From Proc. CAiSE05
LNCS 3520, Pages 460-474

Integrating Unnormalised Semi-Structured Data Sources

Sasivimol Kittivoravitkul and Peter McBrien

Department of Computing, Imperial College London, London SW7 2AZ
sk297@doc.ic.ac.uk, pjm@doc.ic.ac.uk
http://www.doc.ic.ac.uk/automed

Abstract. Semi-structured data sources, such as XML, HTML or CSV files, present special problems when performing data integration. In addition to the hierarchical structure of the semistructured data, the data integration must deal with the redundancy in semi-structured data, where the same fact may be repeated in a data source, but should map into a single fact in a global integrated schema. We term semi-structured data containing such redundancy as being an unnormalised data source, and we define a normal form for semi-structured data that may be used when defining global schemas. We introduce special functions to relate object identifiers used in the global data model to object identifiers in unnormalised data sources, and demonstrate how to use these functions in query processing, update processing and integration of these data sources.

1 Introduction

Areas of application development such as the WWW, electronic commerce, bioinformatics and other scientific disciplines, have led to a growing demand for data representations that support complex, nested and rapidly evolving structures. Often applications in these areas use a **semistructured data** (**SSD**) data model, such as XML, HTML or one of a variety of flat-file formats (including CSV and TSV). With the proliferation of distributed and heterogeneous SSD, there is a clear need for techniques to perform data integration over these SSD sources, and provide a global unified view of the data.

One of the main tasks in data integration is to define the **mappings** between individual data sources and the unified global view of those sources. Two basic approaches for specifying this mapping are **global-as-view (GAV)** and **local-as-view (LAV)** [10]. The former approach defines the concepts in the global schema as views over the local source schemas whereas the latter approach defines the sources as views over the global schema. Recently, a new approach called **both-as-view (BAV)** [13] has been proposed that specifies a bi-directional mapping between each source and the global schema. Such bi-directional mappings allow data and queries to be translated in either direction from the global schema to the sources, and *vice versa*. This is important, for example, when integrating data in peer-to-peer contexts [14]. The use of BAV has been investigated in the integration of structured data sources [11], and some work has been carried out on integrating XML data sources [12, 20]. The work on the XML integration has concentrated on specifying schema relationships and made strong assumptions about

```
⟨root⟩
  ⟨student⟩
    ⟨name⟩Ann⟨/name⟩
    ⟨tutor⟩Peter⟨/tutor⟩
    ⟨course code="DB"⟩
      ⟨dept⟩CS⟨/dept⟩
      ⟨lecturer⟩Simon⟨/lecturer⟩
      ⟨year⟩1⟨/year⟩
      ⟨grade⟩A⟨/grade⟩
    ⟨/course⟩
    ⟨course code="Stat"⟩
      ⟨dept⟩MA⟨/dept⟩
      ⟨lecturer⟩Jane⟨/lecturer⟩
      ⟨year⟩2⟨/year⟩
      ⟨grade⟩C⟨/grade⟩
    ⟨/course⟩
  ⟨/student⟩
  ⟨student⟩
    ⟨name⟩Mark⟨/name⟩
    ⟨tutor⟩Alex⟨/tutor⟩
    ⟨course code="DB"⟩
      ⟨dept⟩CS⟨/dept⟩
      ⟨lecturer⟩Simon⟨/lecturer⟩
      ⟨year⟩1⟨/year⟩
      ⟨grade⟩B⟨/grade⟩
    ⟨/course⟩
  ⟨/student⟩
⟨/root⟩
```

(a) F_1 XML file of undergraduates

```
[DB,CS,Ace]
    student=Ann
        level=UG
        year=1
        grade=A
[OS,CS,Ace]
    student=Mark
        level=UG
        year=2
[Stat,MA,Biet]
    student=Ann
        level=UG
        year=2
        grade=C
    student=Mary
        level=PG
        year=4
        grade=A
```

(b) F_2 Text file of students

Fig. 1. Example unnormalised semistructured data sources

the data. In particular, they have assumed that there is no redundancy in the data to be integrated.

Fig. 1(a) and 1(b) illustrate two SSD sources which both contain a degree of redundancy, and which overlap with each other. Since they contain redundancy we call them **unnormalised** SSD sources — more precisely we regard anything with less than the SSD equivalent of second normal form as unnormalised. Fig. 1(a) contains an XML file F_1 with details of undergraduate students, where each course a student is sitting is placed within the student element. For each course, there is a record of the department that manages the course, the lecturer of that course, and the year of study and grade that the student has achieved in the course. Note that there is redundancy in this SSD, since the fact that the CS department runs the DB lectured by Simon is repeated for the two occurrences of that course.

Fig. 1(b) illustrates a structured text file F_2 containing information of courses taken by undergraduate and postgraduate students. The information in Fig. 1(b) is similar to that of the source in Fig. 1(a), but it is structured in different way *i.e.* student information is nested within course information, and provided information about the department building, which is not in F_1. It avoids the redundancy of F_1, in that the department of each course is only recorded once, but has its own redundancy in that the level of a student (UG or PG) is repeated for each course a student sits.

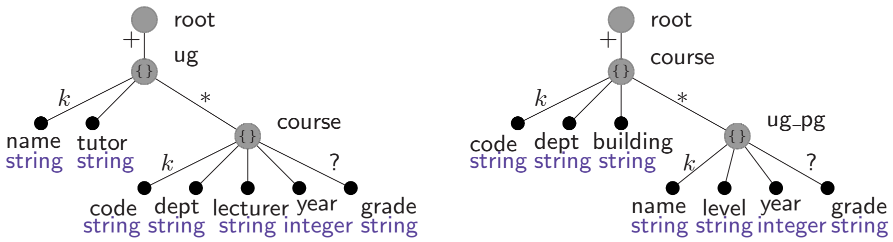
When integrating F_1 and F_2 we have to transform at least one of the files, and in particular deal with inverting the hierarchy present in one file to match that of the other. For example, if we choose in our global schema to model courses as containing multiple students (*i.e.* the F_2 view of the data) then when we transform F_1 we would want to have just one course department pair produced for each distinct code and dept pairing that exists in F_1: *i.e.* produce a set of records containing just two courses, DB and Stat, with two students under the first, and one under the second. Further, for each course, we would only want to have one dept value, but maintain the multiple year and dept associated with each student.

In this paper, we extend the BAV approach to the integration of SSD sources by relaxing assumptions or conditions in the preliminary work *i.e.* allowing unnormalised SSD sources to be integrated. The highlight of our approach is a semistructured data model which includes the notion of key constraints, and a mechanism to deal with redundancy of the data that allows data to be correctly translated in both directions, that is from sources to the global schema and *vice versa*. In contrast, most previous work in semistructured data translation/transformation [3, 5, 15] has focused on retrieving data from data sources to the global schema, and only defined a one way mapping where data can be migrated from source to global schemas, but not vice versa. Popa *et al.* [16, 17] proposed a framework that semi-automatically generates invertible mappings between a relational source and an equivalent nested XML schema. Their mappings are not strictly invertible, since when there is more than one source, the reverse mapping gives back not only an original data but also information acquired by other sources. To our knowledge, no previous work specifically deals with normalisation of SSD sources in the context of data integration. Also, the subject of invertible mappings in SSD integration context has not been fully addressed.

The paper is structured as follows. Section 2 reviews a SSD modelling language called YATTA we shall use for representing SSD data sources. We use the YATTA model as basis for defining normal forms of SSD in Section 3, based on the well known notions of normal forms in relational models. This notion of second normal form is used as a basis for specifying our mappings using the keys in Section 4. It should be noted that it is only the global schema that must obey the second normal form, and the sources may remain unnormalised if we want to take a simple union of data, or they must be in the SSD equivalent of first normal form if actual *data* (as opposed to schema) integration is to take place. Section 5 shows how queries and updates are processed using this mapping approach. Our summary and conclusions are in Section 6.

2 The YATTA Data Model

We adopt the **YAT for Transformation-based Approach** (**YATTA**) model [2] to model SSD sources. YATTA is a variation of the YAT model [4] that has two levels of abstraction, called the **schema level** the **data level**.



(a) YATTA schema S_1 (b) YATTA schema S_2

Fig. 2. The YATTA schemas of the XML and text file Fig. 1(a) and 1(b)

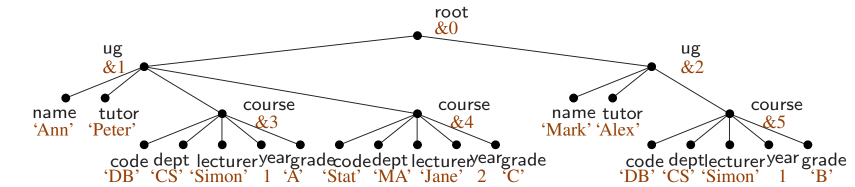
A YATTA **schema** represents the structure of a SSD source. Each node in a YATTA schema is labelled with a pair of strings, representing its name and data type. The data type for a leaf node is one of the atomic types which are string, integer and real whereas the data type for a non-leaf node is of type list or set, represented by '[ ]' and '{}'. Fig. 2(a) and 2(b) represent YATTA schemas for the XML and text files in Fig. 1(a) and 1(b).

Each YATTA schema edge between nodes $\langle i,j \rangle$ can be labelled with a cardinality constraint that determines the number of times corresponding nodes j may occur under each node i in a YATTA data tree, where '$*$' indicates zero or more occurrences, '+' indicates one or more occurrences, '?' indicates zero or one occurrence, and label '1' indicates exactly one occurrence (and is implied if the edge is unlabelled). The symbol 'k' on an edge indicates that a j is a **key node**, the values of which must be distinct from the values of other j nodes that appear as siblings of i. Hence k also implies the '1' constraint. In S_1, the key node of the course node is code, and says that each course for a particular ug will have a distinct code value.

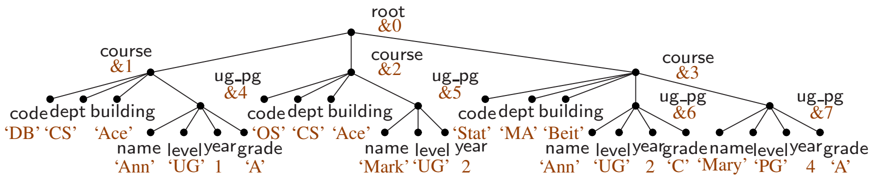
Unlike a YATTA schema, a YATTA **data tree** has no labels on its edges. Each node is labelled by a tuple representing its name and value. The values of leaf nodes are the actual data in a data source whereas the values for non-leaf nodes are assigned by the system using **integer identifiers** (denoted by '&' followed by a number *e.g.* '&0'). The root node is always named root, and its identifier differentiates between particular source files (such as D_1) that obey a schema (such as S_1).

Fig. 3(a) and 3(b) illustrate YATTA data trees of the XML and text files, and which match the YATTA schemas S_1 and S_2. For each data tree node there is a corresponding schema node with the same path, such that the data tree node value is compatible with the data type of that schema node. For example, the path $\langle\langle$root,ug,course,grade$\rangle\rangle$ in D_1 leads to three grade data nodes. We will describe the extent of such nodes by listing their values along with the identifiers of the parent, and hence $\langle\langle$root,ug,course,grade$\rangle\rangle$ gives [{&3,'A'},{&4,'C'},{&5,'B'}]. In the schema, the same path leads to a simple string type node, which matches the type of the second value in each of the tuples in the extent list.

The occurrences of the data node follow the cardinality specified by the symbols on the incoming edges of the corresponding node in the schema. For example, in Fig. 3(a),



(a) D_1 The data tree of the XML file



(b) D_2 The data tree of the text file F_2

Fig. 3. Examples YATTA data trees for Fig. 1

each student has exactly one tutor, as specified by implied '1' on the incoming edges of tutor in S_1, and each code of course only exists once for a particular undergraduate student which is also defined name as a key node. The label 'k' on the incoming edges of name and code in S_1 means the two ug nodes always have different name and no undergraduate student may take two courses with the same code.

3 Normal Forms for Semistructured Data Sources

In practice, many SSD sources have the property that each subrecord has a distinguishing attribute or set of attributes that uniquely identifies the subrecord of a given record. We now formalise this idea into the notion of **normal forms** for SSD that may be used when defining global schemas to help avoid data redundancy, inconsistency and undesirable updating anomalies in the integration.

Normal forms have been extensively investigated in the relational model [6] and have recently been extended to SSD [1, 8, 19]. Both [1] and [8] defined a normal form for XML, called XNF, but the two approaches differ. [1] proposed the concept of functional dependency for XML, and defined BCNF for XML documents. In [8], an XML document is in XNF if its specification does not contain potential redundancy w.r.t. a specified set of constraints. Their definition is comparable to the requirement of 3NF in the relational model. [19] defined a normal form for SSD represented in the XML model, called NS-SS, which appears to be analogous to BCNF. In order to define NS-SS, they introduced the concept of 'extended functional dependency', which extends functional dependency in the relation model to support hierarchical data, and the notion of key constraints.

The BCNF or 3NF proposed by [1, 8, 19] gives a well-designed data source, but also increase the complexity in accessing a data source, and the use of SSD models is to achieve flexibility *i.e.* not too rigid design. Thus we work with weaker normal forms, comparable to first and second normal forms in the relational model, which are the minimum to achieve data integration.

In the relational model, **first normal form** (**1NF**) states that each attribute of a relation is functionally determined by its key value, and implies that each relation has a key, and that the non-key attributes take single values. We will define 1NF in SSD as saying that each non-root, non-leaf node in the schema contains at least one key child leaf node (*i.e.* there is at least one k beneath each non-leaf node), and that all leaf nodes do not use * or + cardinality constraints. This means that the non-key leaf nodes can be identified by combining all the key nodes in the path to the non-key leaf node. Schema S_1 obeys our 1NF, and this means we can identify each ug and its tutor leaf node by name values, and we can identify each course and its dept, lecturer, year and grade leaf nodes, by the combination of its key node values *i.e.* code and name values. Note that in order to truely integrate *data*, as opposed to just the schema holding the data, sources must be in 1NF, since that allows us to identify data values by a **natural key** (**NK**) (*i.e.* a set of values from the real world) rather than the **artificial key** (**AK**) (*i.e.* object identifiers) used by the system.

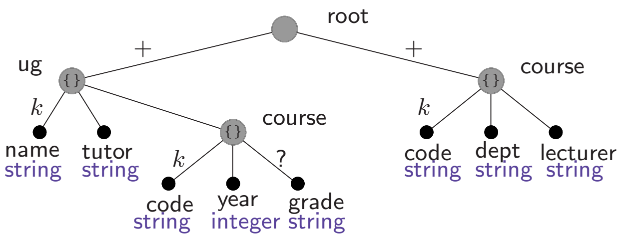
In the relational model, **second normal form** (**2NF**) states that each attribute of a relation is functionally determined by primary key, but not by any proper subset of the key. In the YATTA model, this corresponds to the schema being in 1NF, with the additional constraint that all of the key nodes are necessary to determine each non-key node. S_1 is not in 2NF, since dept and lecturer are determined by code alone, and not code and name combined.

We can **normalise** schema S_1 to that shown in Fig. 4, by forming a copy of the course under the root, with just those non-key nodes which are determined by code alone being moved to the new course node, the remaining nodes staying as they were in S_1.

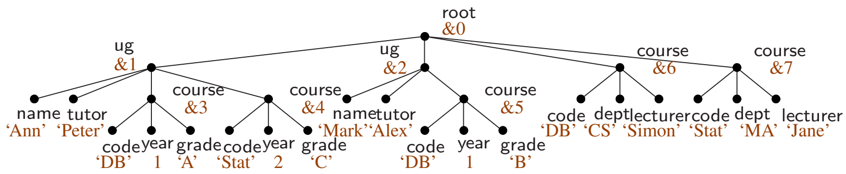
In this paper, we call sources not in the 2NF, **unnormalised** data sources. As in the relational model, the redundancy in unnormalised SSD sources leads to difficulties with updates, and in integration also leads to inefficiencies since the mapping tables based on the keys will contain redundant information. For example, to update the department for the 'DB' course from 'CS' to 'MA' in Fig. 3(a), we are faced with either the problem of searching the tree to find every course containing 'DB' and 'CS' (and changing it) or the possibility of producing an inconsistent result, for example that the department for 'DB' might be given as 'MA' in one record and 'CS' in another. The next section explains how we write mapping rules that take account of this redundancy.

4 Mapping Semistructured Data Sources Using Natural Keys

The BAV approach integrates data sources by transforming source schemas into a global schema through sequences of transformations called **pathways**. Each transformation makes a 'delta change' to a schema, adding, deleting, or renaming a single schema node. Each transformation contains a query that specifies the instances of a node in the



(a) S_1 in 2NF



(b) The corresponding YATTA data tree

Fig. 4. The 2NF of YATTA source in Fig. 3(a)

corresponding data tree. We use the BAV transformation rules for the YATTA model that are defined in [2].

Suppose we want to integrate the 1NF SSD sources in Fig. 1(a) and 1(b). First we design a global schema S_g, such as that in Fig. 5, and then give mapping rules that define how each node of a global schema can be defined from the source schemas. Based on the approach described in [2], when adding or deleting a node, we describe a scheme for the node which contains the pathway to the node, the type of the node, and the cardinality of the edge that leads to the node from its parent. For example, the scheme of the student node in S_g is $\langle\langle \mathsf{root,student,set,+} \rangle\rangle$ and the name is $\langle\langle \mathsf{root,student,name,string},k \rangle\rangle$.

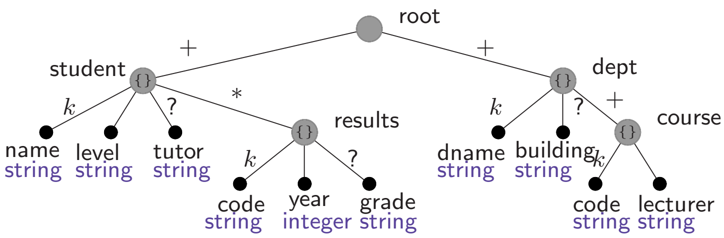


Fig. 5. The global schema S_g

The schemas S_1 and S_2 are transformed into S_g by applying a pathway of YATTA transformation rules, where each pathway consists of a **growth phase** in which nodes in S_g that do not exist in the source schema are added, followed by a **shrinking phase** in which nodes that exist in the source schema but not in $\mathsf{S_g}$ are deleted. When a new node is added in the source schema, the query specifies how the instances of a node in the corresponding data tree should be populated. When an existing node is deleted in a schema, the query specifies how the instances of the node in the corresponding data tree can be restored from the remaining nodes. The mapping of the data sources to the global schema is therefore specified through the queries in a pathway.

To ensure that source data is correctly mapped with the global schema, the transformations must take into account the following issues:

Identifier Conflicts Different data sources might be given different identifiers for the nodes representing the same thing, and the same identifier might be given for different things. For example, the identifier '&1' is assigned to the student node in $\mathsf{S_1}$ in Fig. 3(a) and the course node in $\mathsf{S_2}$ in Fig. 3(b). Hence, in specifying the mapping, a mechanism to resolve these conflicts is required.

Hierarchical structure The global schema might be structured in a different way from the source schemas, as shown in the examples. In this case, the mapping therefore involves preserving the relationships between data elements that are implied by the hierarchical structure of the data source.

To resolve identifier conflicts, we apply the concept of a **surrogate keys** (**SK**) [7], which provides a way of mapping between a NK used in the real world, and an AK used by the system. This mapping is realised using two functions generateGID and generateSID, which are used in the queries of the add and delete transformations, respectively. The functions use the data values (NKs) to associate source identifiers $sids$ (AKs) with global identifiers $gids$, which are generated as surrogates.

The generateGID function takes a source schema name, a sid, a list of data values and the name of a node that the transformation applied to. The function returns a new OID (gid) for every distinct list of data values, and returns the same gid for the same list of data values. Hence each gid serves as a surrogate for some set of data values. When a global schema is in 2NF, these set of data values are the key values of the node that a transformation applied to.

Conversely, the generateSID takes a source schema name, a list of data values and the name of a node in the global schema. The function returns a source OIDs (sid) that have been used in generating gid for the same global schema node. The generateSID function can be thought of as the reverse of the generateGID function in the sense that generateGID takes sid in the source and generates gid for the global view whereas generateSID returns sid of the source.

For these functions to be applied in the pathway, the data sources and global schema must be at least in 1NF. This allows the key values in the different sources to be mapped to those of the global schema, thereby allowing data from those sources to be combined. If data sources or global schema are not in 1NF, the integration will just take the union of the respective sources, which is not a real data integration.

To illustrate the generateGID and generateSID functions, we explain how they are used in the pathways $\mathsf{S_1}\rightarrow\mathsf{S_g}$ and $\mathsf{S_2}\rightarrow\mathsf{S_g}$, an extract from which is shown below. Note that $\mathsf{S_1}$ and $\mathsf{S_2}$ are in 1NF whereas $\mathsf{S_g}$ is in 2NF to avoid redundancy in the integrated data. The student node in $\mathsf{S_g}$ is created by the add transformations 1 in $\mathsf{S_1}\rightarrow\mathsf{S_g}$ and 5 in $\mathsf{S_2}\rightarrow\mathsf{S_g}$.

$S_1 \rightarrow S_g$

① addYattaNode(⟪root,student,set,+⟫, [{r, generateGID(S_1, u, [n], 'student')} |
{r, u} ← ⟪root, ug⟫; {u, n} ← ⟪root, ug, name⟫)])

② addYattaNode(⟪root,student,name,string, k⟫, [{generateGID(S_1, u, [n], 'student'), n} |
{u, n} ← ⟪root, ug, name⟫])

③ addYattaNode(⟪root,student,level,string,1⟫, [{generateGID(S_1, u, [n], 'student'), 'ug'} |
{u, n} ← ⟪root, ug, name⟫])

④ addYattaNode(⟪root,student,tutor,string,?⟫, [{generateGID(S_1, u, [n], 'student'), t} |
{u, n} ← ⟪root, ug, name⟫; {u, t} ← ⟪root, ug, tutor⟫])

$S_2 \rightarrow S_g$

⑤ addYattaNode(⟪root,student,set,+⟫, [{r, generateGID(S_2, p, [n], 'student')} |
{r, c} ← ⟪root, course⟫; {c, p} ← ⟪root, course, ug_pg⟫;
{p, n} ← ⟪root, course, ug_pg, name⟫])

⑥ addYattaNode(⟪root,student,name,string, k⟫, [{generateGID(S_2, p, [n], 'student'), n} |
{c, p} ← ⟪root, course, ug_pg⟫; {p, n} ← ⟪root, course, ug_pg, name⟫])

The IQL [18] query in ① finds in the generator {r,u} ←⟪root, ug⟫ the tuples {&0, &1}, {&0, &2}, and {u,n} ←⟪root, ug, name⟫ the tuples {&1, 'Ann'}, {&2, 'Mark'}. Then the generateGID function is called with (S_1,&1,[Ann],'student') and (S_1,&2,[Mark],'student'). The function generates &101 and &102 as new global integer identifiers (*gids*) for each list of the data value, ['Ann'] and ['Mark'], which are the key values of student in S_g. Hence the list [{&0,&101},{&0,&102}] will be associated with ⟪root,student⟫. A similar analysis for transformation ② will give list [{&101,'Ann'},{&102,'Mark'}] being associated with ⟪root,student,name⟫, and so on for the remaining transformations. The mapping of *sids* to *gids* for the student node through the values of the name node, which is the key node of student, is shown in Fig. 6 (though at this stage, the S_2 part of the graph should be ignored).

Similarly, the query in transformation ⑤ finds the tuples: {&0, &1}, {&0, &2}, {&0, &3} from {r,c} ←⟪root, course⟫, then the tuples {&1, &4}, {&2, &5}, {&3, &6}, {&3, &7} from {c,p} ←⟪root, course, ug_pg⟫, and finally the tuples {&4,'Ann'}, {&5,'Mark'}, {&6,'Ann'}, {&7,'Mary'} from {p,n} ←⟪root, course, ug_pg, name⟫. This causes generateGID to receive (S_2,&4,[Ann],'student'), (S_2,&5,[Mark],'student'), (S_2,&6,[Ann],'student') and (S_2,&7,[Mary],'student'). Since the *gids* for [Ann] and [Mark] already exist, the function returns &101 and &102 and generates a new *gid* &103 for [Mary]. Transformation ⑤ gives a list [{&0,&101},{&0,&102},{&0,&103}] being associated with the scheme ⟪root,student⟫.

As illustrated in Fig. 6, the generateGID function groups together the ug and ug_pg nodes related to the same name, and creates a *gid* for each group. This resolves the identifier conflicts among data sources and minimises the data redundancy in the integration. The data that is related to name such as tutor, is also put under the student node related to such name as specified in the query in the transformation ④. The *gids* created by the generateGID function as well as its parameters can be stored as shown in Fig. 7 and 8.

Below are the transformations in the shrinking phase of $S_1 \rightarrow S_g$ that remove the tutor, name and ug nodes in S_1(we omit details of how before these transformations, grade, year, lecturer, dept, code and course are deleted). The generateSID function is

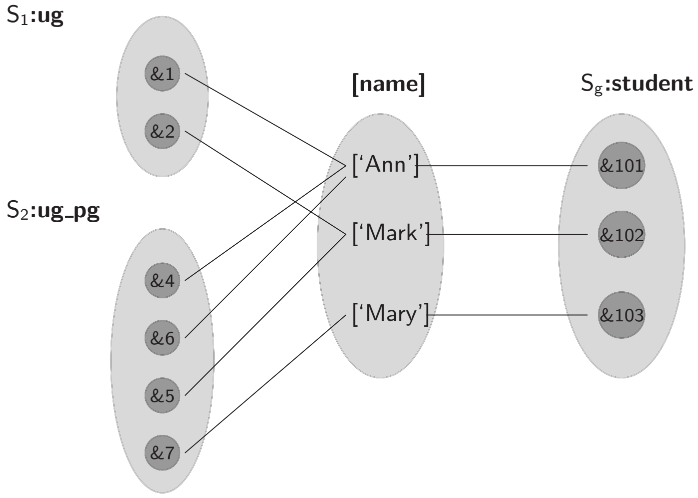


Fig. 6. The mapping between *sids* and *gids* of the student node

local schema	sid	NK	name
S_1	[&1]	['Ann']	student
S_1	[&2]	['Mark']	student
S_2	[&4]	['Ann']	student
S_2	[&5]	['Mark']	student
S_2	[&6]	['Ann']	student
S_2	[&7]	['Mary']	student
.	.	.	.

Fig. 7. *sids* and the data values

gid	NK	name
[&101]	['Ann']	student
[&102]	['Mark']	student
[&103]	['Mary']	student
.	.	.
.	.	.
.	.	.
.	.	.

Fig. 8. *gids* and the data values

applied in this phase to allow the **reverse** transformation to recover the original data. This reversibility ensures *information preservation* in the transformation. Importantly, it allows data, queries and updates to be automatically migrated or translated in either direction between source and the global schemas.

$S_1 \rightarrow S_g$

7 $\mathsf{delYattaNode}(\langle\langle \mathsf{root,ug,tutor,string,1}\rangle\rangle, [\{\mathsf{generateSID}(S_1, [n], \text{'student'}), t\} \mid$
$\{s, n\} \leftarrow \langle\langle \mathsf{root, student, name}\rangle\rangle; \{s, t\} \leftarrow \langle\langle \mathsf{root, student, tutor}\rangle\rangle;$
$\{s, \text{'ug'}\} \leftarrow \langle\langle \mathsf{root, student, level}\rangle\rangle])$

8 $\mathsf{delYattaNode}(\langle\langle \mathsf{root,ug,name,string,}\ k\rangle\rangle, [\{\mathsf{generateSID}(S_1, [n], \text{'student'}), n\} \mid$
$\{s, n\} \leftarrow \langle\langle \mathsf{root, student, name}\rangle\rangle; \{s, \text{'ug'}\} \leftarrow \langle\langle \mathsf{root, student, level}\rangle\rangle])$

9 $\mathsf{delYattaNode}(\langle\langle \mathsf{root,ug,set,+}\rangle\rangle, [\{r, \mathsf{generateSID}(S_1, [n], \text{'student'})\} \mid$
$\{r, s\} \leftarrow \langle\langle \mathsf{root, student}\rangle\rangle; \{s, n\} \leftarrow \langle\langle \mathsf{root, student, name}\rangle\rangle;$
$\{s, \text{'ug'}\} \leftarrow \langle\langle \mathsf{root, student, level}\rangle\rangle])$

Transformation 7 removes the tutor node from ug. The query in the transformation states that the values of tutor in S_1 can be restored from tutor in S_g . It finds the tuples

{&101, ‘Ann’}, {&102, ‘Mark’}, {&103, ‘Mary’} from {s,n} ←⟨⟨root, student, name⟩⟩, then the tuples {&101, ‘Peter’}, {&102, ‘Alex’} from {s,t} ←⟨⟨root, student, tutor⟩⟩, then the tuples {&101, ‘ug’}, {&102, ‘ug’} from {s,‘ug’} ←⟨⟨root, student, level⟩⟩. The generateSID function is called with (S_1,[‘Ann’],‘student’) and (S_1,[‘Mark’],‘student’). The function then looks up the key values in the table of Fig. 7, and restores the values of the ug node in S_1 with &1 and &2, which are the $sids$ related to ‘Ann’ and ‘Mark’ in the source S_1. The queries in ⑧ and ⑨ can be read in a similar manner.

5 Queries and Updates over the Mapping

After defining the mappings of data sources to the global schema, one may want to query or update data sources through the global schema. The bi-directional mappings allow queries and updates posed on the global schema to be automatically translated to ones poses on data sources.

5.1 Query Translation

To translate a query Q_g posed on the global schema to a query on data source S_x, we need only consider delete transformations in the pathway $S_g \rightarrow S_x$. Every deleted construct appearing in the query Q_g are substituted by the query in the transformations. For example, suppose we pose the query Q_1 on $\mathsf{S_g}$ asking for all students, which in the IQL would take the form:

$Q_1 = [\{x, y\} \mid \{x, y\} \leftarrow \langle\langle \mathsf{root, student, name} \rangle\rangle]$

The pathways $\mathsf{S_g}\rightarrow\mathsf{S_1}$ and $\mathsf{S_g}\rightarrow\mathsf{S_2}$ can be automatically derived from $\mathsf{S_1}\rightarrow\mathsf{S_g}$ and $\mathsf{S_2}\rightarrow\mathsf{S_g}$, respectively, by replacing delete for add, and replacing add for delete. Below are the inverse steps ❷–❶ in transformations ①–④, and ❻–❺ in transformations ⑤–⑥.

$\mathsf{S_g} \rightarrow \mathsf{S_1}$
❹ delYattaNode(⟨⟨root,student,tutor,string,?⟩⟩, [{generateGID($\mathsf{S_1}$, u, [n], ‘student’), t} |
{u, n} ← ⟨⟨root, ug, name⟩⟩; {u, t} ← ⟨⟨root, ug, tutor⟩⟩])
❸ delYattaNode(⟨⟨root,student,level,string,1⟩⟩, [{generateGID($\mathsf{S_1}$, u, [n], ‘student’), ‘ug’} |
{u, n} ← ⟨⟨root, ug, name⟩⟩])
❷ delYattaNode(⟨⟨root,student,name,string, k⟩⟩, [{generateGID($\mathsf{S_1}$, u, [n], ‘student’), n} |
{u, n} ← ⟨⟨root, ug, name⟩⟩])
❶ delYattaNode(⟨⟨root,student,set,+⟩⟩, [{r, generateGID(‘student’, u, [n])} |
{r, u} ← ⟨⟨root, ug⟩⟩; {u, n} ← ⟨⟨root, ug, name⟩⟩)])

$\mathsf{S_g} \rightarrow \mathsf{S_2}$
❻ delYattaNode(⟨⟨root,student,name,string, k⟩⟩, [{generateGID($\mathsf{S_2}$, p, [n], ‘student’), n} |
{c, p} ← ⟨⟨root, course, ug_pg⟩⟩; {p, n} ← ⟨⟨root, course, ug_pg, name⟩⟩])
❺ delYattaNode(⟨⟨root,student,set,+⟩⟩, [{r, generateGID($\mathsf{S_2}$, p, [n], ‘student’)} |
{r, c} ← ⟨⟨root, course⟩⟩; {c, p} ← ⟨⟨root, course, ug_pg⟩⟩;
{p, n} ← ⟨⟨root, course, ug_pg, name⟩⟩])

To translate the query Q_1 into source ones, the construct $\langle\langle \mathsf{root}, \mathsf{student}, \mathsf{name} \rangle\rangle$ is replaced by the queries q_1 and q_2 in transformations ❷ in the pathway $\mathsf{S}_1 \rightarrow \mathsf{S}_\mathsf{g}$ and ❻ in the pathway $\mathsf{S}_2 \rightarrow \mathsf{S}_\mathsf{g}$, combined by the OR operator [9].

$$Q_1 = [\{x, y\} \mid \{x, y\} \leftarrow (q_1 \ \mathsf{OR}\ q_2)]$$

The query q_1 returns a list [{&101,'Ann'}, {&102,'Mark'}] whereas the query q_2 gives a list [{&101,'Ann'}, {&102,'Mark'}, {&103,'Mary'}] as described in the previous section. Using union semantics for the OR operator, we therefore get are all students from both data sources:

$$Q_1 = [\{\&101, \text{'Ann'}\}, \{\&102, \text{'Mark'}\}, \{\&103, \text{'Mary'}\}]$$

In this simple example, the generateGID function reduces the redundancy by ensuring that 'Ann' is returned only once. Without the generateGID function, the name 'Ann' would appear three times as a result of different identifiers of student nodes in the data sources.

In general, the generateGID function combines data that requires merging but has different identifiers in the sources (*e.g.* student nodes with identifiers &2 and &5, which are related to 'Mark'), and avoids combining distinct data that has the same identifier in different sources (*e.g.* course nodes with identifiers &2 in D_1 and D_2, which are related to 'Mark' and 'OS', respectively). In addition, the function allows related information in different sources to be brought together by the key values. For example, the information about the department building and the lecturer are in different sources, but by posing the query Q_2 below on S_g,

$$\begin{aligned} Q_2 = [\{x, y, z, w\} \mid\ & \{x, y\} \leftarrow \langle\langle \mathsf{root}, \mathsf{dept}, \mathsf{building} \rangle\rangle; \\ & \{x, c\} \leftarrow \langle\langle \mathsf{root}, \mathsf{dept}, \mathsf{course} \rangle\rangle; \\ & \{c, z\} \leftarrow \langle\langle \mathsf{root}, \mathsf{dept}, \mathsf{course}, \mathsf{code} \rangle\rangle; \\ & \{c, w\} \leftarrow \langle\langle \mathsf{root}, \mathsf{dept}, \mathsf{course}, \mathsf{lecturer} \rangle\rangle] \end{aligned}$$

the query results in a list of the departments, the building names, the course codes, and the lectures for the courses as shown below, since they are joined by the natural keys identifying dept and course.

$$\begin{aligned} Q_2 = [&\{\text{'CS'}, \text{'ACE'}, \text{'DB'}, \text{'Simon'}\}, \{\text{'CS'}, \text{'ACE'}, \text{'OS'}, \text{'Void'}\}, \\ &\{\text{'MA'}, \text{'Beit'}, \text{'Stat'}, \text{'Jane'}\}] \end{aligned}$$

5.2 Update translation

Applying an update U requires giving the scheme of the construct to be updated, and the new value of the construct tuple. For example, to change the tutor which is associated to student with identifer &101 to 'Fred', the update statement can be written as:

$$U_1 = \mathsf{update}(\langle\langle \mathsf{root}, \mathsf{student}, \mathsf{tutor} \rangle\rangle, \{\&101, \text{'Fred'}\})$$

Translating updates posed on a global schema to ones on data sources is different from translating queries described earlier. In update translation, all delete rules in pathways $\mathsf{S}_\mathsf{x} \rightarrow \mathsf{S}_\mathsf{g}$ that use the scheme of the construct to be updated should be retrieved. For the example, only ⑦ in the pathway $\mathsf{S}_1 \rightarrow \mathsf{S}_\mathsf{g}$ matches the criteria for U_1, since its IQL query contains $\langle\langle \mathsf{root,student,tutor} \rangle\rangle$. The data source is then updated by processing the queries accompanying the delete transformation using the new value of the construct tuple. In the example, the query accompanying 7 in the pathway $\mathsf{S_1} \rightarrow \mathsf{S_g}$ is processed using the new value $\{\&101, \text{'Fred'}\}$. The association between the gid &101 and sid &1 is obtained by the generateSID function. The value $\{\&1,\text{'Peter'}\}$ in the $\langle\langle \mathsf{root,ug,tutor} \rangle\rangle$ of $\mathsf{S_1}$ is then updated to $\{\&1,\text{'Fred'}\}$.

Note that there may be multiple source nodes to update for some update statements. For example, with the update on $\mathsf{S_g}$ to change the department name of the course with value &110 to 'Math' such as the following:

$$U_2 = \mathsf{update}(\langle\langle \mathsf{root, course, dept} \rangle\rangle, \{\&110, \text{'Math'}\})$$

There would be potentially several data source nodes that would be returned when the rules contain $\langle\langle \mathsf{root,course,dept} \rangle\rangle$ are found.

6 Summary and Conclusions

In this paper we have considered the issue of normalisation of SSD as part of a data integration process. We defined a SSD first normal form that allows us to identify data values in sources by the use of a natural key, and hence perform data (as opposed to just schema) integration. It is proposed that the global schema be in second normal form to allow updates to data sources to be made in a consistent manner, and to minimise the size of the mapping tables between source and global schema object identifiers. The normal forms considered in this paper are weaker (and simpler) than those defined in the literature, but if required, our approach could easily be extended to use higher normal forms, such as those defined in [19]. However, 2NF is sufficient for our approach to work, and provided a data source is not denormalised by the data integration rules (*i.e.* brought down from a higher normal form to 2NF), then no update anomalies will be introduced by the process of data integration.

We introduced the functions generateGID and generateSID, which use a natural key to relate object identifiers in the 1NF (or higher) source schemas to identifiers in the global schema, and showed how these functions are used in the bi-directional transformation pathways of the BAV approach to data integration. The generateGID function allows data from different sources that require merging/joining to be correctly combined. It solves the identifier conflicts among local data sources, mapping them to a single global schema identifier, by relating them via a natural key. This reduces redundancy in the integrated schema, and allows updates to be performed without introducing anomalies. The generateSID function allows the original data sources to be restored. It ensures the pathways are reversible, and therefore allows updating from the global schema to the sources, and allows data and queries to be migrated in both directions — a functionality particularly important in peer-to-peer contexts.

Our approach has been implemented in the AutoMed data integration system (details of which may be found at (http://www.doc.ic.ac.uk/automed). The generateGID and generateSID functions have been integrated into the query processor to allow the system to correctly combine data from different sources, to support updating of data sources and to enable the original data sources to be restored. For the future work, we will be building larger case studies focusing on the integration of biological data sources, which are often held in the form of flat files, HTML or XML.

References

1. M. Arenas and L. Libkin. A normal form for xml documents. *ACM Transactions on Database Systems*, 29(1):195–232, 2004.
2. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE2004*, volume 3084 of *LNCS*, pages 82–97. Springer-Verlag, 2004.
3. V. Christophides, S. Cluet, and J. Siméon. On wrapping query languages and efficient xml integration. *SIGMOD Rec.*, 29(2):141–152, 2000.
4. S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proc. SIGMOD'98*, pages 177–188. ACM Press, 1998.
5. S. Cluet and J. Siméon. Data integration based on data conversion and restructuring. Technical report, Verso database group- INRIA, France, 1997.
6. C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 8th edition edition, 2004.
7. C.J. Date, H. Darwen, and D. McGoveran. *Relational Database: Selected Writings 1994–1997*. Addison-Wesley, 1998.
8. D.W. Embley and W.Y. Mok. Developing xml documents with guaranteed "good" properties. In *Proc. 20th ER*, pages 426–441, 2001.
9. E. Jasper, N. Tong, P.J. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. In *Proc. Baltic DB&IS04*, volume 672 of *Scientific Papers*, pages 13–30. Univ. Latvia, 2004.
10. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.
11. P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer, 1999.
12. P.J. McBrien and A. Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Proc. CAiSE'01*, volume 2068 of *LNCS*, pages 330–345. Springer, 2001.
13. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, pages 227–238. IEEE, 2003.
14. P.J. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P, at VLDB'03*, Berlin, Germany, 2003.
15. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 413–424. Morgan Kaufmann Publishers Inc., 1996.
16. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin. Translating web data. In *Proc 28th VLDB*, pages 598–609, 2002.
17. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin. Translating web data. Technical report, Department of Computer Science, University of Toronto, 2002.
18. A. Poulovassilis. The automed intermediate query language. Technical report, Department of Computer Science, Birkbeck College, 2001.
19. X. Wu, T.W. Ling, S.Y. Lee, M. Lee, and G. Dobbie. Nf-ss: A normal form for semistructured schema. In *ER 2001 Workshops*, pages 292–305, 2001.
20. L. Zamboulis and A. Poulovassilis. Using automed for xml data transformation and integration. In Z. Bellahsene and P.J. McBrien, editors, *Proc. DIWeb04, CAiSE Workshop Proceedings Volume 3*, pages 58–69, 2004.