Using AutoMed for XML Data Transformation and Integration

Lucas Zamboulis and Alexandra Poulovassilis

School of Computer Science and Information Systems,
Birkbeck College, University of London, London WC1E 7HX
{lucas,ap}@dcs.bbk.ac.uk

Abstract. This paper describes how the AutoMed data integration system is being extended to support the integration of heterogeneous XML documents. So far, the contributions of this research have been the development of two algorithms. One restructures the schema describing an XML document into another schema, and the other materialises an integrated schema resulting from the transformation of several source XML schemas, using the source XML data and the transformation pathways between the source and integrated schemas.

1 Introduction

Today's web-based applications and web services publish their data using XML, as this helps interoperability with other applications and services. However, since equivalent information can be published in many different ways, XML data exchange over the Web is not yet fully automatic. This heterogeneity of XML data has led to recent research in schema matching, schema transformation, and schema integration for XML. The development of algorithms that automate these tasks is essential to many domains: examples range from generic frameworks, such as for XML messaging and component-based development, to applications and services in e-business, e-science, entertainment, leisure and e-learning.

In this paper we describe how the AutoMed data integration system is being extended to support XML schema transformation and integration. Section 2 begins with an overview of the AutoMed system and its general approach to heterogeneous data integration. We also present the schema definition language we use for XML data. Section 3 presents and discusses our algorithms for restructuring and materialising XML schemas. Section 4 compares our approach with related work, and Section 5 gives our concluding remarks.

2 Our XML Data Integration Framework

The AutoMed system (see `http://www.doc.ic.ac.uk/automed/`) provides a low-level **hypergraph-based data model (HDM)**, in terms of which higher-level modelling languages can be defined. An HDM schema consists of a set of nodes, edges and constraints, and each modelling construct of a higher-level modelling language is specified as some combination of HDM nodes, edges and constraints. For any modelling language $\mathcal{M}$ specified in this way (via the API of AutoMed's Model Definitions Repository [1]) AutoMed provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$. In particular, for every construct of $\mathcal{M}$ there is an `add` and a `delete` transformation which add to/delete from a schema an instance of that construct. For those constructs of $\mathcal{M}$ which have textual names, there is also a `rename` transformation.

Schemas can be transformed by applying to them a sequence of primitive transformations each of which adds, deletes or renames just one schema construct. All source, intermediate, and integrated schemas, and the pathways between them, are stored in AutoMed's Schemas & Transformations Repository [1].

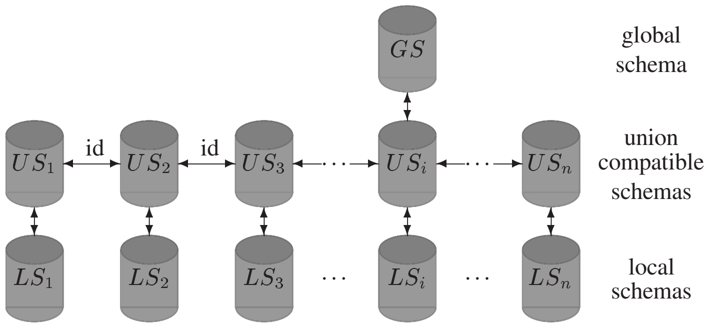

Fig. 1. Data integration in AutoMed

Figure 1 shows the general integration scenario. Each data source is described by a *data source schema*, LS_i. Each LS_i is transformed into a *union-compatible schema*, US_i, by means of a series of primitive schema transformations. All the union schemas[1] are syntactically identical and this is asserted by a series of `id` transformations between each pair US_i and US_{i+1} of union schemas. `id` is a primitive transformation that asserts the semantic equivalence of two syntactically identical constructs in two different schemas. The transformation pathway containing these `id` transformations is automatically generated. An arbitrary one of the union schemas can then be designated as the *global schema* GS, or can be selected for further transformation into a new schema that will become the global schema. We note that in general the LS_i, US_i and GS may be defined in different modelling languages, so that the intermediate schemas along the transformation pathways linking them may contain constructs of more than one modelling language. However, in our specific context here, all schemas will be XML ones.

Each `add` and `delete` transformation is accompanied by a query specifying the extent of the newly added or deleted construct in terms of the other schema constructs. This query is expressed in a functional query language, IQL [3]. In IQL, schema constructs are identified by means of their *scheme* enclosed within double chevrons $\langle\langle \ldots \rangle\rangle$

[1] We use the term 'union schema' synonymously with 'union-compatible schema'.

(see Section 2.1 below for some examples). Also available are `extend` and `contract` transformations which behave in the same way as `add` and `delete` except that they state that the extent of the new/removed construct cannot be derived from the other constructs present in the schema. If there is no information available about this extent, then this is indicated by using the constant `Void`.

The queries supplied with `add`, `extend`, `delete` and `contract` transformations provide the necessary information for these transformations to be automatically reversible. As discussed in [5], this means that AutoMed is a **both-as-view (BAV)** data integration system. It subsumes the LAV and GAV approaches [4], as it is possible to extract a definition of the global schema as a view over the data source schemas, and it is also possible to extract definitions of the data source schemas as views over the global schema.

2.1 A Schema for XML Data Sources

The standard schema definition languages for XML are DTD [11] and XML Schema [12]. However, for our purposes it is the structure of a given XML document that is crucial for schema/data integration. Also, it is possible that an XML document has no referenced DTD or XML Schema. We have therefore defined a simple modelling language called *XML DataSource Schema* (XMLDSS) which abstracts only the structure of an XML document. XMLDSS schemas consist of four kinds of constructs (here, we describe these somewhat informally and refer the reader to [14] for their formal specification in terms of the HDM):

Element: Elements, e, are identified by a scheme $\langle\langle e\rangle\rangle$. An XMLDSS element is represented by a node in the HDM.

Attribute: Attributes, a, belonging to elements, e, are identified by a scheme $\langle\langle e, a\rangle\rangle$. They are represented by a node in the HDM, representing the attribute, an edge between this node and the node representing e, and a cardinality constraint stating that an instance of e can have at most one instance of a associated with it, and an instance of a can be associated with one or more instances of e.

NestList: These are parent-child relationships between two elements e_p and e_c and are identified by a scheme $\langle\langle e_p, e_c, i\rangle\rangle$, where i is the position of e_c within the list of children of e_p. In the HDM, they are represented by an edge between the nodes representing e_p and e_c, and a cardinality constraint that states that each instance of e_p is associated with zero or more instances of e_c, and each instance of e_c is associated with precisely one instance of e_p[2].

PCDATA: In any XMLDSS schema there is one construct with scheme $\langle\langle$PCDATA$\rangle\rangle$, representing all the instances of PCDATA within an XML document. To link it with an element, we treat this as an element and use the NestList construct.

In an XML document there may be elements with the same name occurring at different positions in the tree. To avoid ambiguity, in XMLDSS schemas we use an

[2] Here, the fact that IQL is inherently list-based means that the ordering of children instances of e_c under parent instances of e_p is preserved within the extent of the NestList $\langle\langle e_p, e_c, i\rangle\rangle$.

identifier of the form $\langle elementName \rangle$:$\langle count \rangle$ for each element, where $\langle count \rangle$ is a counter incremented every time the same $\langle elementName \rangle$ is encountered in a depth-first traversal of the schema. For XML documents, we use an identifier of the form $\langle elementName \rangle$:$\langle count \rangle$:$\langle instance \rangle$ for each element, where $\langle instance \rangle$ is a counter identifying each instance in the document of the corresponding schema element.

To illustrate XMLDSS schemas, the XMLDSS schema of the left-hand document in Table 1 is S_1 in Figure 2, while the XMLDSS schema of the right-hand XML document in Table 1 is S_2 in Figure 2.

```
<root>
 <topic type="Fiction">
  <author firstn="U."  lastn="Eco"
          dob="1932 01 05">
   <book>
    <ISBN>0099466031</ISBN>
    <title>The Name of the Rose</title>
    <publisher>Vintage</publisher>
   </book>
   <book>
    <ISBN>0099287153</ISBN>
    <title>Foucault's Pendulum</title>
    <publisher>Vintage</publisher>
   </book>
   ...
  </author>
  ...
 </topic>
 <topic type="Mystery">
  <author firstn="A."  lastn="Reverte"
          dob="1951 11 24">
   <book>
    <ISBN>0099448599</ISBN>
    <title>The Dumas Club</title>
    <publisher>Vintage</publisher>
   </book>
   ...
  </author>
  ...
 </topic>
 ...
</root>
```

```
<root>
 <author birthday="05" birthmonth="10"
         birthyear="1955">
  <name> N.A. Buddy </name>
  <book ISBN="0A7B21C6D2">
   <title>Principles of Computing </title>
   <year> 1995 </year>
   <genre type="Computer Science" />
  </book>
  ...
 </author>
 <author birthday="07" birthmonth="05"
         birthyear="1945">
  <name>B.J. Whitehead</name>
  <book ISBN="A0B1C1D6E2">
   <title>Linear Algebra</title>
   <year> 2000 </year>
   <genre type="Mathematics" />
  </book>
  ...
 </author>
 ...
</root>
```

Table 1. Two source XML documents

The XMLDSS schema, Sch, of an XML document, Doc, can be extracted by means of a depth-first traversal of Doc, as follows:

1. Create the root element R_{Sch} of Sch by copying the root element of Doc, R_{Doc}, together with any attributes it may have (but not their values).
   Initially, let element E_{Doc} be R_{doc} and element E_{Sch} be R_{Sch}.
2. For every child element, C_{Doc}, of E_{Doc} do:
   (a) If an element with the same name as C_{Doc} does not appear in the list of children of E_{Sch}:
      i. Copy C_{Doc} and its attributes (without their values), and append this new element to the current list of children of E_{Sch}. If C_{Doc} has PCDATA, insert a NestList construct linking this new element to PCDATA in Sch.

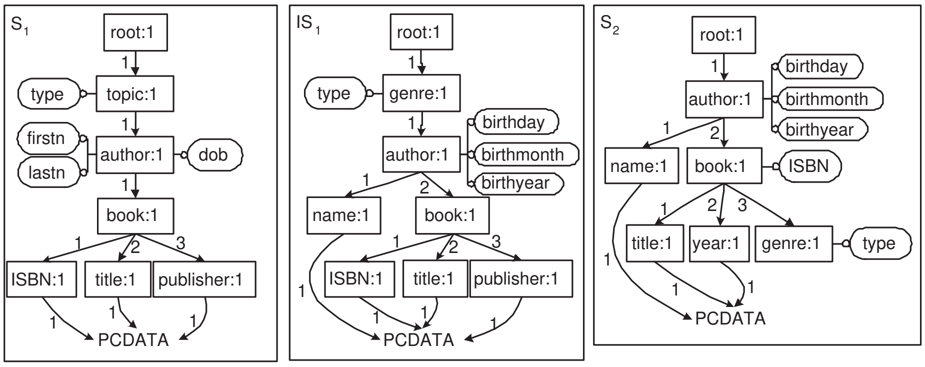

Fig. 2. Example XMLDSS schemas

  ii. Let E_{Doc} be C_{Doc} and recursively perform step 2.

(b) Otherwise, if there is an element C_{Sch} within the list of children of E_{Sch} with the same name as C_{Doc}:

  i. Copy any attributes of C_{Doc} that do not appear as attributes of C_{Sch} to C_{Sch}. If C_{Doc} has PCDATA, insert a NestList construct linking this new element to PCDATA in Sch.

  ii. Let E_{Doc} be C_{Doc} and recursively perform step 2.

The complexity of this algorithm is $O(N \times F)$ where N is the number of elements in the source XML document and F is the average fan-out of elements in the extracted schema. The XMLDSS schema is equivalent to the tree resulting as an intermediate step in the creation of a minimal DataGuide [2]. However, unlike DataGuides, we do not merge common sub-trees and the resulting schema remains a tree rather than a DAG.

3 Our Schema and Data Transformation Algorithms

The main aim of our research is to develop semi-automatic methods for generating the schema transformation pathways shown in Figure 1 in the context of XML data sources. The first step is a *schema matching phase* to discover the semantic equivalences between schema constructs in different source documents, and to use these equivalences in order to reconcile the source documents. This step results in a set of intermediate schemas $IS_1, \ldots, IS_n$ derived from $LS_1, \ldots, LS_n$, and we discuss it Section 3.1. The second step is to restructure $IS_1, \ldots, IS_n$ into a new set of conformed schemas $CS_1, \ldots, CS_n$, each of which has an identical structure. This is accomplished automatically using the schema restructuring algorithm described in Section 3.2. Each CS_i is then automatically extended with the schema constructs of the other conformed schemas. The resulting union schemas US_i are then connected in a pairwise fashion with `id` transformation pathways, also automatically. Finally, the global schema GS is derived and materialised, as we discuss in Section 3.3.

3.1 Schema matching

A recent survey of schema matching techniques is given in [7] several of which are applicable to XML data. The matches output by a schema matching process may be of four types, 1–1, n–1, 1–n and n–m, all of which can be used to automatically generate fragments of AutoMed transformation pathways. A 1–n match would generate n `add` transformations, each supplied with a query specifying the extent of the new construct in terms of that of the original construct, followed by a `delete` transformation to remove the original construct, which is now redundant. For example, schema matching may discover that attribute `dob` in S_1 is equivalent to a concatenation of attributes `birthyear`, `birthmonth` and `birthday` in S_2 (see Figure 2). This information can be used to generate the following sequence of transformations on S_1, assuming the availability of IQL functions `concat` and `word` (where `word i s` extracts the i-th word in string s):

```
addAttribute(<<author:1,birthday>>,  [{a,word 3 d} |
                                      {a,d}<-<<author:1,dob>>]);
addAttribute(<<author:1,birthmonth>>,[{a,word 2 d} |
                                      {a,d}<-<<author:1,dob>>]);
addAttribute(<<author:1,birthyear>>, [{a,word 1 d} |
                                      {a,d}<-<<author:1,dob>>]);
deleteAttribute(<<author:1,dob>>,[{a,concat [y,' ',m,' ',d]} |
                                 {a,d}<-<<author:1,birthday>>;
                                 {a,m}<-<<author:1,birthmonth>>;
                                 {a,y}<-<<author:1,birthyear}])
```

A 1–1 match is a special case of a 1–n match and would generate one `add` and one `delete` transformation. In some cases these may simplify to just one `rename` transformation. For example, the schema matching process may discover that element `topic` in S_1 matches element `genre` in S_2, and generate the transformation

```
renameElement (<<topic:1>>,<<genre:1>>)
```

A n–1 match would generate an `add` transformation, supplied with a query specifying how the extents of the n constructs need to be combined to create the extent of the new construct, followed by n `delete` transformations to remove the n original, and now redundant, constructs. Finally, n–m matches are a generalisation of n–1 and 1–n matches in that they generate m `add` transformations to insert the necessary new constructs, followed by n `delete` transformations to remove the original constructs. For example, schema matching may discover that the concatenation of attributes `firstn` and `lastn` in S_1 is equivalent to the element `name` and its link to `PCDATA` in S_2. This information can be used to generate the following sequence of transformations:

```
addElement(<<name:1>>,
           [{o} | {a,f}<-<<author:1,firstn>>;
            {a,l}<-<<author:1,lastn>>;
            {o}<-generateElemUID {f,l} '<<name:1>>']);
addNestList(<<author:1,name:1>>,
           [{a,o} | {a,f}<-<<author:1,firstn>>;
            {a,l}<-<<author:1,lastn>>;
            {o}<-generateElemUID {f,l} '<<name:1>>']);
addNestList(<<name:1,PCDATA>>,
```

```
            [{o,concat [f,' ',l]} | {a,f}<-<<author:1,firstn>>;
             {a,l}<-<<author:1,lastn>>;
             {o}<-generateElemUID {f,l} '<<name:1>>']);
deleteAttribute(<<author:1,firstn>>,
            [{a,word 1 n} | {a,o}<-<<author:1,name:1>>;
             {o,n}<-<<name:1,PCDATA>>]);
deleteAttribute(<<author:1,lastn>>,
            [{a,word 2 n} | {a,o}<-<<author:1,name:1>>;
             {o,n}<-<<name:1,PCDATA>>])
```

Here, we use a function `generateElemUID` which generates unique identifiers for instances of a new schema element, using as its input an instance of an attribute, or attributes, plus the scheme of the schema element. This function is useful for capturing the semantic equivalence of element with one, or more, attributes.

We finally note that applying the above sequence of transformations to schema S_1 in Figure 2 transforms it to IS_1.

3.2 Restructuring XMLDSS schemas

The restructuring of the set of intermediate schemas $IS_1, \ldots, IS_n$ output by the schema matching phase into a set of conformed schemas $CS_1, \ldots, CS_n$ is carried out one schema at a time. We begin by setting CS_1 to be IS_1. Suppose that $IS_1, \ldots, IS_{i-1}$ have already been restructured into $CS_1, \ldots, CS_{i-1}$ and that IS_i now needs to be restructured. We first extend $CS_1, \ldots, CS_{i-1}$ with any constructs of schema IS_i that they do not contain. We then take an arbitrary one of these conformed schemas, CS, and restructure IS_i to match CS. The restructuring algorithm consists of a *growing phase* where IS_i is augmented with constructs from CS that it is missing, followed by a *shrinking phase* where it is reduced to remove any constructs that are now redundant, followed by a *renaming phase* to rename labels as necessary to create a contiguous ordering of identifiers in parent-child relationships.

The general pseudocode for restructuring an XMLDSS schema S_1 to an XMLDSS schema S_2 is as follows (note that, following the schema matching phase, the algorithm considers an element in S_1 to be equivalent to an element in S_2 if they have the same name):

Growing phase: Consider every element E in S_2 in a depth-first order:

1. If an element E does not exist in S_1:
   (a) If there is an attribute a in S_1 with the same name as E in S_2, insert an element E into S_1 with an `add` transformation, using a query that generates unique IDs for the extent of E from the extent of a. Then, find the equivalent element of $parent(E, S_2)$ in S_1 and insert a NestList from it to E with an `add` transformation.
   (b) Otherwise, insert element E into S_1 using an `extend` transformation with the query `Void`. Then, find the equivalent element of $parent(E, S_2)$ in S_1 and insert a NestList from it to E using an `extend` transformation with the query `Void`.

2. Else if an element E exists in S_1 and $parent(E, S_1) \neq parent(E, S_2)$, find the equivalent element of $parent(E, S_2)$ in S_1, E_P, and insert a NestList from E_P to E.
3. Insert to E in S_1 any attributes from E in S_2 not present in E in S_1. To do this, for each attribute a from S_2, search S_1 for an element or an attribute with the same name as a. The insert transformation is an `add` or an `extend` depending on whether such an element or attribute is found.
4. If E is linked to the PCDATA construct in S_2, insert a NestList from E in S_1 to the PCDATA construct in S_1, using an `extend` if E was inserted into S_1 with an `extend`, otherwise using an `add`.

Shrinking phase: Traverse S_1 in a depth-first fashion and for every construct C in S_1, locate the equivalent construct in S_2. If this fails, remove C. This may be with a `delete` or a `contract` transformation, depending on whether it is possible to express the extent of C using the rest of the constructs of S_1. Note that before removing an element, its attributes and links need to be removed first, in order to preserve the consistency of the schema.

Renaming phase: Traverse S_1 in a depth-first fashion and rename labels as necessary in order to create a contiguous ordering of identifiers of parent-child relationships.

In step 2, the algorithm issues an `add` or an `extend` transformation depending on the edges in the path from E_P to E in S_1. In particular, if this path contains at some point an edge from a child element B to a parent element A then an `extend` transformation will be used to insert the NestList from E_P to E in S_1, otherwise an `add` transformation will be used. The reason an `extend` transformation is used in the first case is that in the data source of S_1 there may be instances of A that do not have instances of B as children. In such cases, the algorithm can `extend` the existing schemes $\langle\langle$B$\rangle\rangle$ and $\langle\langle$A,B$\rangle\rangle$ so as to generate for each such instance of A a new instance of B as a child, and then issue an `extend` transformation for the NestList. Alternatively, if such behaviour is not desired, the user has the option of instructing the algorithm to just issue an `extend` transformation for the NestList.

We note that our `add`/`extend`, `delete`/`contract` and `rename` transformations can be composed to cater for composite transformations such as 'move', transforming elements into attributes, etc. The separation of the growing phase from the shrinking phase ensures the completeness of the restructuring algorithm: since data transformation occurs within the queries supplied with the transformations, inserting all new schema constructs before removing any redundant constructs ensures that the constructs needed to define the extent of a new construct are present in the schema. Finally, we note that our algorithm has a worst-case complexity of $O(N_1 \times N_2)$, where N_1 and N_2 are the number of nodes of S_1 and S_2, respectively.

Table 2 lists the transformations issued by the schema restructuring algorithm in order to transform IS_1 in Figure 2 to S_2. In the 6^{th} step, `complete` is a composite transformation consisting of a sequence of `extend` steps (not shown here) which generate the necessary instances of `author:1` and `book:1` so as not to lose any `genre:1` instances. If this behaviour is not desired, this step would be ommitted.

Table 2. Transformation pathway $IS_1 \rightarrow S_2$

addNestList($\langle\langle$root : 1$\rangle\rangle$,$\langle\langle$author : 1$\rangle\rangle$,2,[{r,a}|{r,g}←$\langle\langle$root : 1, genre : 1$\rangle\rangle$;
{g,a}←$\langle\langle$genre : 1, author : 1$\rangle\rangle$])
addAttribute($\langle\langle$book : 1$\rangle\rangle$,$\langle\langle$book : 1, ISBN$\rangle\rangle$,[{b,p}|{b,i}←$\langle\langle$book : 1, ISBN : 1$\rangle\rangle$;
{i,p}←$\langle\langle$ISBN : 1, PCDATA$\rangle\rangle$])
extendElement($\langle\langle$year : 1$\rangle\rangle$,Void)
extendNestList($\langle\langle$book : 1$\rangle\rangle$,$\langle\langle$year : 1$\rangle\rangle$,4,Void)
extendNestList($\langle\langle$year : 1$\rangle\rangle$,$\langle\langle$PCDATA$\rangle\rangle$,1,Void)
complete($\langle\langle$genre : 1, author : 1$\rangle\rangle$,$\langle\langle$author : 1, book : 1$\rangle\rangle$)
extendNestList($\langle\langle$book : 1$\rangle\rangle$,$\langle\langle$genre : 1$\rangle\rangle$,5,[{b,g}|{g,a}←$\langle\langle$genre : 1, author : 1$\rangle\rangle$;
{a,b}←$\langle\langle$author : 1, book : 1$\rangle\rangle$])
deleteNestList($\langle\langle$root : 1$\rangle\rangle$,$\langle\langle$genre : 1$\rangle\rangle$,[{r,g}|{r,a}←$\langle\langle$root : 1, author : 1$\rangle\rangle$;
{a,b}←$\langle\langle$author : 1, book : 1$\rangle\rangle$;{b,g}←$\langle\langle$book : 1, genre : 1$\rangle\rangle$])
contractNestList($\langle\langle$genre : 1$\rangle\rangle$,$\langle\langle$author : 1$\rangle\rangle$,[{g,a}|{a,b}←$\langle\langle$author : 1, book : 1$\rangle\rangle$;
{b,g}←$\langle\langle$book : 1, genre : 1$\rangle\rangle$])
deleteNestList($\langle\langle$book : 1$\rangle\rangle$,$\langle\langle$ISBN : 1$\rangle\rangle$,[{b,o}|{b,i}←$\langle\langle$book : 1, ISBN$\rangle\rangle$;
{o}← generateElemUID {b,i} '$\langle\langle$ISBN : 1$\rangle\rangle$']
deleteNestList($\langle\langle$ISBN : 1$\rangle\rangle$,$\langle\langle$PCDATA$\rangle\rangle$,[{o,i}|{b,i}←$\langle\langle$book : 1, ISBN$\rangle\rangle$;
{o}← generateElemUID {b,i} '$\langle\langle$ISBN : 1$\rangle\rangle$'])
deleteElement($\langle\langle$ISBN : 1$\rangle\rangle$,[{o}|{b,i}←$\langle\langle$book : 1, ISBN$\rangle\rangle$;
{o}← generateElemUID {b,i} '$\langle\langle$ISBN : 1$\rangle\rangle$'])
contractNestList($\langle\langle$book : 1$\rangle\rangle$,$\langle\langle$publisher : 1$\rangle\rangle$,Void)
contractNestList($\langle\langle$publisher : 1$\rangle\rangle$,$\langle\langle$PCDATA$\rangle\rangle$,Void)
contractElement($\langle\langle$publisher : 1$\rangle\rangle$,Void)
renameNestList($\langle\langle$root : 1, author : 1, 2$\rangle\rangle$,$\langle\langle$root : 1, author : 1, 1$\rangle\rangle$)
renameNestList($\langle\langle$book : 1, title : 1, 2$\rangle\rangle$,$\langle\langle$book : 1, title : 1, 1$\rangle\rangle$)
renameNestList($\langle\langle$book : 1, year : 1, 4$\rangle\rangle$,$\langle\langle$book : 1, year : 1, 2$\rangle\rangle$)
renameNestList($\langle\langle$book : 1, genre : 1, 5$\rangle\rangle$,$\langle\langle$book : 1, genre : 1, 3$\rangle\rangle$)

So far we have assumed that element names in both S_1 and S_2 are unique. In general, this may not be the case and we may have (a) multiple occurrences of an element name in S_1 and a single occurrence in S_2, or (b) multiple occurrences of an element name in S_2 and a single occurrence in S_1, or (c) multiple occurrences of an element name in both S_1 and S_2. The restructuring algorithm described above can be extended as follows to handle these cases. For case (a), suppose that in S_1 there are n elements with the same name e and in S_2 a single element with name e. We can populate the single element in S_2 by combining the extents of the n elements in S_1. For case (b), suppose that in S_1 there is a single element with name e and in S_2 n elements with name e. We can make a choice of which of these elements to migrate the extent of S_1:e to, using a heuristic which favours paths with the fewest `extend` steps, and the shortest of such paths. For case (c), suppose that in S_1 there are n elements with the same name e and in S_2 m elements with name e. Then a combination of the solutions for (a) and (b) can be applied.

3.3 Generating and Materialising the Global Schema

So far we have created the transformation pathways from the data source schemas LS_i to the union schemas US_i. The next step is to create the global schema GS from an arbitrary one of the US_i. The transformation pathway from US_i to GS can be produced in one of two ways: either automatically, using 'append' semantics, or semi-automatically, using a different integration logic. In the first case, the queries populating the extents of the constructs of GS are generated by appending the corresponding constructs of US_1, ..., US_n in turn (thus, if the data sources were integrated in a different order, the extent of each construct of GS would contain the same instances, but with a different ordering). In the second case, the queries specifying the integration logic need to be supplied by the user.

The global schema can then be materialised. Our materialisation algorithm traverses GS in a depth-first fashion, and obtains the necessary data by evaluating the individual schema constructs of GS as global queries over the data sources. Global query processing uses the transformation pathways from GS to $LS_1, \ldots, LS_n$ (and in particular the queries within transformations) to reformulate the global query into a set of local queries on the data sources (for details please see [14]). An issue that arises during the materialisation is to ensure the correct parent-child relationships. For this purpose, we exploit the NestList instances and the instance-level unique IDs to correctly identify the parent of each element.

To conclude our running example, suppose that schemas S_1 and S_2 of Figure 2 have been integrated into a global schema GS which has the same structure as S_2. The materialised XML document produced by our materialisation algorithm, assuming that the data of S_1 precede the data of S_2, is illustrated in Figure 3. One final point is that two root notes from different source documents may contain conflicting data for some attribute. This case is handled by introducing a new generic root element into the final materialised document.

The pseudocode for the materialisation algorithm is as follows (for reasons of simplicity, we do not consider the case where a new generic root element is needed).

1. Create the root element of the materialised document Doc, setting its name to be name of the root element of GS, R_{GS}.
2. Create the attributes of R_{Doc}, setting their name to be the same as the attributes of R_{GS} (if any). Populate these attributes by retrieving their extents.
3. Initially, let N_{GS} be R_{GS} and N_{Doc} be a list consisting of the single element R_{Doc}.
4. For each outgoing edge of N_{GS}, E_{GS}, do:
   (a) Retrieve the extent of E_{GS}.
   (b) For every edge (p, c) in the extent of E_{GS}, do:
      – Find the element n in N_{Doc} that is equal to p and append c to the list of children of n.
      – Materialise the attributes of c in the same manner as the attributes of R_{Doc} in step 2
   (c) Let N'_{GS} be the child node specified by E_{GS} and N'_{Doc} be the list of children of N_{Doc} just created. Recursively perform step 4 with N_{GS} set to N'_{GS} and N_{Doc} set to N'_{Doc}.

```
<root>
 <author birthday="05" birthmonth="01"
        birthyear="1932">
  <name>U. Eco</name>
  <book ISBN="0099466031">
   <title>The Name of the Rose</title>
   <genre type="Fiction" />
  </book>
  <book ISBN="0099287153">
   <title>Foucault's Pendulum</title>
   <year />
   <genre type="Fiction" />
  </book>
  ...
 </author>
 <author birthday="24" birthmonth="11"
        birthyear="1951">
  <name>A. Reverte</name>
  <book ISBN="0099448599">
   <title>The Dumas Club</title>
   <genre type="Mystery" />
  </book>
  ...
 </author>
 ...
 <author birthday="05" birthmonth="10"
        birthyear="1955">
  <name>N.A. Buddy</name>
  <book ISBN="0A7B21C6D2">
   <title>Principles of Computing</title>
   <year>1995</year>
   <genre type="Computer Science" />
  </book>
  ...
 </author>
 <author birthday="07" birthmonth="05"
        birthyear="1945">
  <name>B.J. Whitehead</name>
  <book ISBN="A0B1C1D6E2">
   <title>Linear Algebra</title>
   <year>2000</year>
   <genre type="Mathematics" />
  </book>
  ...
 </author>
 ...
</root>
```

Table 3. Materialised Global Schema

4 Related Work

Clio [6] translates the data source schemas, XML or relational, into an internal representation. After the mappings between the source and the target schemas have been semi-automatically derived, the target schema is materialised from the data of the sources, using a set of rules and the mappings. Xyleme [8] also takes a mapping-based approach. It uses a tree as the global schema and source schemas are mapped to this tree through path mappings. DIXSE [9] transforms the DTD specifications of a set of source XML documents into an internal conceptual representation, using some heuristics to capture semantics and further input from domain experts. The approach in [10] transforms XML documents using a set of transformations on documents' DTDs, encoded in a transformation script. The target document is produced using this script and XSLT. A difference with out approach is that we use a small set of graph-based primitive schema transformations whereas the transformations of [10], and their transformation/materialisation algorithms, are DTD-dependent. The approach in [13] also uses XML-specific transformations for XML schema integration. However, the semantics of their transformations are not captured within queries, as ours are, and they do not generate elements to preserve data that would have otherwise been lost during restructuring.

5 Conclusions

This paper has described how the AutoMed data integration system is being extended to support the integration of heterogeneous XML documents. Assuming a semi-automatic schema matching process has first occurred, the subsequent schema restructuring, integration, and materialisation phases are automatic. We have described two algorithms, the first for restructuring the schema of an XML document into a target schema, and the second for materialising the global schema resulting from the transformation of several source XML documents. Their novelty lies in the use of XML-specific graph restructuring techniques applied to XML schemas.

We note that our algorithms can be applied in a *peer-to-peer setting*. Suppose there is a peer P_2 which needs XML data stored at a peer P_1. We consider P_1 as the peer whose XMLDSS schema S_1 needs to be transformed into the XMLDSS schema S_2 of P_2. After a schema matching phase, our schema restructing algorithm can be applied to S_1 to generate a transformation pathway from it to S_2. This transformation pathway can then be used to populate S_2 using data from S_1.

Although this paper has focussed on materialised integration of XML data, our approach can also be used to support virtual integration and global querying of XML data within the AutoMed system, and this is discussed in [14]. For future work we plan to measure empirically the effectiveness and scalability of our algorithms for different sizes and numbers of input XML documents and their XMLDSS schemas. We will also investigate automatic or semi-automatic techniques for generating more sophisticated integration logic than 'append' semantics within the transformation pathway from a union schema to the global schema. Another issue is supporting incremental materialisation of the global schema if the data or schema of a data source changes.

References

1. M. Boyd, P. McBrien, and N. Tong. The AutoMed Schema Integration Repository. In *Proc. BNCOD'02, LNCS 2405*, pages 42–45, 2002.
2. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB'97*, pages 436–445, 1997.
3. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the automed toolkit. AutoMed Technical Report 20, June 2003.
4. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246, 2002.
5. P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*. ICDE, March 2003.
6. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. VLDB'02*, pages 598–609, 2002.
7. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
8. C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogeneous Data Sources. In *Proc. IDEAS*, pages 199–208, 2001.
9. P. Rodriguez-Gianolli and J. Mylopoulos. A Semantic Approach to XML-based Data Integration. In *Proc. ER'01*, pages 117–132, 2001.
10. H. Su, H. Kuno, and E. A. Rudensteiner. Automating the Transformation of XML Documents. In *Proc. WIDM'01*, pages 68–75, 2001.
11. W3C. Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm#AEN56, June 1998.
12. W3C. XML Schema Specification. http://www.w3.org/XML/Schema, May 2001.
13. X. Yang, M.L. Lee, and T.W.Ling. Resolving structural conflicts in the integration of xml schemas: A semantic approach. In *Proc. ER'03*, pages 520–533, 2003.
14. L. Zamboulis. XML Data Integration by Graph Restructuring. In *To appear in Proc. BNCOD'04, LNCS*. Springer, 2004.