XML Data Integration By Graph Restructuring

Lucas Zamboulis

School of Computer Science and Information Systems,
Birkbeck College, University of London, London WC1E 7HX
E-mail: lucas@dcs.bbk.ac.uk

Abstract. This paper describes the integration of XML data sources within the AutoMed heterogeneous data integration system. The paper presents a description of the overall framework, as well as an overview of and comparison with related work and implemented solutions by other researchers. The main contribution of this research is an algorithm for the integration of XML data sources, based on graph restructuring of their schemas.

1 Introduction

The advent of XML as a new data format has given rise to new research issues. XML is the first step towards the realization of the Semantic Web vision. It is a markup language designed to structure and transmit data in an easy to manipulate form, however it is not the total solution — it does not *do* anything by itself. The second step consists of logic inference tools and tools that automate tasks which have been manual up to now. For this, well-studied research issues concerning mostly relational data need to be explored in the context of XML data. Such issues include schema matching and data integration, both virtual and materialized. In this paper we focus on the virtual integration of heterogeneous XML data sources by transforming the schemas of XML documents using graph restructuring techniques.

Section 2.1 provides an overview of the AutoMed system, and the AutoMed approach to data integration. Section 2.2 presents the schema definition language used for XML data, specifies its representation in terms of AutoMed's Common Data Model and presents the unique IDs used in our framework. Section 3 presents the schema transformation algorithm and describes the query engine and wrapper architecture. Section 4 reviews related work, while Section 5 gives our concluding remarks.

2 Our XML Data Integration Framework

2.1 Overview of AutoMed

AutoMed is a heterogeneous data integration system that supports a schema-transformation approach to data integration (see `http://www.doc.ic.ac.uk/a-`

`utomed`). Figure 1 shows the AutoMed integration approach in an XML setting. Each data source is described by a data source schema, denoted by S_i, which is transformed into a union-compatible schema US_i by a series of reversible primitive transformations, thereby creating a transformation pathway between a data source schema and its respective union-compatible schema. 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 special type of primitive transformation that 'matches' two syntactically identical constructs in two different union schemas, signifying their semantic equivalence. The transformation pathway containing these `id` transformations can be automatically generated. An arbitrary one of the union schemas can then be designated as the global schema GS, or selected for further transformation into a new schema that will become the global schema.

The transformation of a data source schema into a union schema is accomplished by applying a series of primitive transformations, each adding, deleting or renaming one schema construct. 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 AutoMed's Intermediate Query Language, IQL [16, 8]. The query supplied with a primitive transformation provides the necessary information to make primitive transformations automatically reversible. This means that AutoMed is a **both-as-view (BAV)** data integration system [14]. It subsumes the local-as-view (LAV) global-as-view (GAV) and GLAV approaches, 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 [14, 9].

In Figure 1, each US_i may contain information that cannot be derived from the corresponding S_i. These constructs are not inserted in the US_i through an `add` transformation, but rather through an `extend` transformation. This takes a pair of queries that specify a lower and an upper bound on the extent of the new construct. The lower bound may be `Void` and the upper bound may be `Any`, which respectively indicate no known information about the lower or upper bound of the extent of the new construct. There may also be information present in a data source schema S_i that should not be present within the corresponding US_i, and this is removed with a `contract` transformation, rather than with a `delete` transformation. Like `extend`, `contract` takes a pair of queries specifying a lower and upper bound on the extent of the deleted construct.

In our XML data integration setting, each XML data source is described by an XML DataSource Schema (a simple schema definition language presented in Section 2.2), S_i, and is transformed into an intermediate schema, I_i, by means of a series of primitive transformations that insert, remove, or rename schema constructs. The union schemas US_i are then automatically produced, and they extend each I_i with the constructs of the rest of the intermediate schemas. After that, the `id` transformation pathways between each pair US_i and US_{i+1} of union

[1] Henceforth we use the term 'union schema' to mean 'union-compatible schema'.

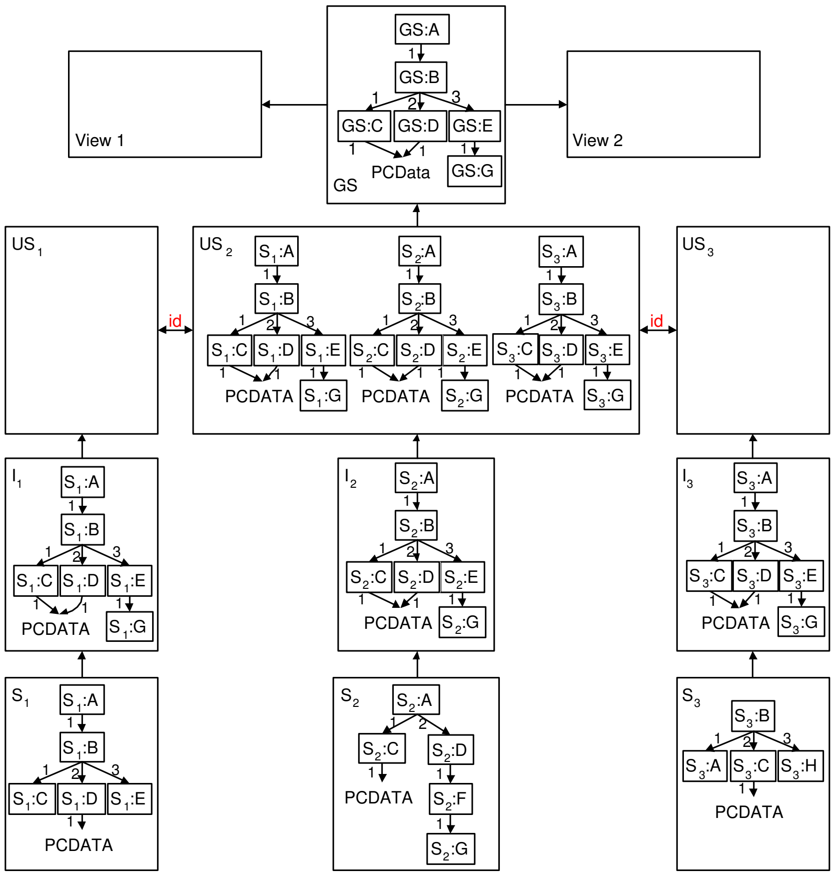

Fig. 1. XML integration in AutoMed

schemas are also automatically produced. Our XML integration framework supports both top-down and bottom-up schema integration. With the top-down approach, the global schema is predefined, and the data source schemas are restructured to match its structure. With the bottom-up approach, the global schema is not predefined and is automatically generated. Both approaches are described in Section 3.1.

2.2 A Schema for Representing XML Data Sources

When encoding data in XML, there two ways to enforce the intended structure and types over the XML files: DTD and XML Schema. However, as these technologies provide a complex grammar for the file, they describe the *possible* structure of a file, not the *actual* one. The structure of an XML file is very important in data integration, both for schema integration and in optimizing query processing. It is also possible that the XML file may have no referenced DTD or XML Schema. For these reasons, we introduce the `XML DataSource Schema`, which abstracts only the structure of an XML file, omitting information such as data types. The concept of XML DataSource Schema is similar to DataGuides [6]. However, XML DataSource Schemas are XML trees whereas DataGuides are OEM graphs.

To obtain an XML file's XML DataSource Schema, we first copy the XML file into memory, in its DOM representation. This copy will then be modified and become the XML DataSource Schema, according to the following pseudocode:

```
1. Get the root R. If it has child nodes, get its list of children, L.
   (a) Get the first node in L, N. For every other node N' in L that has the same tag
       as N do:
         – copy any of the attributes of N' not present in N to N
         – copy and append the list of children of N' to the list of children of N
         – delete N' and its subtree
   (b) Get the next child from the list of children and process it in the same way as
       the first child, L, in step (a) above.
2. Treat each one of the nodes in the new list of children of R, as R in step 1.
```

AutoMed has as its common data model a Hypergraph Data Model (HDM). This is a low-level data model that can represent higher-level modelling languages such as ER, relational, object-oriented and XML [12]. HDM schemas consist of nodes, edges and constraints. The selection of a low-level common data model for AutoMed was intentional, so as to be able to better represent high-level modelling languages without semantic mismatches or ambiguities.

Table 1 shows the representation of XML DataSource Schema constructs in terms of the HDM. XML DataSource Schemas consist of four constructs:

1. An element e can exist by itself and is a nodal construct. It is represented by the scheme $\langle\langle e\rangle\rangle$.
2. An attribute a belonging to an element e is a nodal-linking construct and is represented by the scheme $\langle\langle e, a\rangle\rangle$. In terms of the HDM this means that an attribute actually consists of a node representing the attribute, an edge linking the attribute node to its owner element, and a cardinality constraint.
3. The parent-child relationship between two elements e_p and e_c is a linking construct with scheme $\langle\langle e_p, e_c, i\rangle\rangle$, where i is the order of e_c in the list of children of e_p. In terms of the HDM, this is represented by an edge between e_p and e_c and a cardinality constraint.
4. Text in XML is represented by the PCDATA construct. This is a nodal construct with scheme $\langle\langle \text{PCDATA}\rangle\rangle$. In any schema, there is only one PCDATA construct. To link the PCDATA construct with an element we treat it as an element and use the nest-list construct.

Note that this is a simpler XML schema language than that given in [12]. In our model here, we make specific the ordering of children elements under a common parent in XML DataSource Schemas (the identifiers i in **NestList**

Table 1. XML DataSource Schema representation in terms of HDM

Higher Level Construct	**Equivalent HDM Representation**
Construct: **Element** Class nodal Scheme $\langle\langle e\rangle\rangle$	Node $\langle\langle xml{:}e\rangle\rangle$
Construct: **Attribute** Class: nodal-linking, constraint Scheme: $\langle\langle e, a\rangle\rangle$	Node $\langle\langle xml{:}e{:}a\rangle\rangle$ Edge $\langle\langle \_, xml{:}e, xml{:}e{:}a\rangle\rangle$ Links $\langle\langle xml{:}e\rangle\rangle$ Cons $makeCard(\langle\langle \_, xml{:}e, xml{:}e{:}a\rangle\rangle, 0{:}1, 1{:}N)$
Construct **NestList** Class linking, constraint Scheme $\langle\langle e_p, e_c, i\rangle\rangle$	Edge $\langle\langle i, xml{:}e_p, xml{:}e_c\rangle\rangle$ Links $\langle\langle xml{:}e_p\rangle\rangle$, $\langle\langle xml{:}e_c\rangle\rangle$ Cons $makeCard(\langle\langle i, xml{:}e_p, xml{:}e_c\rangle\rangle, 0{:}N, 1{:}1)$
Construct: **PCDATA** Class nodal Scheme $\langle\langle$PCDATA$\rangle\rangle$	Node $\langle\langle xml{:}$PCDATA$\rangle\rangle$

schemes) whereas this was not captured by the model in [12]. Also, in that paper it was assumed that the extents of schema constructs are sets and therefore extra constructs 'order' and 'nest-set' were required, to respectively represent the ordering of children nodes under parent nodes, and parent-child relationships where ordering is not significant. Here, we make use of the fact that IQL is inherently list-based, and thus use only one **NestList** construct. The n^{th} child of a parent node can be specified by means of a query specifying the corresponding nest-list, and the requested node will be the n^{th} item in the IQL result list.

A problem when dealing with XML DataSource Schema is that multiple XML elements can have the same name. The problem is amplified when dealing with multiple files, as in our case. To resolve such ambiguities, a unique IDs assignment technique had to be devised. For XML DataSource Schemas, the assignment technique is $\langle schemaName\rangle{:}\langle elementName\rangle{:}\langle count\rangle$, where $\langle schemaName\rangle$ is the schema name as defined in the AutoMed repository and $\langle count\rangle$ is a counter incremented every time the same $\langle elementName\rangle$ is encountered, in a depth-first traversal of the schema. As for XML documents themselves, the same technique is used, except that the unique identifiers for elements are of the form $\langle schemaName\rangle{:}\langle elementName\rangle{:}\langle count\rangle{:}\langle instance\rangle$ where $\langle instance\rangle$ is a counter incremented every time a new instance of the corresponding schema element is encountered in the document.

After a modelling language has been defined in terms of HDM via the API of AutoMed's Model Definition Repository [1], a set of primitive transformations is automatically available for the transformation of the schemas defined in the language, as discussed in Section 2.1.

3 Framework Components

Our research focuses on the development of semi-automatic methods for generating the schema transformation pathways shown in Figure 1. The first step is a *schema matching* [17] process, using for example data mining, or semantic mappings to ontologies. Both approaches can be used to automatically generate fragments of AutoMed transformation pathways — see for example [19]. Once this process reveals the semantic equivalences between schema constructs, the algorithm we describe in Section 3.1 integrates the XML DataSource Schemas by transforming each one into its respective union schema, using graph restructuring techniques. When several sources have been integrated, the global schema can be used for querying the data sources, as described in Section 3.2.

3.1 Schema Transformation Algorithm

Our schema transformation algorithm can be applied in two ways: top-down, where the global schema is predefined and the data source schemas are transformed to match it, regardless of any loss of information; or bottom-up, where there is no predefined global schema and the information of all the data sources is preserved. Both approaches create the transformation pathways that produce intermediate schemas with identical structure. These schemas are then automatically transformed into the union schemas US_i of Figure 1, including the `id` transformation pathways between them. The transformation pathway from one of the US_i to GS can then be produced in one of two ways: either automatically, using 'append' semantics, or semi-automatically, in which case the queries supplied with the transformations that specify the integration need to be supplied by the user. By 'append' semantics we mean that the extents of the constructs of GS are created by appending the extents of the corresponding constructs of US_1, US_2, ..., US_n in turn. Thus, if the XML data sources were integrated in a different order, the extent of each construct of GS would contain the same instances, but their ordering would be different.

Top-down approach: Consider a setting where a global schema GS is given, and the data source schemas need to be conformed to it, without necessarily preserving their information capacity. Our algorithm works in two phases. In the *growing phase*, GS is traversed and every construct not present in a data source schema S_i is inserted. In the *shrinking phase*, each schema S_i is traversed and any construct not present in the global schema is removed.

The algorithm to transform an XML DataSource Schema S_1 to have the same structure as an XML DataSource Schema S_2 is specified below. This algorithm considers an element in S_1 to be equivalent to an element in S_2 if they have the same element name. As specified below, the algorithm assumes that element names in both S_1 and S_2 are unique. We discuss shortly the necessary extensions to cater for cases when this does not hold.

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

1. If E does not exist in S_1:
   (a) Search S_1 to find an attribute a with the same name as the name of E in S_2
      i. If such an attribute is found, `add` E to S_1 with the extent of a and `add` an edge from the element in S_1 equivalent to $owner(a, S_2)$ to E.

        ii. Otherwise, `extend` E. Then find the equivalent element of $parent(E, S_1)$ in S_2 and `add` an edge from it to E with an `extend` transformation.
        iii. In both cases, insert the attributes of E from schema S_2 as attributes to the newly inserted element E in S_1 with `add` or `extend` transformations, depending on if it is possible to describe the extent of an attribute using the rest of the constructs of S_1. Note that one or more of these insertions might result in the transformation of an element in S_1 into an attribute.
    (b) If E is linked to the PCDATA construct in S_2:
        i. If the PCDATA construct is not present in S_1, insert it with an `extend` transformation, then insert an edge from E to the PCDATA construct, also with an `extend` transformation.
        ii. Otherwise, `add` an edge from E to the PCDATA construct.
2. If E exists in S_1 and $parent(E, S_2) = parent(E, S_1)$:
    (a) Insert any attributes of E in S_2 that do not appear in E in S_1, similarly to 1(a)iii.
    (b) If E is linked to the PCDATA construct in S_2, do the same as in 1b.
3. If E exists in S_1 and $parent(E, S_2) \neq parent(E, S_1)$:
    (a) Insert an edge from E_P to E, where E_P is the equivalent element of $parent(E, S_2)$ in S_1. This insertion can either be an `add` or an `extend` transformation, depending on the path from E_P to E. The algorithm finds the shortest path from E_P to E, and, if it includes only parent-to-child edges, then the transformation is an `add`, otherwise it is an `extend`.
    (b) Insert any attributes of E in S_2 that do not appear in E in S_1, similarly to 1(a)iii.
    (c) If E is linked to the PCDATA construct in S_2, do the same as in 1b.

Shrinking phase: traverse S_1 and remove any constructs not present in S_2.
Renaming phase: traverse S_1 and rename edge labels as necessary to create a contiguous ordering of identifiers.

In step 3a, the algorithm decides whether to issue an `add` or an `extend` transformation, depending on the type of edges the path from E_P to E contains. To explain this, suppose that the path contains at some point an edge (B, A), where actually, in S_1, element `A` is the parent of element `B`. It may be the case that in the data source of S_1, there are some instances of `A` that do not have instances of `B` as children. As a result, when migrating data from the data source of S_1 to schema S_2, some data will be lost, specifically those `A` instances without any `B` children. To remedy this, the `extend` transformation is issued with both a lower and an upper bound query. The first query retrieves the actual data from the data source of S_1, but perhaps losing some data because of the problem just described. The second query retrieves all the data that the first query retrieves, but also generates new instances of `B` (with unique IDs) in order to preserve the instances of `A` that the lower bound query was not able to. Such a behaviour may not always be desired, so the user has the option of telling the algorithm to just use `Any` as the upper bound query in such cases.

An example application of the algorithm is illustrated in Figure 2. The corresponding transformation pathway is shown in Table 2 and is divided into three sections. The first one illustrates the growing phase, where schema S_1 is augmented with the constructs from schema S_2. The second illustrates the shrinking

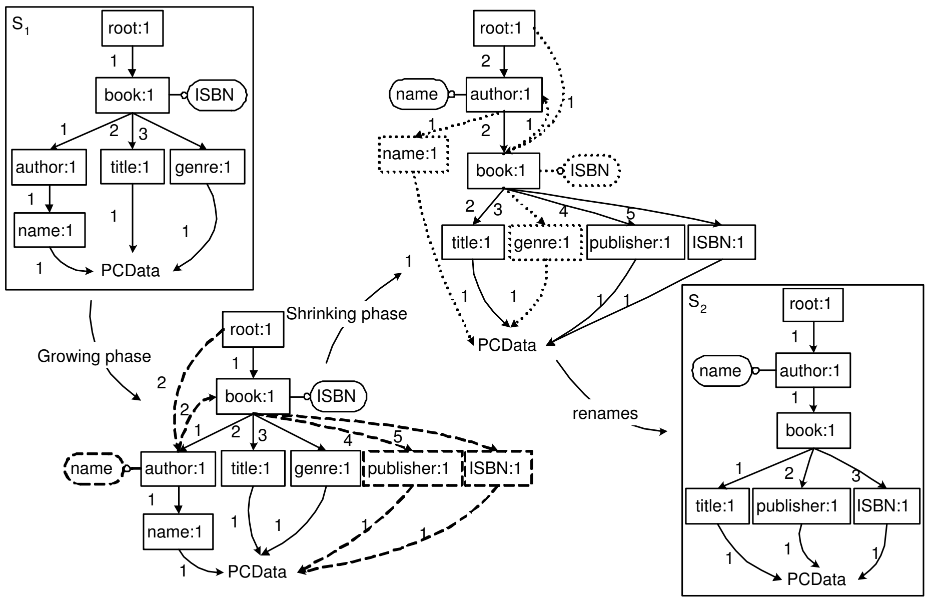

Fig. 2. Example Schema Transformation.

phase, where the constructs of S_1 that do not appear in S_2 are removed. The first one illustrates the growing phase, the second the shrinking phase, and the third the renaming phase. The `generateElemUID` function generates element instances from attribute instances and is useful whenever an attribute is transformed into an element, and vice versa — see g_7–g_9 and s_2–s_4 in Table 2.

Combining AutoMed's insert and remove operations allows more complex transformations to be achieved. For instance, in step 1(a)i of the algorithm, an attribute is transformed into an element, e.g. see g_7 in Table 2; in step 1(a)iii, elements may be transformed into attributes, e.g. see g_2. Finally, step 3a of the algorithm simulates a *move* operation [3], e.g. see g_3.

The algorithm presented above assumes 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.

For case (a), the algorithm needs to generate a query that constructs the extent of the single element in S_2 by combining the extents of all three elements from S_1. For case (b), the algorithm needs to make a choice of which of the elements from S_1 to migrate the extent of the single element in S_2 to. For this, a heuristic can be applied which favours (i) paths with the fewest `extend` steps, and (ii) the shortest of such paths. For case (c), a combination of the solutions for (a) and (b) needs to be applied.

Table 2. Transformations for Figure 2.

g_1: $addNestList(\langle\langle root{:}1\rangle\rangle, \langle\langle author{:}1\rangle\rangle, 2, [\{r,a\} \vert \{r,b\} \leftarrow \langle\langle root{:}1, book{:}1, 1\rangle\rangle; \{b,a\} \leftarrow \langle\langle book{:}1, author{:}1, 1\rangle\rangle])$
g_2: $addAttribute(\langle\langle author{:}1\rangle\rangle, \langle\langle author{:}1, name\rangle\rangle, [\{a,p\} \vert \{a,n\} \leftarrow \langle\langle author{:}1, name{:}1\rangle\rangle; \{n,p\} \leftarrow \langle\langle name{:}1,\text{PCDATA}\rangle\rangle])$
g_3: $extendNestList(\langle\langle author{:}1\rangle\rangle, \langle\langle book{:}1\rangle\rangle, 2, [\{a,b\} \vert \{b,a\} \leftarrow \langle\langle book{:}1, author{:}1, 1\rangle\rangle], Any)$
g_4: $extendElement(\langle\langle publisher{:}1\rangle\rangle, Void, Any)$
g_5: $extendNestList(\langle\langle book{:}1\rangle\rangle, \langle\langle publisher{:}1\rangle\rangle, 4, Void, Any)$
g_6: $extendNestList(\langle\langle publisher{:}1\rangle\rangle, \langle\langle \text{PCDATA}\rangle\rangle, 1, Void, Any)$
g_7: $addElement(\langle\langle ISBN{:}1\rangle\rangle, [\{o\} \vert \{b,i\} \leftarrow \langle\langle book{:}1, ISBN\rangle\rangle;\ \{o\} \leftarrow generateElemUID\ \{b,i\}\ \langle\langle ISBN{:}1\rangle\rangle])$
g_8: $addNestList(\langle\langle book{:}1\rangle\rangle, \langle\langle ISBN{:}1\rangle\rangle, 5, [\{b,o\} \vert \{b,i\} \leftarrow \langle\langle book{:}1, ISBN\rangle\rangle;\ \{o\} \leftarrow generateElemUID\ \{b,i\}\ \langle\langle ISBN{:}1\rangle\rangle])$
g_9: $addNestList(\langle\langle ISBN{:}1\rangle\rangle, \langle\langle \text{PCDATA}\rangle\rangle, 1, [\{o,i\} \vert \{b,i\} \leftarrow \langle\langle book{:}1, ISBN\rangle\rangle; \{o\} \leftarrow generateElemUID\ \{b,i\}\ \langle\langle ISBN{:}1\rangle\rangle])$
s_1: $deleteNestList(\langle\langle root{:}1\rangle\rangle, \langle\langle book{:}1\rangle\rangle, [\{r,b\} \vert \{r,a\} \leftarrow \langle\langle root{:}1, author{:}1, 2\rangle\rangle; \{a,b\} \leftarrow \langle\langle author{:}1, book{:}1, 2\rangle\rangle])$
s_2: $deleteNestList(\langle\langle author{:}1\rangle\rangle, \langle\langle name{:}1\rangle\rangle, [\{a,o\} \vert \{a,n\} \leftarrow \langle\langle author{:}1, name\rangle\rangle; \{o\} \leftarrow generateElemUID\ \{a,n\}\ \langle\langle name{:}1\rangle\rangle])$
s_3: $deleteNestList(\langle\langle name{:}1\rangle\rangle, \langle\langle \text{PCDATA}\rangle\rangle, [\{o,n\} \vert \{a,n\} \leftarrow \langle\langle author{:}1, name\rangle\rangle; \{o\} \leftarrow generateElemUID\ \{a,n\}\ \langle\langle name{:}1\rangle\rangle])$
s_4: $deleteElement(\langle\langle name{:}1\rangle\rangle, [\{o\} \vert \{a,n\} \leftarrow \langle\langle author{:}1, name\rangle\rangle; \{o\} \leftarrow generateElemUID\ \{a,n\}\ \langle\langle name{:}1\rangle\rangle])$
s_5: $deleteAttribute(\langle\langle book{:}1{:}ISBN\rangle\rangle, [\{o\} \vert \{b,i\} \leftarrow \langle\langle book{:}1, ISBN{:}1\rangle\rangle; \{i,p\} \leftarrow \langle\langle ISBN{:}1,\text{PCDATA}\rangle\rangle])$
s_6: $contractNestList(\langle\langle book{:}1\rangle\rangle, \langle\langle author{:}1\rangle\rangle, [\{b,a\} \vert \{a,b\} \leftarrow \langle\langle author{:}1, book{:}1, 2\rangle\rangle], Any)$
s_7: $contractNestList(\langle\langle book{:}1, genre{:}1, 3\rangle\rangle, Void, Any)$
s_8: $contractNestList(\langle\langle genre{:}1,\text{PCDATA}, 1\rangle\rangle, Void, Any)$
s_9: $contractElement(\langle\langle genre{:}1\rangle\rangle, Void, Any)$
r_1: $renameNestList(\langle\langle root{:}1, author{:}1, 2\rangle\rangle, \langle\langle root{:}1, author{:}1, 1\rangle\rangle)$
r_2: $renameNestList(\langle\langle author{:}1, book{:}1, 2\rangle\rangle, \langle\langle author{:}1, book{:}1, 1\rangle\rangle)$
r_3: $renameNestList(\langle\langle book{:}1, title{:}1, 2\rangle\rangle, \langle\langle book{:}1, title{:}1, 1\rangle\rangle)$
r_4: $renameNestList(\langle\langle book{:}1, publisher{:}1, 4\rangle\rangle, \langle\langle book{:}1, publisher{:}1, 2\rangle\rangle)$
r_5: $renameNestList(\langle\langle book{:}1, ISBN{:}1, 5\rangle\rangle, \langle\langle book{:}1, ISBN{:}1, 3\rangle\rangle)$

Bottom-up approach: In this approach, a global schema GS is not present and is produced automatically from the source schemas, without loss of information. A slightly different version of the above schema transformation algorithm is applied to the data source schemas in a pairwise fashion, in order to incrementally derive each one's union-compatible schema (Figure 3). The data source schemas S_i are first transformed into intermediate schemas, IS_i. Then, the union schemas, US_i, are produced along with the `id` transformations. To start with, the intermediate schema of the first data source schema is itself, $S_1 = IS_1^1$. Then, the schema transformation algorithm is employed on IS_1^1 and S_2 (see annotation 1 in Figure 3) The algorithm augments IS_1^1 with the constructs from S_2 it does not contain. It also restructures S_2 to match the structure of IS_1^1, also augmenting it with the constructs from IS_1^1 it does not contain. As a result, IS_1^1 is transformed to IS_1^2, while S_2 is transformed to IS_2^1. The same process is performed between IS_2^1 and S_3, resulting in the creation of IS_2^2 and IS_3^1 (annotation 2). The algorithm is then applied between IS_1^2 and IS_2^2, resulting only in the creation of IS_1^3, since this time IS_1^2 does not have any constructs IS_2^2 does not contain (annotation 3). The remaining intermediate schemas are generated in the same manner: to produce schema IS_i, the schema transformation algorithm is employed on IS_{i-1}^1 and S_i, resulting in the creation of IS_{i-1}^2 and IS_i^1; all other intermediate schemas except IS_{i-1}^2 and IS_i^1 are then extended with the constructs of S_i they do not contain. Finally, we automatically generate the union schemas, US_i, the `id` transformations between them, and the global schema (by using append semantics).

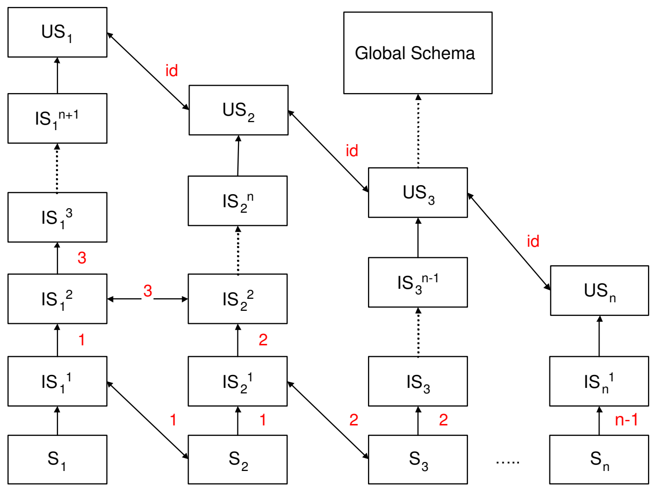

Fig. 3. XML DataSource Schema integration

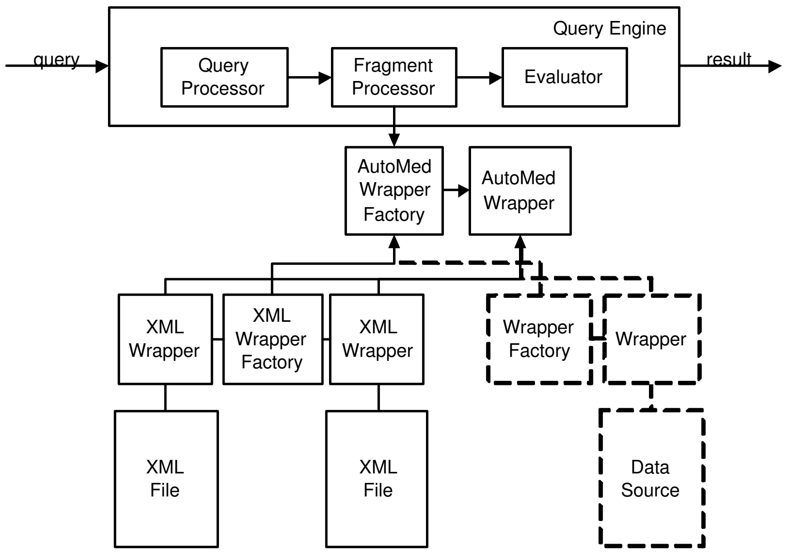

Fig. 4. AutoMed's query engine & wrapper architecture

3.2 Querying XML Files

AutoMed's query engine and wrapper architecture are shown in Figure 4. The `AutoMedWrapperFactory` and `AutoMedWrapper` classes are abstract classes that implement some of the abstract methods, while the `XMLWrapperFactory` and `XMLWrapper` classes implement the remaining abstract methods. Factories deal with model specific aspects, e.g. primary keys for relational databases. The XML-specific factory class contains a validating switch. When it is on, the parsing of the XML file the `XMLWrapper` object is attached to is performed by consulting the DTD/XML Schema the file references. A number of switches, e.g. a switch for collapsing whitespace, will be added in the future. As Figure 4 indicates, the architecture is extensible with wrappers for new data source models.

After the generation of transformation pathways by either top-down or bottom-up integration, queries submitted to the global schema can be evaluated. A query submitted to the query engine is first processed by the query processor, which is responsible for reformulating it into a query that can be evaluated over the data sources. This is accomplished by following the reverse transformation pathways from the global schema to the data source schemas. Each time a `delete`, `contract` or `rename` transformation is encountered, it replaces any occurrences of the removed or renamed scheme with the query supplied with the transformation. As a result, the original query is turned into a query that can be evaluated on the data sources — see [16, 9]. The fragment processor replaces IQL subqueries by XML wrapper objects. The evaluator then evaluates the query, making a call to the XML wrapper object where necessary. At present, querying of XML files is performed by translating IQL queries into XPath. Future plans include XQuery support.

As an example, suppose that schemas S_1 and S_2 of Figure 2 have been integrated into the global schema GS of Figure 5 which is the global schema in this integration scenario. In order to retrieve the title and genre of each book in the global schema, the following query is submitted to GS (for simplicity, we omit here the element counter from each element name):

$$[\{b,t\}|\{b,i\} \leftarrow \langle\langle book, title\rangle\rangle; \{i,t\} \leftarrow \langle\langle title, PCDATA\rangle\rangle] \ ++$$
$$[\{b,g\}|\{b,i\} \leftarrow \langle\langle book, genre\rangle\rangle; \{i,g\} \leftarrow \langle\langle genre, PCDATA\rangle\rangle]$$

The fragments of the transformation pathways from S_1 and S_2 to GS of relevance to this query are:

$S_1 \rightarrow GS$:
$extendElement(\langle\langle publisher{:}1\rangle\rangle, Void, Any)$
$extendNestList(\langle\langle book{:}1\rangle\rangle, \langle\langle publisher{:}1\rangle\rangle, Void, Any)$
$extendNestList(\langle\langle publisher{:}1\rangle\rangle, \langle\langle \text{PCDATA:1}\rangle\rangle, Void, Any)$

$S_2 \rightarrow GS$:
$extendElement(\langle\langle genre{:}1\rangle\rangle, Void, Any)$
$extendNestList(\langle\langle book{:}1\rangle\rangle, \langle\langle genre{:}1\rangle\rangle, Void, Any)$
$extendNestList(\langle\langle genre{:}1\rangle\rangle, \langle\langle \text{PCDATA:1}\rangle\rangle, Void, Any)$

Traversing the pathways $GS \rightarrow US_1$ and $GS \rightarrow US_2$, the above query is reformulated to:
$[\{b,t\}|\{b,i\} \leftarrow (US_1{:}\langle\langle book, title\rangle\rangle \ ++ \ US_2{:}\langle\langle book, title\rangle\rangle);$
$\quad \{i,t\} \leftarrow (US_1{:}\langle\langle title, PCDATA\rangle\rangle \ ++ \ US_2{:}\langle\langle title, PCDATA\rangle\rangle)] \ ++$
$[\{b,g\}|\{b,i\} \leftarrow (US_1{:}\langle\langle book, genre\rangle\rangle \ ++ \ US_2{:}\langle\langle book, genre\rangle\rangle);$
$\quad \{i,g\} \leftarrow (US_1{:}\langle\langle genre, PCDATA\rangle\rangle \ ++ \ US_2{:}\langle\langle genre, PCDATA\rangle\rangle)]$

Then, traversing the pathways $US_1 \rightarrow S_1$ and $US_2 \rightarrow S_2$, we obtain:
$[\{b,t\}|\{b,i\} \leftarrow (S_1{:}\langle\langle book, title\rangle\rangle \ ++ \ S_2{:}\langle\langle book, title\rangle\rangle);$
$\quad \{i,t\} \leftarrow (S_1{:}\langle\langle title, PCDATA\rangle\rangle \ ++ \ S_2{:}\langle\langle title, PCDATA\rangle\rangle)] \ ++$
$[\{b,g\}|\{b,i\} \leftarrow (S_1{:}\langle\langle book, genre\rangle\rangle \ ++ \ (Void, Any));$
$\quad \{i,g\} \leftarrow (S_1{:}\langle\langle genre, PCDATA\rangle\rangle \ ++ \ (Void, Any))]$

Instances of `(Void,Any)` can be eliminated, using the techniques described in [9], giving the following final query:
$[\{b,t\}|\{b,i\} \leftarrow (S_1{:}\langle\langle book, title\rangle\rangle \ ++ \ S_2{:}\langle\langle book, title\rangle\rangle);$
$\quad \{i,t\} \leftarrow (S_1{:}\langle\langle title, PCDATA\rangle\rangle \ ++ \ S_2{:}\langle\langle title, PCDATA\rangle\rangle)] \ ++$
$[\{b,g\}|\{b,i\} \leftarrow S_1{:}\langle\langle book, genre\rangle\rangle;$
$\quad \{i,g\} \leftarrow S_1{:}\langle\langle genre, PCDATA\rangle\rangle]$

4 Related Work

Schema matching is a problem well-studied in a relational database setting. A recent survey on schema matching is [17]. A machine-learning approach to schema matching is [4] and an approach using neural networks is [11]; however both

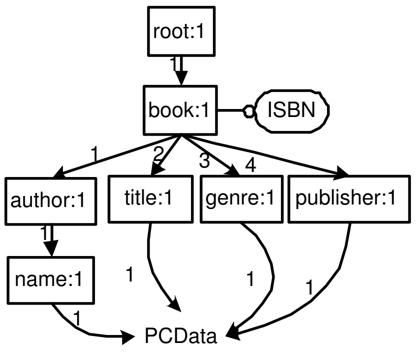

Fig. 5. Global schema *GS*.

approaches are semi-automatic. In [10], the schema authors provide themselves the semantics of the schema elements, by providing mappings between elements of their schemas to a global ontology.

Concerning schema integration, Clio [15] first transforms the data source schemas, XML or relational, into an internal representation. Then, after the mappings between the source and the target schemas have been semi-automatically derived, Clio materializes the target schema with the data of the source, using a set of internal rules, based on the mappings. DIXSE [20] follows a similar approach, as it transforms the DTD specifications of the source documents into an inner conceptual representation, with some heuristics to capture semantics. Most work, though, is done semi-automatically by the domain experts that augment the conceptual schema with semantics. The approach in [18] has an abstract global DTD, expressed as a tree, very similar to a global ontology. The connection between this DTD and the DTDs of the data sources is through path mappings: each path between two nodes in a source DTD is mapped to a path in the abstract DTD. Then, query rewriting is employed to query the sources.

In the context of virtual data integration, SilkRoute [5] and XPERANTO [2] are both middleware systems that use query rewriting to translate user queries submitted in XML query languages to the language of the underlying data sources. These systems are GAV. The approach in [7] on the other hand combines GAV, LAV and GLAV integration rules in a peer-to-peer setting.

The framework we have presented in this paper approaches the XML data integration problem using graph restructuring techniques. Our approach allows for the use of multiple types of schema matching methods (use of ontologies, semantics provided in RDF, data-mining), which can all serve as an input to the schema integration algorithm. The main contributions of the work presented here is the automatic integration of XML data using a purely XML solution.

5 Concluding Remarks

This paper has presented a framework for the virtual integration of XML data within the AutoMed heterogeneous data integration system. Assuming a semi-automatic schema matching process, the schema transformation algorithm succeeds in integrating XML data sources automatically. The algorithm makes use of a simple schema definition language for XML data sources and an assignment technique for unique identifiers, both developed specifically for the purposes of XML data integration. The novelty of the algorithm is the use of XML-specific graph restructuring techniques. Our framework also supports the materialization of the virtual global schema, as discussed in [21].

We note that our schema transformation algorithm can also be applied in a *peer-to-peer setting*. Suppose there is a peer P_T that needs to query XML data stored at a peer P_S. We can consider P_S as the peer whose XML DataSource Schema needs to be transformed to the XML DataSource Schema of peer P_T. After the application of our algorithm, P_T can then query P_S for the data it needs via its own schema, since AutoMed's query engine can treat the schema of P_T as the 'global' schema and the schema of P_S' as the 'local schema'.

Evolution of applications or changing performance requirements may cause the schema of a data source to change. In the AutoMed project, research has already focused on the schema evolution problem in the context of virtual data integration [13, 14]. For future work we will investigate the application of these general solutions specifically for XML. The main advantage of AutoMed's both-as-view approach in this context is that it is based on pathways of reversible schema transformations. This enables the development of algorithms that update the transformation pathways and the global schema, instead of regenerating them when data source schemas are modified. These algorithms can be fully automatic if the information content of a data source schema remains the same or contracts, though require domain knowledge or human intervention if it expands.

References

1. M. Boyd, P. McBrien, and N. Tong. The AutoMed Schema Integration Repository. In *Proceedings of the 19th British National Conference on Databases*, pages 42–45. Springer-Verlag, 2002.
2. M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.
3. G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *ICDE (Data Engineering, also in BDA 2001)*, 2002.
4. A. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *SIGMOD Conference*, 2001.
5. M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middleware Queries. In *SIGMOD Conference*, 2001.
6. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, 1997.

7. Alon Halevy, Zachary Ives, Peter Mork, and Igor Tatarinov. Piazza: Data management infrastructure for semantic web. In *Proceedings of the Twelfth International World Wide Web Conference*, pages 556–567. ACM, May 2003.
8. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL Queries and Migrating Data in the AutoMed toolkit. AutoMed Technical Report 20, June 2003.
9. E. Jasper, N. Tong, P. Brien, and A. Poulovassilis. View Generation and Optimisation in the AutoMed Data Integration Framework. AutoMed Technical Report 16, October 2003. To appear in *6th International Baltic Conference on Databases & Information Systems 2004*.
10. L. Lakshmanan and F. Sadri. XML Interoperability. In *ACM SIGMOD Workshop on Web and Databases (WebDB), San Diego, CA*, pages 19–24, June 2003.
11. W. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, September 1994.
12. P. McBrien and A. Poulovassilis. A Semantic Approach to Integrating XML and Structured Data Sources. In *Conference on Advanced Information Systems Engineering*, pages 330–345, 2001.
13. P. McBrien and A. Poulovassilis. Schema Evolution In Heterogeneous Database Architectures, A Schema Transformation Approach. In *CAiSE*, pages 484–499, 2002.
14. P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *19th International Conference on Data Engineering*. ICDE, March 2003.
15. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. VLDB'02*, pages 598–609, 2002.
16. A. Poulovassilis. The AutoMed Intermediate Query Langauge. AutoMed Technical Report 2, June 2001.
17. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
18. C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogeneous Data Sources. In *IDEAS*, pages 199–208. IEEE Computer Society, 2001.
19. N. Rizopoulos. BAV Transformations on Relational Schemas Based on Semantic Relationships between Attributes. AutoMed Technical Report 22, August 2003.
20. P. Rodriguez-Gianolli and J. Mylopoulos. A Semantic Approach to XML-based Data Integration. In *ER*, volume 2224 of *Lecture Notes in Computer Science*, pages 117–132. Springer, November 2001.
21. L. Zamboulis. XML Data Integration By Graph Restructuring. AutoMed Technical Report 27, February 2004.