XML Data Integration By Graph Restructuring

Lucas Zamboulis and Alexandra Poulovassilis
School of Computer Science and Information Systems
Birkbeck College, University of London,
{lucas,ap}@dcs.bbk.ac.uk

Abstract

This technical report describes the XML data integration framework being built within the AutoMed heterogeneous data integration system. It presents a description of the overall framework, as well as an overview of and comparison with related work and solutions by other researchers. The contributions of this research are the development of two algorithms for XML data integration, the first for schema integration and the second for view materialization, both based on graph restructuring.

1 Introduction

In the past ten years, the Internet and the World Wide Web have become an important part of everyday life. However, the Web still receives a limited amount of help from computers. Computers are effective in low-level operations, such as handling large amounts of data, search facilities, and transmitting and displaying data, but lack functionality in more sophisticated tasks, such as the ones envisaged in the Semantic Web vision [3]. To remedy this, data has to be put on the Web in a form that machines can understand, or convert it into that form, then provide the machines with the means to process it.

XML is the first step towards this end. It is a markup language designed to structure and transmit data in an easy to manipulate form, but 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 specific tasks which have been manual up to now. These tools involve the combination of XML with RDF and ontologies so as to enable computers to provide high-level services by communicating with other web services and applications. Other tools are concerned with automating tasks that are up to now expensive and time-consuming, as they require the development of new programs every time. Such tasks include importing/exporting data from/to XML files, automatic schema matching and migrating and integrating XML data.

Apart from the research issues concerning the Semantic Web, the advent of XML as a new data format has given rise to new research issues. Efficient storing of XML data has resulted in the evolution of commercial relational databases to support importing and exporting of XML data and also the development of native XML database products. The need for efficient querying of XML data has led to the development of various query languages for XML, subsumed now by the XPath [31] and XQuery [34] languages. Moreover, well-studied research issues concerning mostly the relational domain need to be redefined in XML, in order to find domain-specific solutions. At present, there is a lot of research effort on developing XML-specific solutions for the schema matching and data integration problems.

This report describes a framework for XML data integration within the AutoMed heterogeneous data integration system. It presents the basis for this framework, which is a new schema definition language for XML data, and a technique for assigning unique identifiers to XML elements. Two new algorithms have been developed which perform schema integration and view materialization, respectively. The report also discusses related work and compares the work presented here with the work of other researchers.

Report outline: Section 2.1 provides an overview of the AutoMed system, as well as the AutoMed approach to data integration. Section 2.2 presents the schema definition language used for XML data, specifies the representation of the language in terms of AutoMed's Common Data Model and presents the unique identifiers used in our framework. Section 3 presents the schema transformation and the view materialization algorithms and describes the query engine and wrapper architecture. Section 4 reviews related work and compares it with the work presented here. Finally, Section 5 gives some concluding remarks on the framework together with plans for future work.

2 The XML Data Integration Framework

2.1 Overview of AutoMed

The AutoMed heterogeneous data integration system supports a schema-transformation approach to data integration. Figure 1 shows the general integration scenario. Each data source is described by a data source schema, denoted by LS_i. Each LS_i 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.

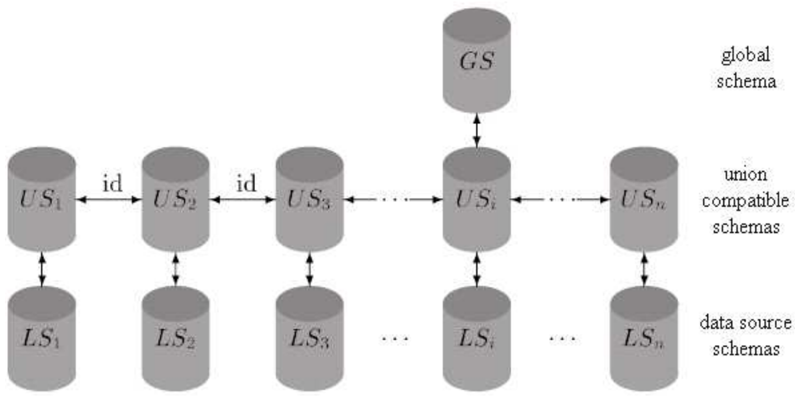

Figure 1: The AutoMed integration approach

The transformation of a data source schema into a union schema is accomplished by applying a series of primitive transformations, each making a 'delta' change to the current schema by either 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 [23, 12]. The result is a sequence of intermediate schemas, connecting each data source schema to its respective union schema. The query supplied with a primitive transformation defines the new or removed schema construct in terms of the other schema constructs, and thus provides the necessary information to make primitive transformations automatically reversible [17]. This means that AutoMed is a **both-as-view (BAV)** data integration system [21]. It subsumes the LAV and GAV 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.

In Figure 1, each US_i may contain information that cannot be derived from the corresponding LS_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 LS_i that should not be present within the corresponding US_i, and this is removed with a `contract` transformation, rather

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

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.

Figure 2 shows the AutoMed integration approach in an XML 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, $S_i^{'}$, 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 $S_i^{'}$ 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 schemas are also automatically produced. Our XML integration framework supports both top-down and bottom-up schema integration. With the top-down approach, described in Section 3.1.1, the global schema is predefined, and the data source schemas are restructured to match its structure. With the bottom-up approach, described in Section 3.1.2, the global schema is not predefined and is automatically generated.

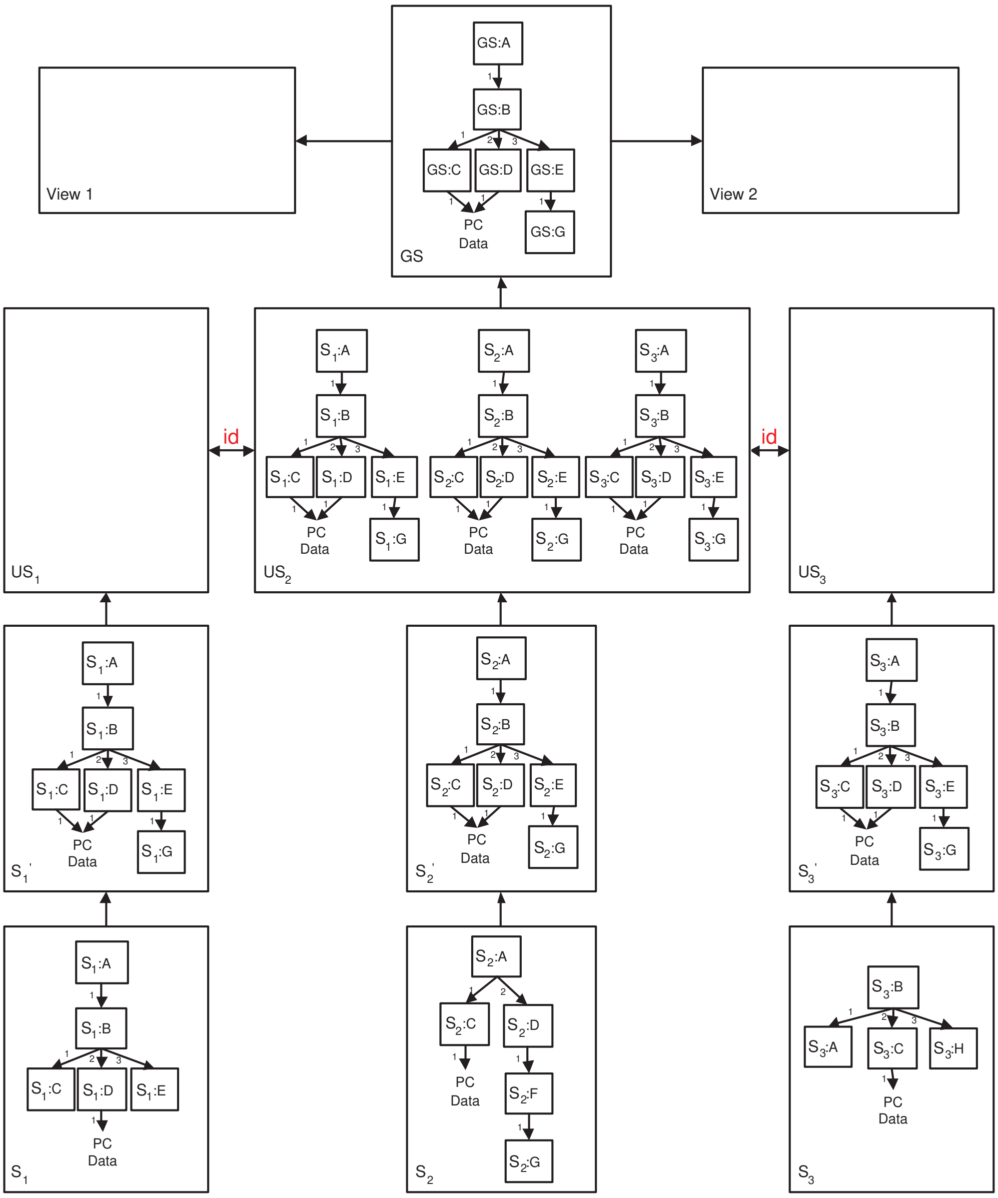

Figure 2: XML integration in AutoMed

2.2 A Schema for Representing XML Data Sources

XML files may or may not have a DTD [30] or XML Schema [32] associated with them. These are complex grammars to which the file must conform. In a data integration setting, a schema definition language this complex is not necessary. However, the file's structure is crucial for schema and data integration and in optimizing the query processing algorithms. Also, it is possible that the XML file has no referenced DTD or XML Schema. For these reasons, we introduce the *XML DataSource Schema*, which is automatically derivable from an XML file, and abstracts only its structure.

To obtain an XML file's DataSource Schema, we copy the file's DOM representation (see [33]) into memory, then we modify it to become the DataSource Schema, as specified by the following algorithm:

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.
      - Make a 'deep' copy of the list of children of N' and append them to the list of children of N. 'Deep' means that the copy contains the whole tree whose root is the copied node.
      - Delete N' and its subtree.

   (b) Get the next child from L and process it in the same way as the first child, N, in step (a).
2. R now has a new list of children L_{new}. Apply step (1) for every node N_{new} in L_{new}.

AutoMed has as its Common Data Model a Hypergraph Data Model (HDM) [24]. This is a low-level data model that can represent higher-level modelling languages such as ER, relational, object-oriented and XML [18, 19]. HDM schemas consist of nodes, edges and constraints. Constructs of higher-level schemas are identified by their *scheme* (see below). 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.

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 **nest-list** Class linking, constraint Scheme $\langle\langle e_p, e_c, i\rangle\rangle$	Edge $\langle\langle xml : i, e_p, xml : e_c\rangle\rangle$ Cons $makeCard(\langle\langle xml : i, e_p, xml : e_c\rangle\rangle, 0 : N, 1 : 1)$
Construct **pcdata** Class nodal Scheme $\langle\langle$pcdata$\rangle\rangle$	Node $\langle\langle xml : pcdata\rangle\rangle$

Table 1: XML DataSource Schema representation in terms of HDM

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 element e is a nodal-linking construct. In terms of the HDM this means that an attribute is represented by a node representing the attribute with scheme $\langle\langle xml : e : a\rangle\rangle$, an edge linking the attribute node to its owner element with scheme $\langle\langle \_, xml : e, xml : e : a\rangle\rangle$, and a cardinality constraint that states that an instance of e can have at most one instance of a associated with it, while an instance of a can be associated with one or more instances of e.
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 that states that each

instance of e_p is associated with 0 or more instances of e_c, while an instance of e_c is associated with precisely one instance of e_p.

4. Text in XML is represented by the PCData construct. This is a nodal construct with scheme $\langle\langle$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 somewhat different from the XML schema language given in [19]. In our model here, we make specific the ordering of children elements under a common parent in XML DataSource Schemas whereas this was not captured by the model in [19]. 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 **nest-list** 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.

After a modelling language has been defined in terms of HDM via the API of AutoMed's Model Definition Repository [4], a set of primitive transformations is automatically available for the transformation of the schemas defined in the language. The XML DataSource Schema definition language consists of four different constructs, namely `element`, `attribute`, `nest-list` and `pcdata`. The available transformations on XML DataSource Schemas are shown in Table 2. The `lowerBound` and `upperBound` parameters in the `extend` and `contract` transformations are queries that partially specify the extent of the construct being inserted/removed.

Insert primitive transformations	
addElem(schema, elemNode, query)	extendElem(schema, elemNode, lowerBound, upperBound)
addAttr(schema, elemNode, attNode, query)	extendAttr(schema, elemNode, attNode, lowerBound, upperBound)
addPCData(schema, query)	extendPCData(schema)
addList(schema, parent, child, position, query)	extendList(schema, parent, child, position, lowerBound, upperBound)
Remove primitive transformations	
deleteElem(schema, elemNode, query)	contractElem(schema, elemNode, lowerBound, upperBound)
deleteAttr(schema, elemNode, attNode, query)	contractAttr(schema, elemNode, attNode, lowerBound, upperBound)
deletePCData(schema, query)	contractPCData(schema)
deleteList(schema, parent, child, query)	contractList(schema, parent, child, position lowerBound, upperBound)
Rename primitive transformations	
renameElem(schema, elemNode, newName)	renameAttr(schema, elemNode, attrNode, newName)
renameList(schema, nestList, position)	

Table 2: XML primitive transformations

XML DataSource Schemas are very similar to DataGuides. According to [9], the benefit of having a DataGuide is threefold: define data structure, help users understand the structure of the database and form queries over it and help the query processor devise efficient query plans for computing query results. XML DataSource Schema also fulfills these aims. The main reason for creating a new schema definition language is simplicity: DataGuides are OEM graphs, whereas XML DataSource Schemas are trees. This means that they are very easy to parse, traverse and manipulate. A more detailed comparison between XML DataSource Schemas and DataGuides is given in Section 2.2.1.

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 new unique identifiers assignment technique had to be implemented. For XML DataSource Schemas, the assignment technique is $\langle schemaName\rangle$:$\langle elementName\rangle$:$\langle count\rangle$, where $\langle schemaName\rangle$ is the name of the DataSource Schema 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. Attributes are then identified as $\langle elementUID\rangle$:$\langle attributeName\rangle$.

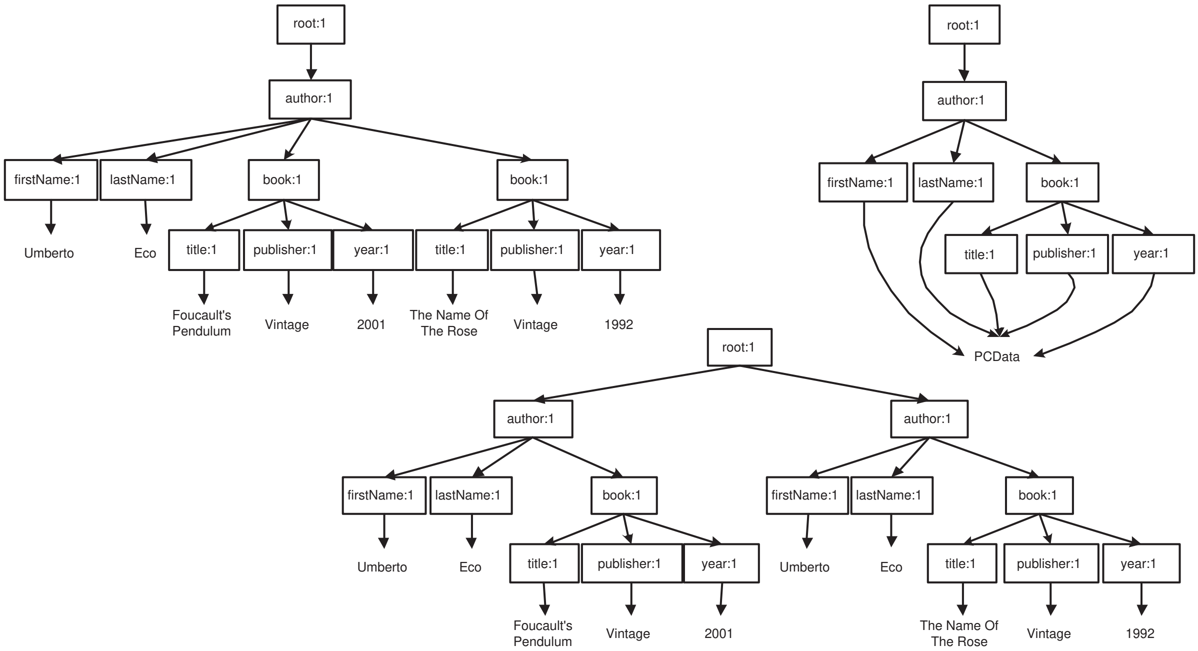

Figure 3: Two different XML files conforming to the same XML DataSource Schema.

In order to capture node identity in XML data, we use a similar technique: the unique identifiers for nodes 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.

2.2.1 XML DataSource Schema vs. DataGuides

The concept of XML DataSource Schema is very similar to DataGuides. According to [9], the benefit of having a DataGuide is threefold: define the structure of the data, enable users to understand the structure of the database and form meaningful queries over it and help the query processor devise efficient query plans for computing query results. Looking at these aims, we can see that XML DataSource Schema also fulfills them. The main reason for creating a new schema definition language is simplicity: DataGuides are OEM graphs, whereas XML DataSource Schemas are XML trees. This means that they are very easy to parse, traverse and manipulate.

A difference between the two types of schemas is that a source can have many DataGuides, but only one XML DataSource Schema. On the other hand, both types of schemas may correspond to more than one data sources. For example, the sources in Figure 3 have the same XML DataSource Schema, shown in the upper right corner, even though they are not the same.

Another difference between XML DataSource Schemas and DataGuides is the way they handle the ordering of child elements. In [10], the authors define the problem and suggest three different approaches. XML DataSource Schema does not use any of these techniques, as it does not try to solve the ordering problem. The reason for this is that XML DataSource Schema is used only for single files, contrary to DataGuides. Of course, even in single files there is the issue of an element having the same child elements with different ordering in different instances. Consider the following case:

```
<W>
  <X><A><B/><C/></A></X>
  <X><A><C/><B/></A></X>
</W>
```

Even in this case, there is no need to try and find the best ordering possible. The fact that an element does not present a specific policy on its children's ordering, means that there isn't one, so there is no reason to try and enforce one. This also agrees with the XML Schema specification, which either enforces a strict ordering policy, or none at all.

One thing that will probably be added to the algorithm producing the XML DataSource Schema is the ability to preserve a file's ordering policy, if there is one, even in the presence of optional elements. Consider the following example:

```
<W>
  <X><A/><C/></X>
  <X><A/><B/><C/></X>
</W>
```

This shows that a possible scenario is that the file has an ordering policy, namely first element `A`, then `B`, then `C`, but B is optional. The algorithm, if not changed, will create the following schema:

```
<W>
  <X><A/><C/><B/></X>
</W>
```

Of course, this could be very easily detected if the file references an XML Schema. This is a topic for future work, as discussed in section 5.

3 Framework Components

The main aim of our research is to develop semi-automatic methods for generating the schema transformation pathways shown in Figure 2. This includes two aspects: first, the matching of individual elements, also known as *schema matching* [15, 25], using for example data mining techniques, or semantic mappings to ontologies. Both approaches can be used to automatically generate fragments of AutoMed transformation pathways - see for example [27]. Next, graph restructuring is applied to restructure the heterogeneous XML DataSource Schemas into a uniform structure.

Once the semantic equivalences between schema constructs have been identified with schema matching, our framework integrates the data source schemas by transforming each one into its respective union schema (see Figure 2). This *schema transformation* process is accomplished by an algorithm that automatically creates the transformation pathway from a data source schema to its corresponding union schema. The algorithm, described in Section 3.1, is based on the restructuring of XML DataSource Schemas. Once several sources have been integrated under a virtual global schema, this can be used for querying the data sources via the `XMLWrapper`, as described in Section 3.2, or for materializing the data from one or more data sources. This *view materialization* algorithm is described in Section 3.3.

3.1 Schema Transformation Algorithm

The schema transformation algorithm can be applied in two ways: top-down, where there is a global schema and the schemas of the data sources are transformed into the global schema, regardless of any loss of information; or bottom-up, where there is no global schema and the data of all the data sources are 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 policy need to be supplied by the user. By 'append' semantics we mean that that the lists containing the extents of the constructs of GS are created by appending 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.

3.1.1 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*, the global schema 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. These two phases represent the fact that first the source schemas are augmented with constructs from the global schema, then they are reduced by removing the redundant constructs. However, some removals also occur in the first phase. This is in order to reduce the cost of the traversal of the schemas: if a necessary removal is detected in the first phase, it is cheaper to issue the transformation at that stage than detect it again in the second phase.

The algorithm to transform an XML Datasource Schema S_2 to have the same structure as an XML Datasource Schema S_1 is described as follows. 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_1 in a depth first order.
  1. If E does not exist in S_2:
     (a) Search the attributes in S_2 to find one whose name is the same as the name of E in S_1
        i. If such an attribute a is found, `add` E with the extent of a and `add` an edge from the element in S_1 equivalent to $owner(a, S_2)$ to E. Then, insert the attributes of E from schema S_1 as attributes to the newly inserted element E in S_2 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_2. Then, `delete` a. This situation is illustrated in case 1 in Figure 4.
        ii. Otherwise, insert E with an extend transformation. 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. Next, insert the attributes of E from 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_2 (case 2 in Figure 4).
     (b) If E is linked to the PCData construct in S_1:
        i. If S_2 does not contain the PCData construct, insert it with an `extend` transformation.
        ii. Insert an edge from E to the PCData construct. The transformation is an `add`, if E was inserted with an `add` transformation and there was already a PCData construct in S_2 before the application of the algorithm. In any other case, the transformation is an `extend`.
  2. If E exists in S_2 and $parent(E, S_1) = parent(E, S_2)$ (case 3 in Figure 4):
     (a) If E in S_1 has attributes that E in S_2 does not contain, insert them with `add` or `extend` transformations, depending on if it is possible to describe their extents using other constructs of S_2.
     (b) If E is linked to the PCData construct in S_1 and there was already a PCData construct in S_2 before the application of the algorithm, `add` an edge from E to the PCData construct, otherwise `extend` the edge.
  3. If E exists in S_2 and $parent(E, S_1) \neq parent(E, S_2)$:
     (a) Insert an edge from E_P to E, where E_P is the equivalent element of $parent(E, S_1)$ in S_2. 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`. To explain this, suppose that the path contains at some point an edge (B, A), where actually, in S_2, element `A` is the parent of element `B`. It may be the case that in the data source of S_2, there are some instances of `A` that do not have instances of `B` as children. This means that, when migrating data from the data source of S_2 to schema S_1, 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 bound and an upper bound query. The lower bound query retrieves the actual data from the data source of S_2, but perhaps losing some data because of the problem just described. The upper bound query retrieves all the data that the lower bound query retrieves, but also generates new instances of `B` (with unique IDs) that are needed in order to preserve the

instances of `A` that the lower bound query was not able to. Because such behavior may not always be desired, the user has the option of telling the algorithm to just use `Any` as the upper bound query in such cases. In Figure 4, case 4 illustrates a situation where the edge from E_P to E is inserted with an `add` transformation, whereas in cases 5 and 6 it is inserted with an `extend` transformation. Case 6 in particular represents an edge 'flip'.

- Shrinking phase: traverse S_2 and remove any constructs not present in S_1. The transformation is a `delete` or a `contract` one, depending on whether it is possible to describe the extent of the construct with the remaining constructs of the schema, or not, respectively.

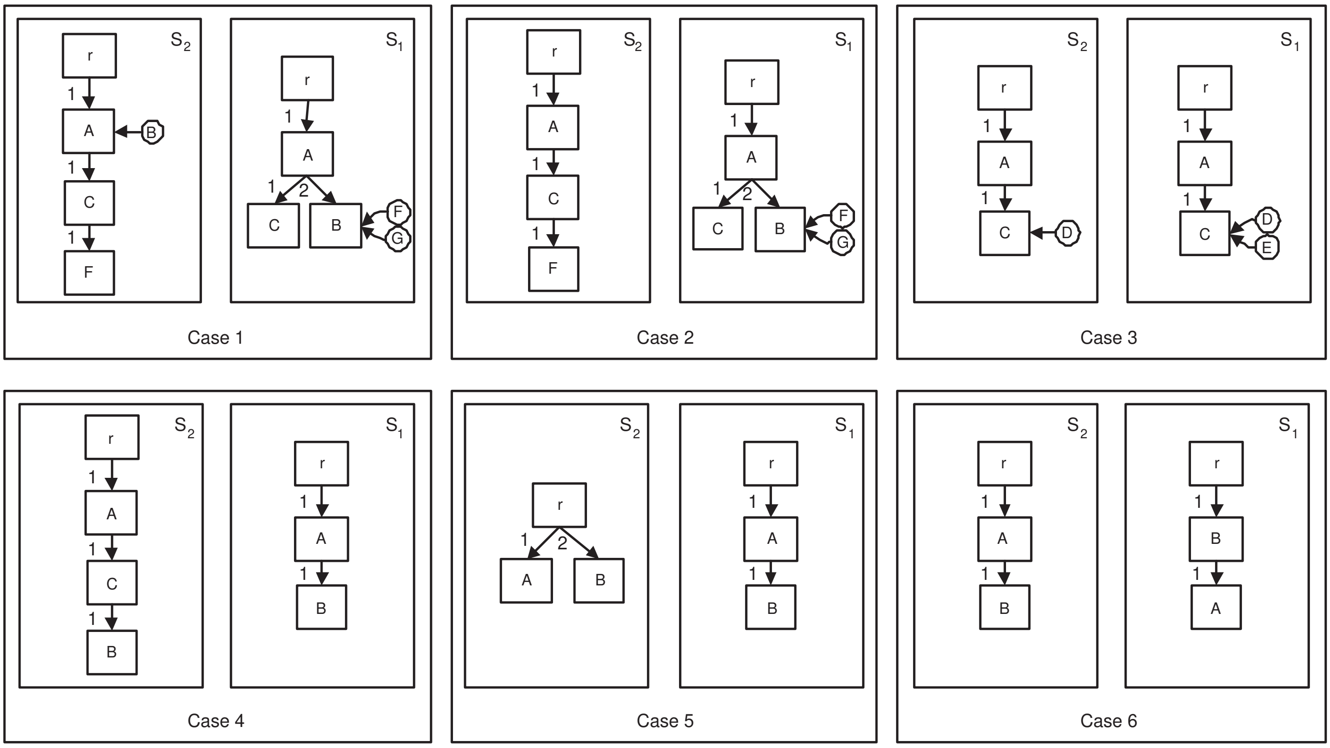

Figure 4: Example cases for the schema transformation algorithm.

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_2 and a single occurrence in S_1, or (b) multiple occurrences of an element name in S_1 and a single occurrence in S_2, or (c) multiple occurrences of an element name in both S_1 and S_2.

For case (a), suppose that in S_2 there are three elements S_2:*employees*:1, S_2:*employees*:2, S_2:*employees*:3, and in S_1 there is a single element S_1:*employees*:1. The algorithm then needs to generate a query that constructs the extent of the single element in S_1 by combining the extents of all three elements from S_2.

For case (b), suppose that in S_2 there is a single element S_2:*employees*:1 while in S_1 there are three elements S_1:*employees*:1, S_1:*employees*:2, S_2:*employees*:3. Then the algorithm needs to make a choice of which of these elements to migrate the extent of S_2:*employees*:1 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), suppose that in S_2 there are three elements S_2:*employees*:1, S_2:*employees*:2, S_2:*employees*:3, and in S_1 there are also three elements S_1:*employees*:1, S_1:*employees*:2, S_2:*employees*:3. Then a combination of the solutions for (a) and (b) needs to be applied.

Case 1
g_1: $addElem(S_2, \langle\langle B : 1 \rangle\rangle, [\{b\} \mid \{b\} \leftarrow \langle\langle A : B \rangle\rangle])$
g_2: $addList(S_2, \langle\langle A \rangle\rangle, \langle\langle B, \rangle\rangle, 2, [\{a, b, 2\} \mid \{a, b\} \leftarrow \langle\langle A, A : B \rangle\rangle])$
g_3: $addAttr(S_2, \langle\langle B \rangle\rangle, \langle\langle B : F \rangle\rangle, [\{f\} \mid \{f\} \leftarrow \langle\langle F \rangle\rangle])$
g_4: $extendAttr(S_2, \langle\langle B \rangle\rangle, \langle\langle B : G \rangle\rangle, Void, upperBound)$
g_5: $deleteAttr(S_2, \langle\langle A \rangle\rangle, \langle\langle A : B \rangle\rangle, [\{b\} \mid \{b\} \leftarrow \langle\langle B \rangle\rangle])$
s_1: $contractList(S_2, \langle\langle C \rangle\rangle, \langle\langle F \rangle\rangle, 1, [\{c, f, 1\} \mid \{c, f, o\} \leftarrow \langle\langle C, F, 1 \rangle\rangle], upperBound)$
s_2: $deleteElem(S_2, \langle\langle F \rangle\rangle, [\{f\} \mid \{f\} \leftarrow \langle\langle B : F \rangle\rangle])$
Case 2
g_1: $extendElem(S_2, \langle\langle B \rangle\rangle, Void, upperBound)$
g_2: $extendList(S_2, \langle\langle A \rangle\rangle, \langle\langle B \rangle\rangle, 2, Void, upperBound)$
g_3: $addAttr(S_2, \langle\langle B \rangle\rangle, \langle\langle B : F \rangle\rangle, [\{f\} \mid \{f\} \leftarrow \langle\langle F \rangle\rangle])$
g_4: $extendAttr(S_2, \langle\langle B \rangle\rangle, \langle\langle B : G \rangle\rangle, Void, upperBound)$
s_1: $contractList(S_2, \langle\langle C \rangle\rangle, \langle\langle F \rangle\rangle, 1, [\{c, f, 1\} \mid \{c, f, o\} \leftarrow \langle\langle C, F, 1 \rangle\rangle], upperBound)$
s_2: $deleteElem(S_2, \langle\langle F \rangle\rangle, [\{f\} \mid \{f\} \leftarrow \langle\langle B : F \rangle\rangle])$
Case 3
g_1: $extendAttr(S_2, \langle\langle B \rangle\rangle, \langle\langle B : E \rangle\rangle, Void, upperBound)$
Case 4
g_1: $addList(S_2, \langle\langle A \rangle\rangle, \langle\langle B \rangle\rangle, 2, [\{a, b, 2\} \mid \{a, c, o1\} \leftarrow \langle\langle A, C, 1 \rangle\rangle; \{c, b, o2\} \leftarrow \langle\langle C, B, 1 \rangle\rangle])$
s_1: $contractList(S_2, \langle\langle A, C, 1 \rangle\rangle, [\{a, c, 1\} \mid \{a, c, o\} \leftarrow \langle\langle A, C, 1 \rangle\rangle], upperBound)$
s_2: $contractList(S_2, \langle\langle, C, B, 1 \rangle\rangle, [\{c, b, 1\} \mid \{c, b, o\} \leftarrow \langle\langle C, B, 1 \rangle\rangle], upperBound)$
s_3: $contractElem(S_2, \langle\langle C \rangle\rangle, Void, upperBound)$
s_4: $renameList(S_2, \langle\langle A, B, 2 \rangle\rangle, 1)$
Case 5
g_1: $extendList(S_2, \langle\langle A \rangle\rangle, \langle\langle B \rangle\rangle, 1, [\{a, b, 1\} \mid \{a, r, o1\} \leftarrow \langle\langle r, A, 1 \rangle\rangle;$ $\{r, b, o2\} \leftarrow \langle\langle r, B, 2 \rangle\rangle], upperBound)$
s_1: $deleteList(S_2, \langle\langle r \rangle\rangle, \langle\langle B \rangle\rangle, 2, [\{r, b, 2\} \mid \{r, a, o1\} \leftarrow \langle\langle r, A, 1 \rangle\rangle; \{a, b, o2\} \leftarrow \langle\langle A, B, 1 \rangle\rangle])$
Case 6
g_1: $addList(S_2, \langle\langle r \rangle\rangle, \langle\langle B \rangle\rangle, 2, [\{r, b, 2\} \mid \{r, a, o1\} \leftarrow \langle\langle r, A, 1 \rangle\rangle; \{a, b, o2\} \leftarrow \langle\langle A, B, 1 \rangle\rangle])$
g_2: $extendList(S_2, \langle\langle B \rangle\rangle, \langle\langle A \rangle\rangle, 1, [\{b, a, 1\} \mid \{a, b, o\} \leftarrow \langle\langle A, B, 1 \rangle\rangle], upperBound)$
s_1: $deleteList(S_2, \langle\langle A \rangle\rangle, \langle\langle B \rangle\rangle, 1, [\{a, b, 1\} \mid \{b, a, o\} \leftarrow \langle\langle B, A, 1 \rangle\rangle])$
s_2: $renameList(S_2, \langle\langle r, B, 2 \rangle\rangle, 1)$

Table 3: Transformations for Figure 4.

3.1.2 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. In order to integrate the data sources, a slightly different version of the schema transformation algorithm is applied to the data source schemas in a pairwise fashion, in order to derive each one's union-compatible schema (Figure 5). The data source schemas LS_i are transformed into intermediate schemas, IS_i, so that they have the same structure. 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, $LS_1 = IS_1^1$. Then, the schema transformation algorithm is employed on IS_1^1 and LS_2 (see annotation 1 in Figure 5) The algorithm augments IS_1^1 with the constructs from LS_2 it does not contain. It also restructures LS_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 LS_2 is transformed to IS_2^1. The same process is performed between IS_2^1 and LS_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 LS_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 LS_i they do not contain. Finally, we automatically generate the union schemas, US_i, the `id` transformations between them, and the global schema by applying append semantics. The bottom-up integration of the data sources of Figure 2 is

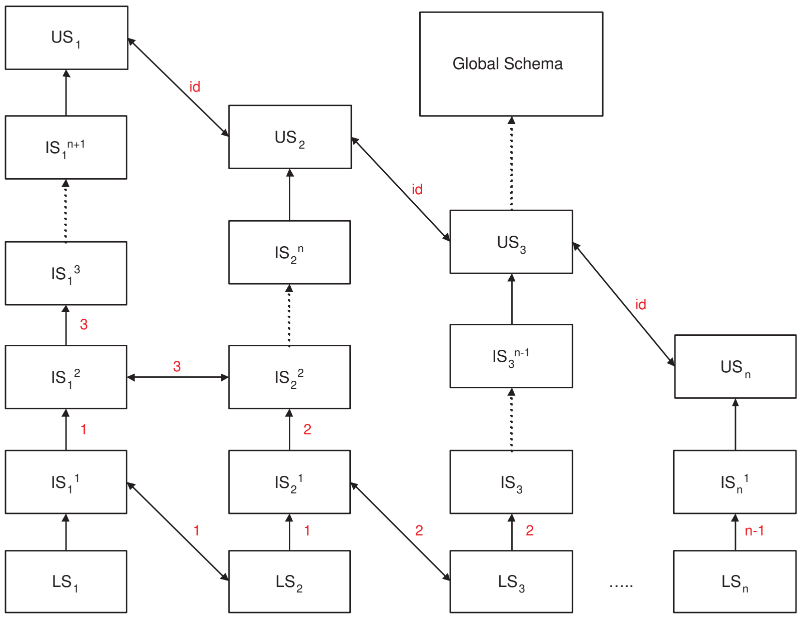

Figure 5: XML DataSource Schema integration

shown in Figure 6.

3.2 Querying XML Files

After the generation of transformation pathways by either top-down or bottom-up integration, queries posed on the global schema can now be evaluated. A query sent to AutoMed's query engine is first processed by the query processor, which is responsible for reformulating the input query into queries suitable for 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` transformation is encountered, the query processor replaces any occurrences of the deleted scheme by the query supplied with the `delete` transformation. As a result, the original query is turned into a query with multiple branches, each one suitable for each data source - see [23]. Note that, for the moment, querying of XML files is performed by DOM traversal. Future plans include XPath and XQuery support.

AutoMed's query engine and wrapper architecture are displayed in Figure 7. The AutoMedWrapperFactory and AutoMedWrapper classes are abstract classes providing some implementation, while the the `XMLWrapperFactory` and `XMLWrapper` classes implement the remaining abstract methods. Factories deal with model specific aspects, like primary keys for relational databases. The `XMLWrapperFactory` 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 or XML Schema the file references. A number of switches, such as a switch for collapsing whitespace, will be added in the future. As Figure 7 suggests, the whole architecture is extensible with wrappers for new data source models.

3.3 View Materialization Strategy

After the creation of the transformation pathways between the data source schemas and the global schema, there exists a direct connection between the data residing in the data sources and GS, through the transformation pathways and the queries they contain. Our framework provides an algorithm that can materialize GS into a new XML file. The algorithm traverses GS in a depth-first fashion, and obtains the necessary data by evaluating the individual schema constructs of GS as global queries. An issue that arises during this process is to determine the correct parent-child relationships, so that the resulting XML file precisely reflects the integration semantics. Our algorithm uses the edge constructs and schema- and instance-level unique IDs for this purpose. After materializing a schema element, E_P say, the algorithm retrieves the edge schema constructs that have E_P as the parent node. It retrieves

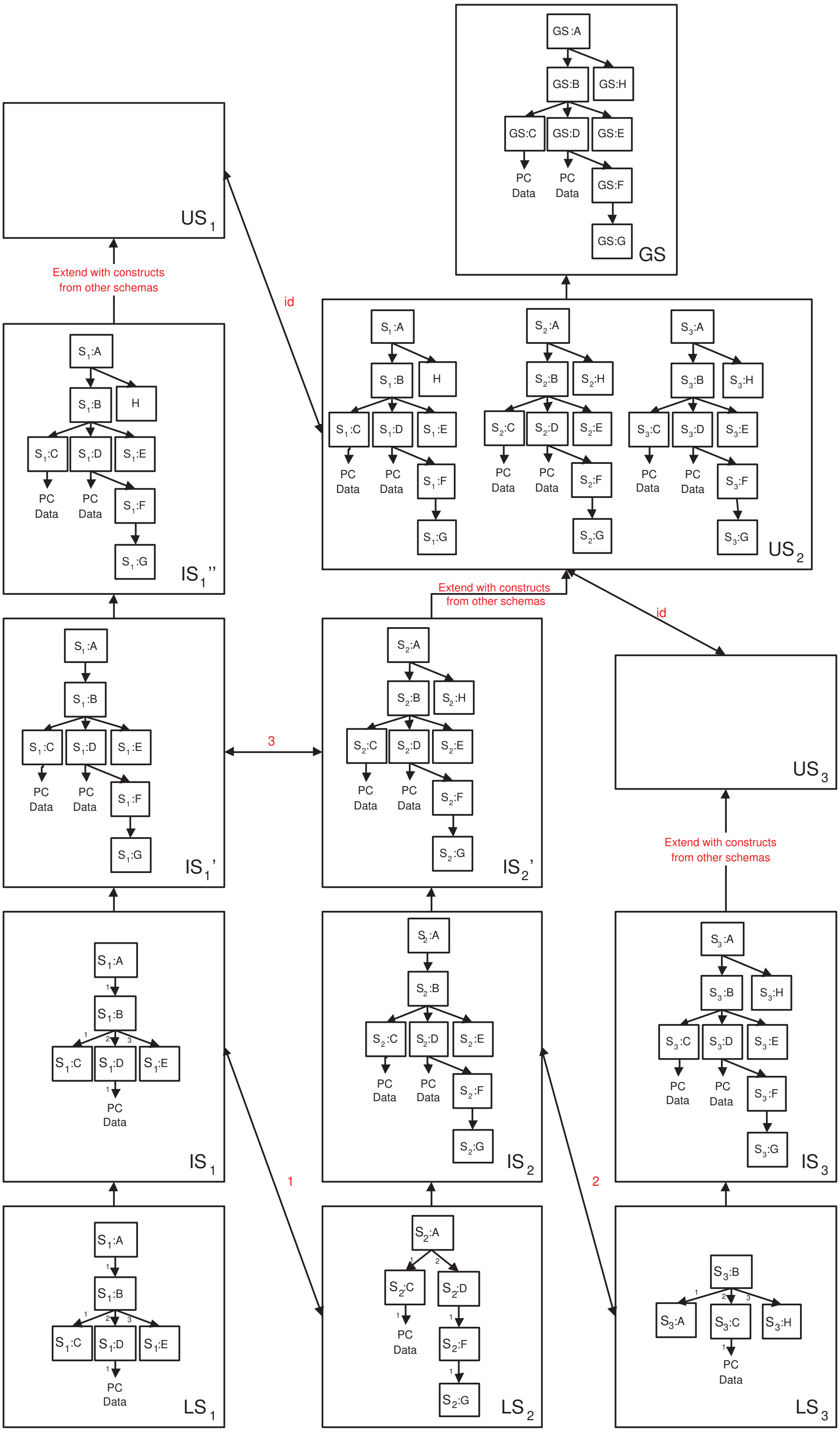

Figure 6: Bottom-up integration of the data sources of Figure 2

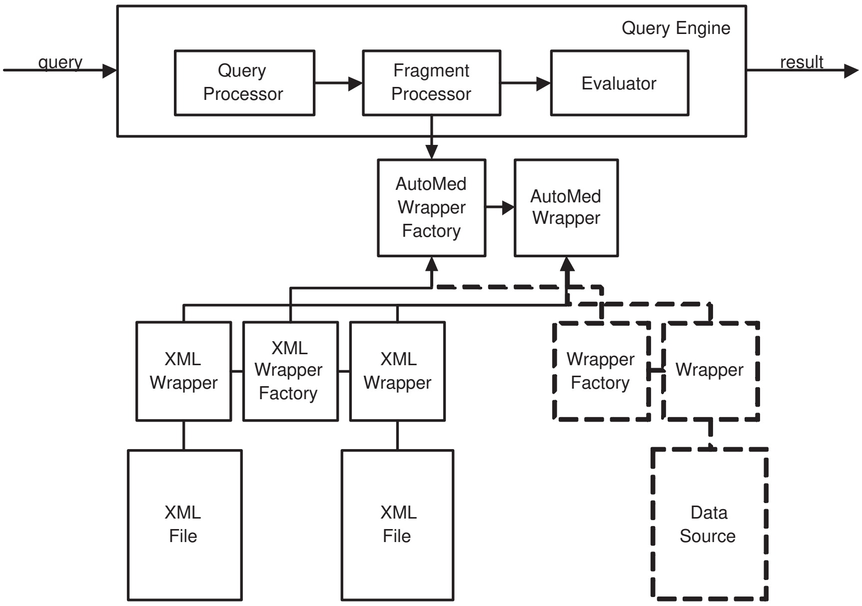

Figure 7: AutoMed's query engine & wrapper architecture

the extent of each of these constructs in turn, and inserts the children instances under the appropriate parent instances of E_P, as indicated by the instance-level unique IDs.

4 Related Work

There has been a lot of research effort concentrated on solving issues concerning XML and its connectivity with other models, mainly the relational model. The framework presented here aims at creating a complete solution for the integration of XML data, by focusing on finding XML-specific solutions.

Schema matching is a problem well-studied in a relational database setting. A seminal paper on schema matching, focusing on relational databases, but also outlining the general principles of the problem is [15]. A more recent survey, focused on an XML setting is [25]. In [14], the schema authors provide themselves the semantics of the schema elements, by providing mappings between elements of their schemas to a global ontology. These mappings are then used for query reformulation to produce data source specific queries. The ontology is then used for query reformulation, avoiding the need for a global schema. Other similar approaches are the ones described in [11] and [22]. The major problem with these approaches, and the approaches of this category as a whole, is that, if source A and B are mapped together, if B does not have some of the elements of A, then a user knowing only about schema B will never know about these elements.

Concerning schema integration, DIXSE [28] 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. [26] 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. [13] applies schema matching techniques on input DTDs, in order to create an integrated DTD from the sources' DTDs. The framework presented in this technical report approaches the schema integration problem using graph restructuring and is a purely XML solution. Furthermore, our approach allows for the use of multiple types of schema matching methods (use of ontologies, provision of semantics in the form of RDF, data-mining), which can all serve as an input to the schema integration algorithm.

In the context of XML views, an XML-specific tool is Active Views [1] that has advanced features like active rules for view updates, but is semi-automatic in that the user must program the creation of the view in a high-level language. WHAX [16] also defines views programmatically using WHAX-QL. Xyleme [6] offers automated view creation via tag, DTD and path mappings: the system exploits these mappings stored in the system to create the view whenever a user specifies its DTD. A similar approach is followed by [13]. A more straightforward approach to XML data integration is through XQuery [34]. The problem with this approach is automation: a user has to programmatically define the view instead of just defining its schema. MIX [2] and UXQuery [29] follow this approach, the former using its own query language (XMAS) and the latter using a subset of the XQuery language.

To our knowledge, there is no approach that considers the XML-specific problem of ordering policy when materializing views over multiple sources. Therefore all views are created by appending elements at the end of the parent's list of children. SilkRoute [8] implements an XML view materialization approach that supports ordering, but does so because the input is relational data, and is ordered using the `order by` SQL clause for all results prior to materialization. The same applies for XPERANTO [5], which provides a 'pure XML' middleware on top of an object-relational database. The term 'pure' is used because users do not need to know anything else than XML technologies to create and query views.

5 Concluding Remarks

This report presented a framework for the integration of XML data within the AutoMed heterogeneous data integration system. Assuming that a schema matching process has already occurred which has identified equivalent individual schema constructs, our schema transformation and view materialization algorithms succeed in integrating and materializing XML data sources automatically. Our algorithms make use of a simple schema definition language for XML data and a technique for assigning unique IDs to schema- and instance-level elements, both developed specifically for the purpose of XML data integration. The novelty of our algorithms lies in the use of XML-specific graph restructuring techniques applied to XML schemas. Future work will extend the schema transformation algorithm to cater for cases where there are multiple occurrences of an element name within an XML Datasource Schema, as discussed at the end of Section 3.1.1.

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 application of our schema transformation algorithm, P_T can then query P_S for the data it needs via its own schema, since AutoMed's query evaluator 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 an XML data source to change. In the AutoMed project, research has already focused on the schema evolution problem, both in the context of virtual data integration [20, 21] and materialized data integration [7]. For future work we will investigate the application of these general solutions specifically in the case of XML data. 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 having to regenerate them, when data source schemas are modified. These algorithms can be fully automatic if the information content of a data source schema contracts or remains the same, though require domain knowledge or human intervention if their information content expands.

The materialization algorithm also opens up several issues. One problem is that of respecting the data source schemas' constraints when creating the integrated XML file. For this, we can exploit constraints supplied within an DTD/XML Schema. However, an XML file may not reference a DTD/XML Schema, or the authors may not exploit the full capabilities of these languages. Moreover, even if such constraints exist, they determine intra-schema rather than inter-schema relationships. In cases of ambiguity, global schema constraints must be supplied. Another issue is supporting partial re-materialization of the global schema, after one or more data source schemas evolve.

The schema definition language used in our framework, XML DataSource Schema, will also be extended to capture the semantics of optional elements. This is because if a data source at first does not contain some optional elements, attributes or PCData sections, when these optional data appear, the XML Datasource Schema describing the data source will not be valid, and the problem will appear as a schema evolution problem.

Finally, the framework currently assumes up to now that the data sources are single XML files, and therefore each one is described by one XML DataSource Schema. However, we aim to include Native XML Databases as data sources in the future. In such a setting, a single data source may consist of multiple very similar XML files. The algorithm producing the XML DataSource Schema must be extended to handle such a case.

References

[1] S. Abiteboul, B. Amann, S. Cluet, E. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. In *Proc. VLDB'99*, pages 138–149, 1999.

[2] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *Proc. SIGMOD'99*, pages 597–599, 1999.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 17 2001.

[4] 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.

[5] 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.

[6] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *The VLDB Journal*, pages 271–280, 2001.

[7] H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. *Proc Int. Workshop on Data Warehousing and OLAP (DOLAP'03), New Orleans*, November 2003.

[8] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *SIGMOD Conference*, 2001.

[9] 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.

[10] R. Goldman and J. Widom. Summarizing and Searching Sequential Semistructured Sources. Technical report, March 2000.

[11] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the Structure Chasm.

[12] E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL Queries and Migrating Data in the AutoMed toolkit. Technical Report 20, Birkbeck College, June 2003.

[13] E. Jeong and C. Hsu. View Inference for Heterogeneous XML Information Integration. *Journal of Intelligent Information Systems (JIIS)*, 20(1):81–99, January 2003.

[14] L. Lakshmanan and F. Sadri. XML Interoperability. In *ACM SIGMOD Workshop on Web and Databases (WebDB), San Diego, CA*, pages 19–24, June 2003.

[15] J. Larson, S. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, April 1989.

[16] H. Liefke and S. Davidson. View Maintenance for Hierarchical Semistructured Data. In *Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[17] P. McBrien and A. Poulovassilis. Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach. In *Proc. ER'99, LNCS 1728*, pages 96–113, 1999.

[18] P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *LNCS*, volume 1626, pages 333–348. Springer-Verlag, June 1999.

[19] 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.

[20] P. McBrien and A. Poulovassilis. Schema Evolution In Heterogeneous Database Architectures, A Schema Transformation Approach. In *CAiSE*, pages 484–499, 2002.

[21] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *19th International Conference on Data Engineering*. ICDE, March 2003.

[22] L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. VLDB'02*, pages 598–609, 2002.

[23] A. Poulovassilis. The AutoMed Intermediate Query Langauge. Technical Report 2, Birkbeck College, June 2001.

[24] A. Poulovassilis and P. McBrien. A general formal framework for schema transformation. In *Data & Knowledge Engineering*, volume 28 of *1*, pages 47–71, 1998.

[25] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[26] C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogeneous Data Sources. In *IDEAS*, pages 199–208. IEEE Computer Society, July 2001.

[27] N. Rizopoulos. BAV transformations on relational schemas based on semantic relationships between attributes. Technical Report 22, Imperial College, August 2003.

[28] 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.

[29] V. Braganholo and S. Davidson and C. A. Heuser. UXQuery: building updatable XML views over relational databases. In *Proceedings of the Brazilian Symposium on Databases*, 2003.

[30] 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.

[31] W3C. XML Path Language (XPath). http://www.w3c.org/TR/xpath, November 1999.

[32] W3C. XML Schema Specification. http://www.w3c.org/TR/xmlschema-0, http://www.w3c.org/TR/xmlschema-1, http://www.w3c.org/TR/xmlschema-2, May 2001.

[33] W3C. Document Object Model (DOM). http://www.w3.org/DOM/, June 2002.

[34] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, November 2003.