except a position query of the construct after the tutor node *i.e.* course node needs to be changed to 'after(⟪root.student, tutor⟫)'.

5 YATTA and the Relational Model Transformations

In the previous section, the transformation of semistructured data of different sources represented in the YATTA models has been shown. In this section, how our approach can handle the transformation between the YATTA model and the relational model is illustrated.

This will facilitate the integration between semistructured data sources and convensional databases, relational databases in particular. The additional advantage is that the transformation allows the automatic migration of data from semistructured data sources to relational databases.

Relational Construct	Scheme of Construct
relation R	$\langle\langle R\rangle\rangle$
attribute a	$\langle\langle R, a, n\rangle\rangle$ if n=null then constraint is {0,1}, {1..N} else if n=key then constraint is {1}, {1} else if n=notnull then constraint is {1}, {1..N}
primary key	$\langle\langle R, k_1, ..., k_n\rangle\rangle$
foreign key	$\langle\langle R, R_f, k_1, ..., k_n\rangle\rangle$ where R_f is a relation containing the primary key, attributes k_1,...,k_n, in R

Table 2: Relational model constructs

	YATTA model Construct	Relational model Construct
	a YATTA node v, $\langle\langle a, v, t, c\rangle\rangle$	
Rule 1	if t=complex type and id(v)=generateId($\langle k_1, ..., k_n\rangle$)	a relation $\langle\langle v\rangle\rangle$ with a primary key $\langle\langle v, k_1, ..., k_n\rangle\rangle$

Rule 2	if t=atomic type
	if (c='*') or (c='?') or (c_p='*') or (c_p='?')
	else
		an attribute $\langle\langle a_p, v, n\rangle\rangle$
		n=null
		n=not null

Rule 3	if t=atomic type and (c=' ') and id(a_p)=generateId(v) and type(s_v) = complex type and s_v=generatedId(v)	a foreign key $\langle\langle s_v, a_p, v\rangle\rangle$
where	a_p is the last node in a k_1...k_n are YATTA nodes c_p is the cardinality constraint of a_p s_v is a sibling of v id() is a function returned integer identifiers of an input node generateId() is a function returned distinct identifiers for distinct input values type() is a function returned type of an input node	

Table 3: The association between YATTA and the relational model constructs

5.1 The association of YATTA and the Relational Model Constructs

Constructs of the relational model are relations, attributes, a primary key for each relation and foreign keys. Here we use the definition of relational model constructs given in [MP99b]. The schemes of the constructs as defined in Section [MP99b] are shown in Table 2. Note that the extent of relation is represented by its primary key attributes.

The construct of the YATTA model, a YATTA node $\langle\langle a, v, t, c\rangle\rangle$, can be mapped to each construct of the relational model according to the rules in Table 3. According to Rule 1, YATTA nodes of complex type can be defined as relations in the relational model. A primary key of the relation is the child nodes of the YATTA node, whose associated values determine the YATTA node's identifier.

In Rule 2, YATTA nodes of atomic type are attributes in the relational model. The cardinality

constraints on attributes are determined by the cardinality constraints of the YATTA node and its parent. If the cardinality constraint of the YATTA node is either '$*$' or '?', the cardinality constraint of the attribute is {1},{1..N}. Otherwise, the cardinality constraint of the attribute is {0,1},{1..N} unless the cardinality constraint of the parent of the YATTA node is '$*$' or '?'.

In Rule 3, the YATTA node $\langle\langle a, v, \text{atomic}, c\rangle\rangle$ is a foreign key in a relation s_v if c=' ' ,and v is a primary key whose values determined its parent, and there exists some siblings of v which is of complex type and their values are determined by v.

5.2 Examples of Transforming between YATTA and the Relational Model

To illustrate how the rules can be used to derive the relational model from the YATTA model, the rules are applied to the YATTA model of S_2, Figure 5.

The scheme of the nodes in the YATTA model of S_2 is as follows:

YATTA node	Scheme
course	⟨⟨root, course, list, $*$⟩⟩
code	⟨⟨root.course, code, String, -⟩⟩
student	⟨⟨root.course, student, list, +⟩⟩
name	⟨⟨root.course.student, name, String, -⟩⟩
sex	⟨⟨root.course.student, sex, String, ?⟩⟩
year	⟨⟨root.course.student, year, Integer, ⟩⟩
grade	⟨⟨root.course.student, grade, String, ⟩⟩

The course node in the YATTA model of S_2 is of complex type and its values are determined by the values associated to the YATTA nodecode. According to Rule 1, the course node is represented as a relation course with a primary key code.

In the YATTA model of S_2, the student node is also of complex type. As can be seen from the YATTA data, the values of student, integer identifiers, depend on the values of code and name. From Rule 1, we get a relation student with a primary key code and name. Since the values the atomic node code determine the values of student which is its sibling , the attribute code is a foreign key of the relation student according to Rule 3.

The atomic nodes sex, year and grade are attributes of the relation student according to Rule 2 and their cardinality constraints are {0,1},{1..N} for sex and {1},{1..N} for year and grade. Since the cardinality constraints of the sex node and its parent are ' ' and '+', the cardinality constraints on the sex attribute is {0,1},{1..N}. The cardinality constraints of the year, grade, and their parent are ' ' and '+', therefore the cardinality constraints on the attribute year and grade are {1},{1..N}.

As a result, we get the relational model S_2^R below. In S_2^R, we change the name of the relation student to cs to be more descriptive.

S_2^R : course(<u>code</u>)

cs(<u>code</u>, <u>name</u>, sex, year, grade)

The schema S_2^R contains two relations: course and student. The course relation holds course records identified by the attribute code. The student relation holds records of students who attended the courses. The key attribute of cs relation are code and name. The attribute code in the cs relation is a foreign key, referencing the key attribute code in the courses relation. The other attributes in the cs relation are sex, year and grade which indicate the sex of student, the year student attended the course and the grade student obtained.

Note that S_2^R is a canonical schema which gets from the YATTA model of S_2. It does not guarantee a good database design. However, the process of normalising the canonical schema can be done by applying transformation rules described in the next section.

The transformations from the YATTA schema of S_2, S_2^Y, to the schema S_2^R can be defined by replacing the construct of the YATTA model with those of the relational model as follows:

Pathway $S_2^Y \to S_2^R$

⑰ addRel(⟨⟨course,code⟩⟩, $\{x \mid \langle y, x\rangle \in$ ⟨⟨root.course, code⟩⟩$\}$)
⑱ addRel(⟨⟨cs,code,name⟩⟩, $\{\langle x, y\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge \langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, name⟩⟩$\}$)
⑲ addAtt(⟨⟨cs,sex,{0,1},{1..N}⟩⟩, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge \langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, sex⟩⟩$\}$)
⑳ addAtt(⟨⟨cs,year,{1},{1..N}⟩⟩, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge \langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, year⟩⟩$\}$)
㉑ addAtt(⟨⟨cs,grade,{1},{1..N}⟩⟩, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge \langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, grade⟩⟩$\}$)
㉒ deleteYattaNode(⟨⟨root.course.student, grade, string, ⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,grade⟩⟩$\}$, after(⟨⟨root.course.student, year⟩⟩))
㉓ deleteYattaNode(⟨⟨root.course.student, year, integer, ⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,year⟩⟩
$\wedge x = \mathrm{generateId}(\langle w, z\rangle)\}$, after(⟨⟨root.course.student, sex⟩⟩))
㉔ deleteYattaNode(⟨⟨root.course.student, sex, integer,?⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,sex⟩⟩
$\wedge x = \mathrm{generateId}(\langle w, z\rangle)\}$, after(⟨⟨root.course.student, name⟩⟩))
㉕ deleteYattaNode(⟨⟨root.course.student, name, string,-⟩⟩, $\{\langle x, y\rangle \mid \langle z, y\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \mathrm{generateId}(\langle z, y\rangle)\}$, any))
㉖ deleteYattaNode(⟨⟨root.course, student, list,+⟩⟩, $\{\langle x, y\rangle \mid \langle u, v\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \mathrm{generateId}(v) \wedge y = \mathrm{generateId}(\langle u, v\rangle)\}$, after(⟨⟨root.course, code⟩⟩))
㉗ deleteYattaNode(⟨⟨root.course, code, string,-⟩⟩, $\{\langle x, y\rangle \mid \langle y, w\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \mathrm{generateId}(y)\}$, any)
㉘ deleteYattaNode(⟨⟨root, course, list,∗⟩⟩, $\{\langle \&0, x\rangle \mid \langle y, w\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \mathrm{generateId}(y)\}$)

As can be seen from the pathway, the constructs of S_2^Y are replaced by the constructs of S_2^R.

In the transformation ⑰-㉑, the constructs of S_2^R are added into S_2^Y and their instances are the extents of equivalent constructs in S_2^Y.

In the transformation ⑰ the course relation with a primary key attribute code is added into S_2^Y. The values of the relation are the values of its key code which are equal to those of code node in S_2^Y.

Transformation ⑱ adds the cs relation and its primary key attributes, code and name into S_2^Y. The values of the relation are the values of its primary key which in this case are the combinations of instances of code and name node. Since the parent of name is the sibling of code and is of complex type, code is a foreign key in the cs relation referencing code attribute in course.

Transformation ⑲ ⑳ and ㉑ add attributes sex, year and grade into the new relation cs, the values of sex, year and grade are the values of the YATTA node sex, year and grade in S_2^Y respectively.

Once every construct on S_2^R appears in S_2^Y, all constructs of S_2^Y are removed.

In the transformation ㉒-㉕, the nodes grade, year sex and name are deleted from S_2^Y. The values of these nodes can be restored from the values of attributes grade, year, sex and name in the cs relation.

Transformation ㉖ removes the student node. Its values can be restored using the function generateId which returns the identifiers for the distinct values of code and name in the cs relation.

Transformation ㉗ and ㉘ deletes code and course node respectively. The values of code can be restored from code attribute in the cs relation whereas the values of course can be restored using the function generateId to get the identifiers of course for every distinct values of code attribute.

The reversibility of transformation on the YATTA and relational model allows the relational model to be automatically transformed back to the YATTA model. A pathway for the mapping from S_2^R back to S_2^Y can be automatically derived as below:

Pathway $S_2^R \rightarrow S_2^Y$

❷❽ addYattaNode(⟨⟨root, course, list,∗⟩⟩, $\{\langle \&0, x\rangle \mid \langle y, w\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \text{generateId}(y)\}$)

❷❼ addYattaNode(⟨⟨root.course, code, string,-⟩⟩, $\{\langle x, y\rangle \mid \langle y, w\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \text{generateId}(y)\}$, any)

❷❻ addYattaNode(⟨⟨root.course, student, list,+⟩⟩, $\{\langle x, y\rangle \mid \langle u, v\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \text{generateId}(v) \wedge y = \text{generateId}(\langle u, v\rangle)\}$, after(⟨⟨root.course, code⟩⟩))

❷❺ addYattaNode(⟨⟨root.course.student, name, string,-⟩⟩, $\{\langle x, y\rangle \mid \langle z, y\rangle \in$ ⟨⟨cs,code,name⟩⟩
$\wedge x = \text{generateId}(\langle z, y\rangle)\}$, any))

❷❹ addYattaNode(⟨⟨root.course.student, sex, integer,?⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,sex⟩⟩
$\wedge x = \text{generateId}(\langle w, z\rangle)\}$, after(⟨⟨root.course.student, name⟩⟩))

❷❸ addYattaNode(⟨⟨root.course.student, year, integer, ⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,year⟩⟩
$\wedge x = \text{generateId}(\langle w, z\rangle)\}$, after(⟨⟨root.course.student, sex⟩⟩))

❷❷ addYattaNode(⟨⟨root.course.student, grade, string, ⟩⟩, $\{\langle x, y\rangle \mid \langle w, z, y\rangle \in$ ⟨⟨cs,grade⟩⟩$\}$,
after(⟨⟨root.course.student, year⟩⟩))

❷❶ deleteAtt(⟨⟨cs,grade,{1},{1..N}⟩⟩, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge\langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, grade⟩⟩$\}$)

❷⓿ deleteAtt(⟨⟨cs,year,{1},{1..N}⟩⟩, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in$ ⟨⟨root.course, code⟩⟩
$\wedge\langle w, u\rangle \in$ ⟨⟨root.course, student⟩⟩ $\wedge \langle u, y\rangle \in$ ⟨⟨root.course.student, year⟩⟩$\}$)

⓳ deleteAtt($\langle\langle$cs,sex,{0,1},{1..N}$\rangle\rangle$, $\{\langle x, y, z\rangle \mid \langle w, x\rangle \in \langle\langle$root.course, code$\rangle\rangle$
$\wedge \langle w, u\rangle \in \langle\langle$root.course, student$\rangle\rangle$ $\wedge \langle u, y\rangle \in \langle\langle$root.course.student, sex$\rangle\rangle \})$

⓲ deleteRel($\langle\langle$cs,code,name$\rangle\rangle$, $\{\langle x, y\rangle \mid \langle w, x\rangle \in \langle\langle$root.course, code$\rangle\rangle$
$\wedge \langle w, u\rangle \in \langle\langle$root.course, student$\rangle\rangle$ $\wedge \langle u, y\rangle \in \langle\langle$root.course.student, name$\rangle\rangle \})$

⓱ deleteRel($\langle\langle$course,code$\rangle\rangle$, $\{x \mid \langle y, x\rangle \in \langle\langle$root.course, code$\rangle\rangle \})$

In our example, the ordering information in the YATTA model is ignored. However, if the ordering information needs to be preserved in the relational model, an additional attribute order is added into the relational model. The values of the attribute order indicate where a YATTA node appears under its parent node. Similary, the occurrence information in the YATTA node ,*i.e.* YATTA nodes are of bag type, can be preserved by the use of an attribute count which indicates how many occurences of a YATTA node are present.

Here we only show the transformations between a YATTA model and a relational model of the same source. The transformations between a YATTA model and a relation model of different sources which are related can be defined in a similar fashion.

5.3 A Methodology for producing the normalised YATTA-Relational schema

Although we can transform from the YATTA model to the relational model by replacing the construct of the YATTA model with those of the relational model, the transformation does not always give a 'good' relation schema. As illustrated in the previous section, in the canonical schema S_2^R which gets from the YATTA model of S_2, the values of name and sex are repeated in several tuples of the student relation. The redundancy arises for the reason that the sex attribute is functionally dependent on the name. Such reduncancy problems can be addressed by decomposing a relation schema into relation schemas according to functional dependency of the relation. The decomposition also allows a canonical schema to be transformed into a good designed schema.

Given a relation R with a set of attributes A and a set of funtional dependencies on R, FD_R, a decomposition of R which is its replacement by a collection of $R_1, R_2, ...R_n$ of subsets of R such that $R = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, can be derived by

1. Apply a union inference rule $\{X \to Y, X \to Z\} \models X \to YZ$ on FD_R
2. For each functional dependency $f : X \to Y$ where $X, Y \in A$ from Step 1,

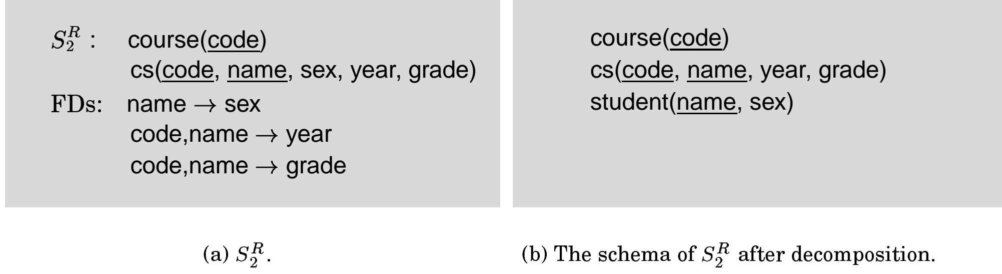
   if X is a non-key attribute or X is a part of a primary key of R, then

   (a) set up a new relation with a primary key X and a non-key attribute Y

   (b) remove an attribute Y from R but leave X in R

The methodology ensures that the decomposition is lossless *i.e.* $R = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ since every key attribute of new relations always appears in R.

A correspondence of the transformations on the relational model to this decomposition technique is as follows:



(a) S_2^R. (b) The schema of S_2^R after decomposition.

Figure 8: The relational schemas of S_2.

In the Step 2, for each functional dependency $f : X \rightarrow Y$ where $X, Y \in A$,

if X is a non-key attribute or X is a part of a primary key of R, then

(a) $\mathsf{addRel}(\langle\langle R_i, X\rangle\rangle, \{u \mid u \in X\})$

$\mathsf{addAtt}(\langle\langle R_i, Y\rangle\rangle, \{\langle u, v\rangle \mid u \in X \wedge v \in Y\})$

(b) $\mathsf{delAtt}(\langle\langle R, Y\rangle\rangle, \{\langle u, v\rangle \mid u \in Z \wedge \langle w, v\rangle \in \langle\langle R_i, Y\rangle\rangle\})$

where Z is a primary key of R

To illustrate the methodology, we show how a good design of S_2^R, Figure 8(b), is derived. Recall the schema S_2^R with its functional dependencies, Figure 8(a), the relation cs is decomposed to the relation student and cs using the methodology as shown below:

1. By applying a union inference rule of FDs, we get the following functional dependencies:

   f_1 : name → sex

   f_2 : code,name → year, grade

2. From Step 2 in the methodology, we check each functional dependency.

   - In f_1, name is a part of a primary key of S_2^R. Hence we define the transformations:

     $\mathsf{addRel}(\langle\langle \mathsf{student,name}\rangle\rangle, \{x \mid x \in \langle\langle \mathsf{cs,name}\rangle\rangle\})$
     $\mathsf{addAtt}(\langle\langle \mathsf{student,sex}\rangle\rangle, \{\langle x, y\rangle \mid x \in \langle\langle \mathsf{cs,name}\rangle\rangle \wedge y \in \langle\langle \mathsf{cs,sex}\rangle\rangle\})$
     $\mathsf{delAtt}(\langle\langle \mathsf{cs,sex}\rangle\rangle, \{\langle x, y, z\rangle \mid \langle x, y\rangle \in \langle\langle \mathsf{cs,code,name}\rangle\rangle \wedge z \in \langle\langle \mathsf{cs,sex}\rangle\rangle\})$

   - In f_2, code and name is a primary key so we do nothing.

6 Conclusion

In this report, a transformation-based approach for integrating semistructured data has been introduced. It has shown that semistructured data sources can be naturally represented using a new model, YATTA. The YATTA model facilitates the integration by clearly distinguishing the schema from the data, and providing information on the order, type and cardinality of the data.

The approach has focused on providing a global view of semistructured data sources to the user. By applying the BAV approach, a set of primitive transformation rules to manipulate YATTA has been defined. The integration of two heterogeneous sources using a sequence of primitive transformations, a pathway, has been illustrated. Since each primitive transformation has an automatically derivable reverse transformation, pathways are bi-directional. This allows data and query to be automatically migrated in either directions between the global view and the source models linkded by a pathway. The evolution of data sources and the global view can be easily handled by incrementally extending and modifying a sequence of transformations to account for recent changes.

The transformation between the YATTA model and the relational model has also been presented. It has shown that the transformation allows semistructured data sources to be integrated with the data stored in the relational model. With the transformation between the YATTA model and the relational model, the migration of semistructured data to relational databases can be automatically processed.

References

[BDFS97] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350, Delphi, Greece, 8–10 1997. Springer.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 177–188. ACM Press, 1998.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.

[Kit02] S. Kittivoravitkul. Models for semistructured data integration. Technical report, Imperial College London, UK, January 2002.

[MP99a] P.J. Mcbrien and A. Poulovassilis. Automatic migration and wrapping of database applications - a schema transformation approach. In *Proceedings of ER99*, volume 1728 of LNCS, pages 96–113. Springer-Verlag, 1999.

[MP99b] P.J. Mcbrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99*, volume 1626 of LNCS, pages 333–348. Springer-Verlag, 1999.

[MP02] P.J. Mcbrien and A. Poulovassilis. Schema evolution in heterogeneour database architectures, a schema transformation approach. In *Advanced Information Systems Engineering, 14th International Conference CAiSE2002*, volume LNCS. Springer-Verlag, 2002.

[MP03] Peter McBrien and Alexandra Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proceeding of 19th IEEE Internaltional Conference on Data Engineering ICDE'03*, pages 227–238, 2003.

[Pet96] Peter Buneman and Susan Davidson and G. Hillebrand and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data, Jun 1996.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[SBS00] S.Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.