

Information Sharing for the Semantic Web —

a Schema Transformation Approach

Version 3

Lucas Zamboulis and Alexandra Poulovassilis

School of Computer Science and Information Systems,

Birkbeck College, University of London, London WC1E 7HX

{lucas,ap}@dcs.bbk.ac.uk

February 15, 2006

Abstract

This paper proposes a framework for transforming and integrating heterogeneous XML data sources, making use of known correspondences from them to ontologies expressed in the form of RDFS schemas. Our algorithms generate schema transformation/integration rules which are implemented in the AutoMed heterogeneous data integration system. The paper first illustrates how correspondences to a single ontology can be exploited. The approach is then extended to the case where correspondences may refer to multiple ontologies, themselves interconnected via schema transformation rules. The contribution of this research is a set of automatic, XML-specific algorithms for the transformation and integration of XML data, making use of RDFS ontologies as a ‘semantic bridge’.

1 Introduction

The emergence of several new standards from the World Wide Web Consortium has provided a strong basis for enhancing the interoperability of web-based applications. XML is emerging as the common data interchange format, while RDF, RDFS and OWL provide a framework for assigning meaning to heterogeneous web-based data.

This paper proposes a framework for the automatic transformation and integration of heterogeneous XML data sources by exploiting known correspondences between them to ontologies expressed as RDFS

schemas. Our algorithms generate schema transformation/integration rules which are implemented in the AutoMed heterogeneous data integration system¹. These rules can be used to transform an XML data source into a target format, or to integrate a set of heterogeneous XML data sources into a common format. The transformation or integration may be *virtual*, in the sense that it allows queries posed on the target or common format to be answered by using the transformation rules and the data sources. Or the transformation/integration may be *materialised*, in the sense that the transformation rules can be used to generate from the data sources equivalent XML data that conforms to the target/common format.

There are two main advantages of our approach, compared with say constructing pairwise mappings between the XML data sources, or between each data source and some known global XML format. Firstly, known semantic correspondences between data sources and domain and other ontologies can be utilised for transforming or integrating the data sources. Secondly, the correspondences from the data sources to the ontology do not need to perform a complete mapping of the data sources. Furthermore, with our approach, changes in a data source, or addition or removal of a data source, do not affect the other sets of correspondences, thus promoting the reusability of sets of correspondences.

Paper outline: Section 2 compares our approach with related work. Section 3 gives an overview of AutoMed to the level of detail necessary for the purposes of this paper. Section 4 presents the process of transforming and integrating XML data sources which conform to the same ontology, while Section 5 extends this to the more general case of conformance to different ontologies. Section 6 gives our concluding remarks and plans for future work.

2 Related Work

The work in [4, 5] also undertakes data integration through the use of ontologies. However, this is by transforming the source data into a common RDF format, in contrast to our integration approach in which the common format is an XML schema. In [10], mappings from DTDs to RDF ontologies are used in order to reformulate path queries expressed over a global ontology to equivalent XML data sources. In [1], an ontology is used as a global virtual schema for heterogeneous XML data sources using LAV mapping rules. SWIM [3] uses mappings from various data models (including XML and relational) to RDF, in order to integrate data sources modelled in different modelling languages. In [11], XML Schema constructs are mapped to OWL constructs and evaluation of queries on the virtual OWL global schema are supported.

¹<http://www.doc.ic.ac.uk/automed/>

In contrast to all of these approaches, we use RDFS schemas merely as a ‘semantic bridge’ for transforming/integrating XML data, and the target/global schema is in all cases an XML schema.

Other approaches to transforming or integrating XML data which do not make use of RDF/S or OWL include [16, 18, 19, 20, 24]. Clio [16] translates data source schemas, XML or relational, into an internal representation and the mappings between the source and the target schemas are semi-automatically derived. Xyleme [18] also takes a mapping-based approach to XML data integration; it uses a tree structure as the global schema and source schemas are mapped to this via path mappings. DIXSE [19] transforms the DTD specifications of a set of source XML documents into an internal representation, using some heuristics to capture semantics as well as input from domain experts. Another DTD-dependent approach is presented in [20] which transforms XML documents using a set of transformations on documents’ DTDs, encoded in a transformation script; the target document is produced using this script and XSLT. The approach in [24] uses XML-specific transformations for XML schema integration, similarly to our approach. However, the semantics of these transformations are not captured within queries, as ours are, and they do not generate elements to preserve data that would have otherwise been lost during the transformation.

Finally, our own earlier work in [25, 26] also discussed the transformation and integration of XML data sources. However, this work was not able to make use of correspondences between the data sources and ontologies.

3 Overview of AutoMed

AutoMed is a heterogeneous data transformation and integration system which offers the capability to handle virtual, materialised and indeed hybrid data integration across multiple data models. It supports a low-level **hypergraph-based data model (HDM)** and provides facilities for specifying higher-level modelling languages in terms of this HDM. An HDM schema consists of a set of nodes, edges and constraints, and each modelling construct of a higher-level modelling language is specified as some combination of HDM nodes, edges and constraints. For any modelling language \mathcal{M} specified in this way (via the API of AutoMed’s Model Definitions Repository) AutoMed provides a set of primitive schema transformations that can be applied to schema constructs expressed in \mathcal{M} . In particular, for every construct of \mathcal{M} there is an **add** and a **delete** primitive transformation which add to/delete from a schema an instance of that construct. For those constructs of \mathcal{M} which have textual names, there is also a **rename** primitive transformation.

Instances of modelling constructs within a particular schema are identified by means of their *scheme* enclosed within double chevrons $\langle\langle \dots \rangle\rangle$. AutoMed schemas can be incrementally transformed by applying to them a sequence of primitive transformations, each adding, deleting or renaming just one schema construct (thus, in general, AutoMed schemas may contain constructs of more than one modelling language). A sequence of primitive transformations from one schema S_1 to another schema S_2 is termed a *pathway* from S_1 to S_2 and denoted by $S_1 \rightarrow S_2$. All source, intermediate, and integrated schemas, and the pathways between them, are stored in AutoMed’s Schemas & Transformations Repository.

Each **add** and **delete** transformation is accompanied by a query specifying the extent of the added or deleted construct in terms of the rest of the constructs in the schema. This query is expressed in a functional query language, IQL, and we will see some examples of IQL queries in Section 4. Also available are **extend** and **contract** primitive transformations which behave in the same way as **add** and **delete** except that they state that the extent of the new/removed construct cannot be precisely derived from the other constructs present in the schema. More specifically, each **extend** and **contract** transformation takes a pair of queries that specify a lower and an upper bound on the extent of the 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.

The queries supplied with primitive transformations can be used to translate queries or data along a transformation pathway $S_1 \rightarrow S_2$ by means of query unfolding: for translating a query on S_1 to a query on S_2 the **delete**, **contract** and **rename** steps are used, while for translating data from S_1 to data on S_2 the **add**, **extend** and **rename** steps are used — we refer the reader to [14] for details.

The queries supplied with primitive transformations also provide the necessary information for these transformations to be automatically *reversible*, in that each **add/extend** transformation is reversed by a **delete/contract** transformation with the same arguments, while each **rename** is reversed by a **rename** with the two arguments swapped.

As discussed in [15], this means that AutoMed is a **both-as-view (BAV)** data integration system: the **add/extend** steps in a transformation pathway correspond to Global-As-View (GAV) rules since they incrementally define target schema constructs in terms of source schema constructs; while the **delete** and **contract** steps correspond to Local-As-View (LAV) rules since they define source schema constructs in terms of target schema constructs. If a GAV view is derived from solely **add** steps it will be *exact* in the terminology of [12]. If, in addition, it is derived from one or more **extend** steps using their lower-bound (upper-bound) queries, then the GAV view will be *sound (complete)* in the terminology of [12]. Similarly,

if a LAV view is derived solely from `delete` steps it will be exact. If, in addition, it is derived from one or more `contract` steps using their lower-bound (upper-bound) queries, then the LAV view will be *complete* (*sound*) in the terminology of [12].

An in-depth comparison of BAV with the GAV, LAV and GLAV [6, 13] approaches to data integration can be found in [15, 9], while [14] discusses the use of BAV in a peer-to-peer data integration setting.

3.1 Representing XML schemas in AutoMed

The standard schema definition languages for XML are DTD [21] and XML Schema [22]. Both of these provide grammars to which conforming documents adhere, and do not abstract the tree structure of the actual documents. In our schema transformation and integration context, knowing the actual structure facilitates schema traversal, structural comparison between a source and a target schema, and restructuring of the source schema(s) that are to be transformed and/or integrated. Moreover, such a schema type means that the queries supplied with the AutoMed primitive transformations are essentially path queries, which are easily generated and easily translated into XPath/XQuery for interaction with the XML data sources. In addition, it may not be the case that the all data sources have an accompanying DTD or XML Schema they conform to.

We have therefore defined a simple modelling language called *XML DataSource Schema* (XMLDSS) which summarises the structure of an XML document. XMLDSS schemas consist of four kinds of constructs:

Element: Elements, e , are identified by a scheme $\langle\langle e \rangle\rangle$ and are represented by nodes in the HDM.

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

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

of e_c is associated with precisely one instance of e_p .²

PCData: In any XMLDSS schema there is one construct with scheme $\langle\langle\text{PCData}\rangle\rangle$, representing all the instances of PCData within an XML document.

In an XML document there may be elements with the same name occurring at different positions in the tree. In XMLDSS schemas we therefore use an identifier of the form $elementName\$count$ for each element in the schema, where $count$ is a counter incremented every time the same $element-Name$ is encountered in a depth-first traversal of the schema. If the suffix $\$count$ is omitted from an element name, then the suffix $\$1$ is assumed. For the XML documents themselves, our XML wrapper generates a unique identifier of the form $elementName\$count\&instanceCount$ for each element where $instanceCount$ is a counter identifying each instance of $elementName\$count$ in the document.

The XMLDSS schema, S , of an XML document, D , is derived by our XML wrapper by means of a depth-first traversal of D and is equivalent to the tree resulting as an intermediate step in the creation of a minimal dataguide [7]. However, unlike dataguides, we do not merge common sub-trees and the schema remains a tree rather than a DAG. Therefore the complexity of XMLDSS schema derivation is $O(N \times F)$ where N is the number of elements in the source XML document and F is the average fan-out of elements.

To illustrate XMLDSS schemas, consider the following XML document, named University.xml:

```
<university>
  <school name="School of Law">
    <academic>
      <name>Dr. Nicholas Petrou</name>
      <office>123</office></academic>
    <academic>
      <name>Prof. George Lazos</name>
      <office>111</office></academic>
  </school>
  <school name="School of Economics">
    <academic>
      <name>Dr. Anna Georgiou</name>
```

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

```

    <office>321</office></academic>
  </school>
</university>

```

The XMLDSS schema extracted from this document is S_1 in Figure 1. Note that a new root element r is generated for each XMLDSS schema, populated by a unique instance $r\&1$. This is useful in adopting a more uniform approach to schema restructuring and schema integration by not having to consider whether schemas have the same or different roots.

As mentioned earlier, after a modelling language has been specified in terms of the HDM, AutoMed automatically makes available a set of primitive transformations for transforming schemas defined in that modelling language. Thus, for XMLDSS schemas there are transformations `addElement` ($\langle\langle e \rangle\rangle, query$), `addAttribute` ($\langle\langle e, a \rangle\rangle, query$), `addNestList` ($\langle\langle\langle e_p, e_c, i \rangle\rangle, query$), and similar transformations for the `extend`, `delete`, `contract` and `rename` of `Element`, `Attribute` and `NestList` constructs.

3.2 Representing RDFS in AutoMed

RDFS [23] is a schema definition language for RDF which constrains the classes and properties that can be used to create RDF metadata, and indicates the relationships between these constructs. An RDFS schema can be represented in AutoMed using the following five kinds of constructs:

Class: An RDFS class, c , is identified by a scheme $\langle\langle c \rangle\rangle$, and classes are represented by nodes in the HDM.

Property: The `rdfs:property` class and its `rdfs:domain` and `rdfs:range` attributes are represented as a single construct. In particular, a property with name p , applying to instances of class c_d , and allowed to take values that are instances of class c_r , is identified by a scheme $\langle\langle p, c_d, c_r \rangle\rangle$. In the HDM, a property is represented by an edge with label p between the nodes representing c_d and c_r .

subClassOf: The `rdfs:subClassOf` RDFS schema construct is identified by a scheme $\langle\langle c_{sub}, c_{sup} \rangle\rangle$. In the HDM, this is represented by a constraint stating that class c_{sub} is a subclass of class c_{sup} .

subPropertyOf: The `rdfs:subPropertyOf` RDFS schema construct is identified by a scheme $\langle\langle p_{sub}, c_{d_1}, c_{r_1}, p_{sup}, c_{d_2}, c_{r_2} \rangle\rangle$. In the HDM, this is represented by a constraint stating that property $\langle\langle p_{sub}, c_{d_1}, c_{r_1} \rangle\rangle$ is a subproperty of property $\langle\langle p_{sup}, c_{d_2}, c_{r_2} \rangle\rangle$.

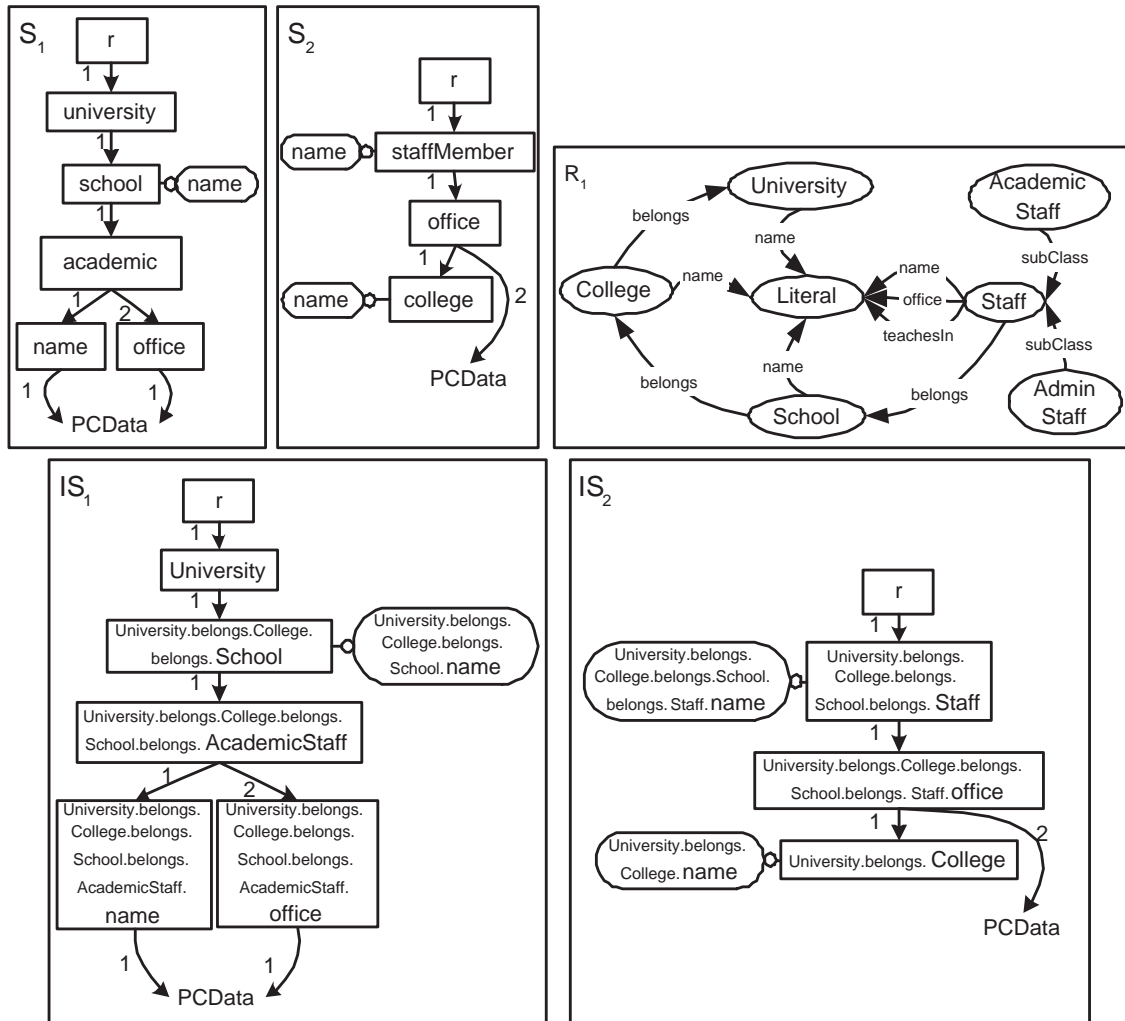


Figure 1: XMLDSS schemas S_1 and S_2 , RDFS schema R_1 and conformed schemas IS_1 and IS_2

Literal: In any AutoMed RDFS schema there is one construct with scheme $\langle\langle\text{Literal}\rangle\rangle$, representing all the instances of text. To link it with a class, we treat it as a class and use the `Property` construct to link it with `Class` constructs.

4 Transforming and Integrating XML Data Sources

In this section we consider first a scenario in which two XMLDSS schemas S_1 and S_2 are each semantically linked to an RDFS schema by means of a set of correspondences. These correspondences may be defined by a domain expert or extracted by a process of schema matching from the XMLDSS schemas and/or underlying XML data, e.g. using the techniques described in [17]. Each correspondence maps an XMLDSS `Element` or `Attribute` construct to an IQL query over the RDFS schema (so correspondences are LAV

mappings).

In Section 4.1 we first present a renaming algorithm which uses these correspondences to generate a transformation pathway from S_1 to an intermediate schema IS_1 , and a pathway from S_2 to an intermediate schema IS_2 . The schemas IS_1 and IS_2 are ‘conformed’ in the sense that they use the same terms for the same RDFS concepts.

Due to the bidirectionality of BAV, from these two pathways $S_1 \rightarrow IS_1$ and $S_2 \rightarrow IS_2$ can be automatically derived the reverse pathways $IS_1 \rightarrow S_1$ and $IS_2 \rightarrow S_2$.

In Section 4.2 we then describe a schema restructuring algorithm which automatically generates a transformation pathway from $IS_1 \rightarrow IS_2$. An overall transformation pathway from S_1 to S_2 can finally be obtained by composing the pathways $S_1 \rightarrow IS_1$, $IS_1 \rightarrow IS_2$ and $IS_2 \rightarrow S_2$.

This pathway can subsequently be used to automatically translate queries expressed on S_2 to operate on S_1 , using AutoMed’s XML Wrapper over source S_1 to return the query results. Or the pathway can be used to automatically transform data that conforms to S_1 to conform to S_2 and an XML document conforming to S_2 can be output.

In Section 4.3 we present an algorithm for the automatic integration of a number of XML data sources described by XMLDSS schemas S_1, \dots, S_n , each semantically linked to a single RDFS schema by a set of correspondences. This schema integration algorithm uses the algorithms of Sections 4.1 and 4.2 to integrate schemas S_1, \dots, S_n into a single global XMLDSS schema.

4.1 Renaming algorithm

In our context, a *correspondence* defines an **Element** or **Attribute** of an XMLDSS schema by means of an IQL *path query* over an RDFS schema. In particular, an **Element** e may map either to a **Class** c , or to a path ending with a class-valued property of the form $\langle\langle p, c_1, c_2 \rangle\rangle$, or to a path ending with a literal-valued property of the form $\langle\langle p, c, \text{Literal} \rangle\rangle$; additionally, the correspondence may state that the instances of a class are constrained by membership in some subclass. An **Attribute** may map either to a literal-valued property or to a path ending with a literal-valued property. Our correspondences are similar to path-path correspondences in [1], in the sense that a path from the root of an XMLDSS schema to a node corresponds to a path in the RDFS schema.

For example, Tables 1 and 2 show the correspondences between the XMLDSS schemas S_1 and S_2 and the RDFS schema R_1 (Figure 1). In Table 1 the 1st correspondence maps element $\langle\langle \text{university} \rangle\rangle$ to class $\langle\langle \text{University} \rangle\rangle$. The 2nd correspondence states that the extent of element $\langle\langle \text{school} \rangle\rangle$ corresponds to the

Table 1: Correspondences between XMLDSS schema S_1 and R_1

S_1	R_1
⟨⟨university⟩⟩	⟨⟨University⟩⟩
⟨⟨school⟩⟩	$[s \mid \{c, u\} \leftarrow \langle\langle\text{belongs, College, University}\rangle\rangle;$ $\{s, c\} \leftarrow \langle\langle\text{belongs, School, College}\rangle\rangle]$
⟨⟨school, name⟩⟩	$[s, l \mid \{c, u\} \leftarrow \langle\langle\text{belongs, College, University}\rangle\rangle;$ $\{s, c\} \leftarrow \langle\langle\text{belongs, School, College}\rangle\rangle;$ $\{s, l\} \leftarrow \langle\langle\text{name, School, Literal}\rangle\rangle]$
⟨⟨academic⟩⟩	$[s_2 \mid \{c, u\} \leftarrow \langle\langle\text{belongs, College, University}\rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle\text{belongs, School, College}\rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle\text{belongs, Staff, School}\rangle\rangle;$ $member\ s_2\ \langle\langle\text{AcademicStaff}\rangle\rangle]$
⟨⟨name⟩⟩	$[o \mid o \leftarrow generateElemUID\ 'name'$ $(count\ [l \mid \{c, u\} \leftarrow \langle\langle\text{belongs, College, University}\rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle\text{belongs, School, College}\rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle\text{belongs, Staff, School}\rangle\rangle;$ $member\ s_2\ \langle\langle\text{AcademicStaff}\rangle\rangle;$ $\{s_2, l\} \leftarrow \langle\langle\text{name, Staff, Literal}\rangle\rangle)]]$
⟨⟨office⟩⟩	$[o \mid o \leftarrow generateElemUID\ 'office'$ $(count\ [l \mid \{c, u\} \leftarrow \langle\langle\text{belongs, College, University}\rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle\text{belongs, School, College}\rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle\text{belongs, Staff, School}\rangle\rangle;$ $member\ s_2\ \langle\langle\text{AcademicStaff}\rangle\rangle;$ $\{s_2, l\} \leftarrow \langle\langle\text{office, Staff, Literal}\rangle\rangle)]]$

instances of class `School` derived from the join of properties $\langle\langle\text{belongs, College, University}\rangle\rangle$ and $\langle\langle\text{belongs, School, College}\rangle\rangle$ on their common class construct, `College`.³ In the 3rd correspondence, the IQL function `generateElemUID` generates as many instances for element $\langle\langle\text{name}\rangle\rangle$ as specified by its second argument i.e. the number of instances of the property $\langle\langle\text{name, Staff, Literal}\rangle\rangle$ in the path expression specified as the argument to the `count` function. In the 4th correspondence, element $\langle\langle\text{academic}\rangle\rangle$ corresponds to the instances of class `Staff` derived from the specified path expression and that are also members of `AcademicStaff`. The remaining correspondences in Tables 1 and 2 are similar.

Conforming a pair of XMLDSS schemas S_1 and S_2 to equivalent XMLDSS schemas IS_1 and IS_2 that represent the same RDFS concepts in the same way is achieved by applying the **renaming algorithm** given below to S_1 and to S_2 . Note that not all constructs of the XMLDSS schemas S_1 and S_2 need be mapped by correspondences to an ontology. Such constructs are not affected by the renaming algorithm

³The IQL query defining this correspondence may be read as “return all values s such that the pair of values $\{c, u\}$ is in the extent of construct $\langle\langle\text{belongs, College, University}\rangle\rangle$ and the pair of values $\{s, c\}$ is in the extent of construct $\langle\langle\text{belongs, School, College}\rangle\rangle$ ”. IQL is a comprehensions-based language and we refer the reader to [8] for details of its syntax, semantics and implementation. Such languages subsume query languages such as SQL and OQL in expressiveness [2]. There are AutoMed wrappers for SQL and OQL data sources and these translate fragments of IQL into SQL or OQL. Translating between fragments of IQL and XPath/XQuery is also straightforward — see Section 4.4 below.

Table 2: Correspondences between XMLDSS schema S_2 and R_1

S_2	R_1
$\langle\langle \text{staffMember, name} \rangle\rangle$	$[\{s_2, l\} \{c, u\} \leftarrow \langle\langle \text{belongs, College, University} \rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle \text{belongs, School, College} \rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle \text{belongs, Staff, School} \rangle\rangle;$ $\{s_2, l\} \leftarrow \langle\langle \text{name, Staff, Literal} \rangle\rangle]$
$\langle\langle \text{staffMember} \rangle\rangle$	$[s_2 \{c, u\} \leftarrow \langle\langle \text{belongs, College, University} \rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle \text{belongs, School, College} \rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle \text{belongs, Staff, School} \rangle\rangle]$
$\langle\langle \text{office} \rangle\rangle$	$[o o \leftarrow \text{generateElemUID 'office'}$ $(\text{count } [\{s_2, l\} \{c, u\} \leftarrow \langle\langle \text{belongs, College, University} \rangle\rangle;$ $\{s_1, c\} \leftarrow \langle\langle \text{belongs, School, College} \rangle\rangle;$ $\{s_2, s_1\} \leftarrow \langle\langle \text{belongs, Staff, School} \rangle\rangle;$ $\{s_2, l\} \leftarrow \langle\langle \text{office, Staff, Literal} \rangle\rangle])]$
$\langle\langle \text{college, name} \rangle\rangle$	$[\{c, l\} \{c, u\} \leftarrow \langle\langle \text{name, College, University} \rangle\rangle;$ $\{c, l\} \leftarrow \langle\langle \text{name, College, Literal} \rangle\rangle]$
$\langle\langle \text{college} \rangle\rangle$	$[c \{c, u\} \leftarrow \langle\langle \text{belongs, College, University} \rangle\rangle]$

and are treated as-is by the subsequent schema restructuring phrase.

For every correspondence i in the set of correspondences between an XMLDSS schema S and an ontology R , the **renaming algorithm** does the following:

1. If i concerns an **Element** e :
 - (a) If e maps directly to a **Class** c , **rename** e to c . If the instances of c are constrained by membership in a subclass c_{sub} of c , **rename** e to c_{sub} .
 - (b) Else, if e maps to a path in R ending with a class-valued **Property**, **rename** e to s , where s is the concatenation of the labels of the **Class** and **Property** constructs of the path, separated by ‘.’. If the instances of a **Class** c in this path are constrained by membership in a subclass, then the label of the subclass is used instead within s .
 - (c) Else, if e maps to a path in R ending with a literal-valued **Property** $\langle\langle p, c, Literal \rangle\rangle$, **rename** e as in step 1b, but without appending the label **Literal** to s .
2. If i concerns an **Attribute** a , then a must map to a path in R ending with a literal-valued **Property** $\langle\langle p, c, Literal \rangle\rangle$, and it is renamed as **Element** e in step 1c.

The application of this algorithm to schemas S_1 and S_2 in produces schemas IS_1 and IS_2 (Figure 1).

4.2 Schema restructuring

In order to next transform schema IS_1 to have the same structure as schema IS_2 , we have developed a **restructuring algorithm** that, given a source XMLDSS schema S and a target XMLDSS schema T , automatically transforms S to the structure of T , given that S and T have been previously conformed. This algorithm is able to use information that identifies an element/attribute in S to be equivalent to, a superclass of, or a subclass of an element/attribute in T . This information may be produced by, for example, a schema matching tool or, in our context here, via correspondences to an RDFS ontology.

Our restructuring algorithm consists of a **growing phase** where S is augmented with any constructs from T that it is missing, followed by a **shrinking phase** where it is reduced to remove any constructs now redundant. The algorithm for the growing phase is illustrated in Panels 1 and 2.

In the **growing phase**, every element e in the target XMLDSS schema T is visited in a depth-first fashion. If e is present in the source schema S , but under a different parent element than in T , a `NestList` construct is inserted from the correct parent to e in S (Proc1, line 6) — the existing incoming `NestList` construct to e in S will be removed later on in the shrinking phase. If e is not present in S , the missing element is inserted (Proc1, line 9) in one of four ways. First, we search the equivalent of $parent(e, T)$ ⁴ in S for an attribute a with the same name as e ; if the search succeeds, an attribute-to-element transformation is performed (Proc2, line 2). If no suitable attribute exists, but S contains elements corresponding to subclasses of the class to which e corresponds, e is inserted with an `add` transformation, using these elements to populate its extent (Proc2, line 5). Otherwise, if S contains elements corresponding to superclasses of e , e is inserted with an `extend` transformation, using `Void` as the lower-bound query, while for the upper-bound query the element corresponding to the class that is the lowest in the class hierarchy is used (Proc2, line 9). If all else fails, e is inserted with an `extend` transformation using `Void` as the lower-bound query, while the upper-bound query synthetically generates an extent for e (Proc2 line11); this synthetic extent is used to ensure that if e has child nodes or contains attributes, their extent will not be lost. If the user does not wish for synthetic extent generation, this option can be disabled.

Next, if there is a link from e to the `PCData` construct in T , one is also inserted in S , if not already present (Proc1, line 10). The attributes of e in T are then processed (Proc1, line11). These are treated in a similar manner to elements, in that if an attribute of e in T is not present in e in S , an element-to-attribute transformation is first attempted (Proc4, line 7), otherwise the attribute is inserted using subproperties (Proc4, line 12) or a superproperty (Proc4, line 18) to populate its extent. If all else fails,

⁴ $parent(e, T)$ denotes the parent element of element e in target schema T .

```

// Proc1: Growing Phase
1 for every element  $e$  in  $T$ , corresponding to class  $c$ , in a depth-first fashion do
2   - Let  $p_T$  be the parent element of  $e$  in  $T$  and  $p_S$  the element with the same label as  $p_T$  in  $S$ . if
    $e$  is present in  $S$  and the parent of  $e$  in  $S$  is not  $p_S$  then
3     |   Insert a NestList from  $p_S$  to  $e$  in  $S$ , populating its extent using the existing path from  $p_S$ 
     |   to  $e$ .
4   else if  $e$  is not present in  $S$  then
5     |   InsertMissingElement( $e$ )
6   ProcessPCDataLink( $e$ )
7   ProcessAttributes( $e$ )
// Proc2: InsertMissingElement( $e$ )
8 if  $p_S$  has an attribute  $a$  with the same label as  $e$  then
9   |   Attr2Element('e', $\langle\langle p_S, a \rangle\rangle$ , $\langle\langle p_S \rangle\rangle$ )
10 else if there are elements  $e_1 \dots e_n$  in  $S$ , corresponding to subclasses of  $c$ ,  $c_1 \dots c_n$ ,  $n \geq 1$  then
11   |   AddElemUsingSubClasses( $e$ , $\langle\langle e_1 \rangle\rangle \dots \langle\langle e_n \rangle\rangle$ )
12 else if  $S$  contains one or more elements corresponding to superclasses of  $c$ , choose element  $e'$ 
   corresponding to the class that is the lowest in the class hierarchy then
13   |   ExtendElemUsingSuperClass( $e, e'$ )
14 else
15   |   ExtendElement( $e, \langle\langle p_S \rangle\rangle$ )
// Proc3: ProcessPCDataLink( $e$ )
16 if  $e$  in  $T$  is connected to the PCData construct with a NestList construct while  $e$  in  $S$  is not then
17   |   Insert a NestList construct from  $e$  to the PCDataconstruct in  $S$  using the constants Void and
   |   Any
// Proc4: ProcessAttributes( $e$ )
18 for each attribute  $a$  of element  $e$  in  $T$ , corresponding to literal-valued property  $p_a$  do
19   if there is an element  $e'$  in  $S$ , that has the same label as  $a$ , is connected to the PCData
   construct and also the parent element of  $e'$ ,  $p_{e'}$  has the same label with  $e$  then
20   |   Element2Attr( $e, \langle\langle p_{e'}, e' \rangle\rangle, \langle\langle e', PCData \rangle\rangle$ )
21   else if there is no construct in  $S$  with the same label as  $a$  then
22     if  $S$  contains attribute constructs  $n_i$  that correspond to subproperties of  $p_a$  then
23       |   AddAttrUsingSubProp( $e, a,$ 
24       |    $n_1 \dots n_n$ )
25     else if  $S$  contains attribute constructs  $n_i$  that correspond to superproperties of  $p_a$ , choose
   attribute  $n$  corresponding to the property that is the lowest in the hierarchy then
26     |   ExtendAttrUsingSuperProp
27     |   ( $e, a, n$ )

```

Panel 1: Schema Restructuring Algorithm: Growing Phase (part 1)

there is no need to insert the attribute with an `extend` transformation and a synthetic extent (as is the case with elements), as missing attributes cannot cause further loss of data.

Note that in Proc1, line 6, an `add` or an `extend` transformation is issued, depending on the edges in the path from $parent(e, T)$ in S to e . In particular, if this path contains at some point an edge from a

```

// Proc5: Attr2Element('e',⟨⟨pS, a⟩⟩,⟨⟨pS⟩⟩)
28 - add e in S and a NestList from pS to e, populating their extents with queries that use the
    extents of ⟨⟨pS, a⟩⟩ and ⟨⟨pS⟩⟩
29 if e in T is connected to the PCData construct then
30   |   add a NestList from e to the PCData construct, using a query that generates its extent using
    |   the extents of ⟨⟨e⟩⟩ and ⟨⟨pS, a⟩⟩
// Proc6: AddElemUsingSubClasses(e, ⟨⟨e1⟩⟩...⟨⟨en⟩⟩)
31 - add e in S, populating its extent with the union of the extents of the elements ei. In the query
    supplied with the transformation, the instances of elements ei are renamed to have the same name
    as e.
32 - Insert a NestList from pS to e, populating its extent with the union of the extents derived from
    each path from pS to each ei.
33 if e in T is connected to PCData and elements ei in S are connected to PCData with NestLists ni
    then
34   |   add NestList ⟨⟨e, PCData⟩⟩ in S, and populate its extent with the union of the extents of ni.
// Proc7: ExtendElemUsingSuperClass(e, e')
35 - Insert e in S with an extend transformation. The lower-bound query supplied with the
    transformation is the constant Void, while the upper-bound one is the extent of e', with its
    instances renamed to have the same name as e.
36 - Insert a NestList from pS to e with an extend transformation. The lower-bound query supplied
    with the transformation is the constant Void, while the upper-bound one is the path from pS to e.
37 if both e in T and e' in S are connected to the PCData construct then
38   |   Insert NestList ⟨⟨e, PCData⟩⟩ in S with an extend transformation. The lower-bound query is
    |   the constant Void, while the upper-bound query uses the extents of ⟨⟨e⟩⟩ and ⟨⟨e', PCData⟩⟩.
// Proc8: ExtendElement(e, ⟨⟨pS⟩⟩)
39 - Insert element e into S using an extend transformation. The lower-bound query is the constant
    Void, while the upper-bound one is a query that generates an extent for e of the same size as pS.
40 - Insert a NestList from pS to e with an extend transformation. The lower-bound query is the
    constant Void, while the upper-bound one is a query that generates its extent using the extents of
    ⟨⟨pS⟩⟩ and ⟨⟨e⟩⟩.
// Proc9: Element2Attr(e, ⟨⟨pe', e'⟩⟩, ⟨⟨e', PCData⟩⟩)
41 - add ⟨⟨e', a⟩⟩ in S, using the extents of ⟨⟨pe', e'⟩⟩ and ⟨⟨e', PCData⟩⟩ and to populate its extent.
// Proc10: AddAttrUsingSubProp(e, n1...nn)
42 - add ⟨⟨e, a⟩⟩ in S using the extents of constructs ni to populate its extent.
// Proc11: ExtendAttrUsingSuperProp(e, a, n)
43 - Insert attribute ⟨⟨e, a⟩⟩ with an extend transformation; the lower-bound query is the constant
    Void, while the upper-bound one uses the extent of n.

```

Panel 2: Schema Restructuring Algorithm: Growing Phase (part 2)

child element *B* to a parent element *A* then an **extend** transformation will be used to insert the **NestList** from *parent(e, T)* in *S* to *e*. This is because in the data source of *S* there may be instances of *A* that do not have instances of *B* as children. Similar logic is applied in Proc6, line 5.

Also note that in Proc1, line 6, a **NestList** is inserted from the correct parent of *e* in source schema *S* to *e*. Together with the transformation of the shrinking phase that removes the old link to element *e* in *S*, these transformations perform a ‘move’ operation. Thus, instead of having to regenerate a subtree

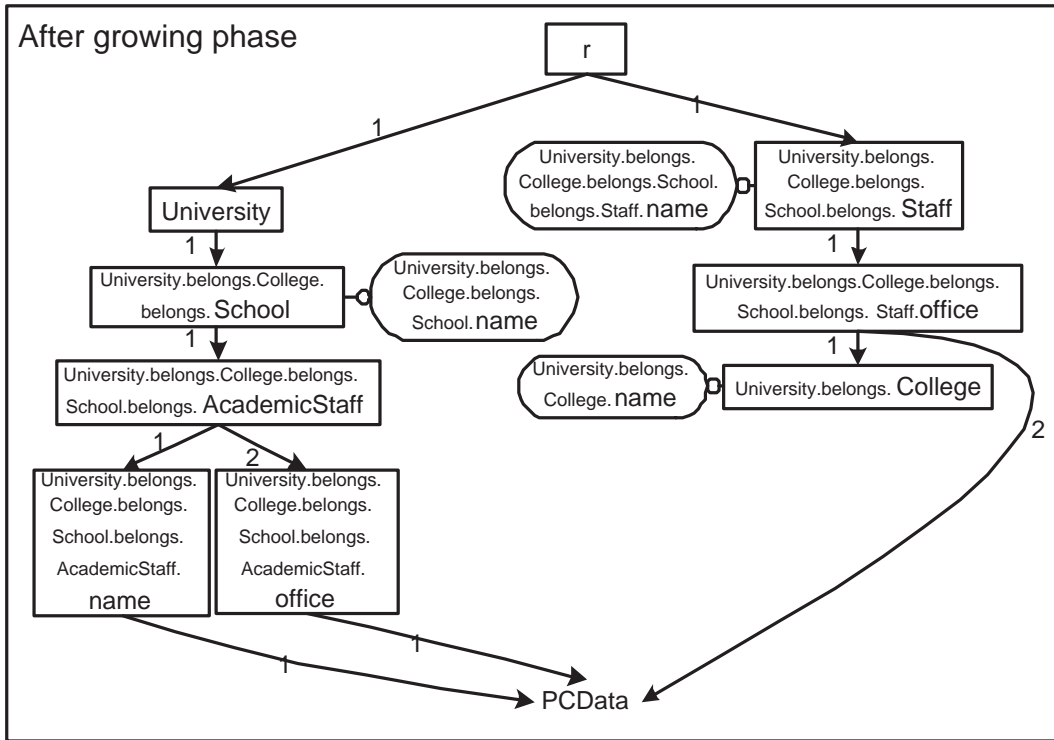


Figure 2: Applying the growing phase to schema IS_1 .

into a new position and remove it from the old one, the algorithm simply inserts a `NestList` from the subtree's new parent to the root of the subtree, and removes the `NestList` from the subtree's old parent.

The **shrinking phase** of the restructuring algorithm does the converse of the growing phase: S is now traversed in a depth-first fashion, and every construct present in S but not in T is removed with a `delete` transformation, if it is possible to describe its extent using the rest of the schema constructs, or with a `contract` transformation otherwise. In order to preserve the consistency of schemas, before removing any `Element`, its `Attribute` and incoming and outgoing `NestList` constructs are first removed. For reasons of space, and because the shrinking phase operates similarly to the growing phase, we have not given its pseudocode here.

In our running example, the transformations generated by the schema restructuring algorithm for transforming schema IS_1 to schema IS_2 are illustrated in Table 3. In the growing phase, the first three transformations concern the element $\langle\langle\text{Staff}\rangle\rangle$ of IS_2 . This element is inserted in IS_1 using `Element` $\langle\langle\text{AcademicStaff}\rangle\rangle$, which corresponds to a class that is a subclass of the class $\langle\langle\text{Staff}\rangle\rangle$ corresponds to in the RDFS ontology; the `ren` IQL function is used here to rename the instances of `Element` $\langle\langle\text{AcademicStaff}\rangle\rangle$ appropriately. After that, a `NestList` is inserted, linking $\langle\langle\text{Staff}\rangle\rangle$ to its parent, which is the root r , using the path from r to `AcademicStaff`. $\langle\langle\text{Staff}\rangle\rangle$ in T is not linked to the `PCData` construct, and there-

Table 3: Transformation pathways $IS_1 \rightarrow IS_2$. For readability, only the part of the name of an element/attribute needed to uniquely identify it within the schema is used.

Growing phase:
addElement($\langle\langle\text{Staff}\rangle\rangle, [\text{ren } a \text{ 'Staff'} a \leftarrow \langle\langle\text{AcademicStaff}\rangle\rangle]$)
addNestList($\langle\langle r, \text{Staff}, 2 \rangle\rangle, [\{r, s\} \{r, u\} \leftarrow \langle\langle r, \text{University}, 1 \rangle\rangle; \{u, s\} \leftarrow \langle\langle \text{University}, \text{School}, 1 \rangle\rangle; \{s, a\} \leftarrow \langle\langle \text{School}, \text{AcademicStaff}, 1 \rangle\rangle]$)
addAttribute($\langle\langle \text{Staff}, \text{name} \rangle\rangle, [\{o, p\} \{a, p\} \leftarrow [\{a, p\} \{a, n\} \leftarrow \langle\langle \text{AcademicStaff}, \text{name}, 1 \rangle\rangle; \{n, p\} \leftarrow \langle\langle \text{name}, \text{PCData}, 1 \rangle\rangle]; o \leftarrow [\text{ren } a \text{ 'Staff'}]]$)
addElement($\langle\langle \text{Staff.office} \rangle\rangle, [\text{ren } o \text{ 'Staff.office'} o \leftarrow \langle\langle \text{AcademicStaff.office} \rangle\rangle]$)
addNestList($\langle\langle \text{Staff}, \text{Staff.office}, 1 \rangle\rangle, [\{s, o2\} \{a, o1\} \leftarrow \langle\langle \text{AcademicStaff}, \text{AcademicStaff.office} \rangle\rangle; s \leftarrow [\text{ren } a \text{ 'Staff'}]; o2 \leftarrow [\text{ren } o1 \text{ 'Staff.office'}]]$)
addNestList($\langle\langle \text{Staff.office}, \text{PCData}, 2 \rangle\rangle, [\{o2, p\} \{o1, p\} \leftarrow \langle\langle \text{AcademicStaff.office}, \text{PCData}, 1 \rangle\rangle; o2 \leftarrow [\text{ren } o1 \text{ 'Staff.office'}]]$)
extendElement($\langle\langle \text{College} \rangle\rangle, \text{Void}, [c c \leftarrow \text{generateElemUID 'College' } \langle\langle \text{AcademicStaff.office} \rangle\rangle]$)
extendNestList($\langle\langle \text{Staff.office}, \text{College} \rangle\rangle, \text{Void}, [\{s, c\} \{s, c\} \leftarrow \text{generateNestLists } \langle\langle \text{Staff.office} \rangle\rangle \langle\langle \text{College} \rangle\rangle]$)
extendAttribute($\langle\langle \text{College}, \text{College.name} \rangle\rangle, \text{Void}, \text{Any}$)
Shrinking phase:
deleteNestList($\langle\langle r, \text{University} \rangle\rangle, [\{r \& 1, \text{University} \& 1\}]$)
contractNestList($\langle\langle \text{University}, \text{School} \rangle\rangle, \text{Void}, \text{Any}$)
contractElement($\langle\langle \text{University} \rangle\rangle, \text{Void}, \text{Any}$)
contractNestList($\langle\langle \text{School}, \text{AcademicStaff} \rangle\rangle, \text{Void}, \text{Any}$)
contractAttribute($\langle\langle \text{School}, \text{name} \rangle\rangle, \text{Void}, \text{Any}$)
contractElement($\langle\langle \text{School} \rangle\rangle, \text{Void}, \text{Any}$)
contractNestList($\langle\langle \text{AcademicStaff}, \text{AcademicStaff.name} \rangle\rangle, \text{Void}, [\{(\text{ren } o1 \text{ 'AcademicStaff'}), o2\} \{o1, o2, o3\} \leftarrow \text{skolemiseEdge } \langle\langle \text{Staff}, \text{Staff.name} \rangle\rangle \langle\langle \text{AcademicStaff.name} \rangle\rangle]$)
contractNestList($\langle\langle \text{AcademicStaff.name}, \text{PCData} \rangle\rangle, \text{Void}, [\{o2, o3\} \{o1, o2, o3\} \leftarrow \text{skolemiseEdge } \langle\langle \text{Staff}, \text{Staff.name} \rangle\rangle \langle\langle \text{AcademicStaff.name} \rangle\rangle]$)
contractElement($\langle\langle \text{AcademicStaff.name} \rangle\rangle, \text{Void}, [o2 \{o1, o2, o3\} \leftarrow \text{skolemiseEdge } \langle\langle \text{Staff}, \text{Staff.name} \rangle\rangle \langle\langle \text{AcademicStaff.name} \rangle\rangle]$)
contractNestList($\langle\langle \text{AcademicStaff}, \text{AcademicStaff.office} \rangle\rangle, \text{Void}, \langle\langle \text{Staff}, \text{Staff.office} \rangle\rangle$)
contractNestList($\langle\langle \text{AcademicStaff.office}, \text{PCData} \rangle\rangle, \text{Void}, \langle\langle \text{Staff.office}, \text{PCData} \rangle\rangle$)
contractNestList($\langle\langle \text{AcademicStaff.office} \rangle\rangle, \text{Void}, \langle\langle \text{Staff.office} \rangle\rangle$)
contractNestList($\langle\langle \text{AcademicStaff} \rangle\rangle, \text{Void}, \langle\langle \text{Staff} \rangle\rangle$)

fore its attribute is handled next. The `addAttribute` transformation performs an element-to-attribute transformation by inserting `Attribute` $\langle\langle \text{Staff}, \text{name} \rangle\rangle$ using the extents of $\langle\langle \text{AcademicStaff}, \text{name} \rangle\rangle$ and $\langle\langle \text{name}, \text{PCData} \rangle\rangle$. The following three transformations insert `Element` $\langle\langle \text{Staff.office} \rangle\rangle$ along with its incoming and outgoing `NestList` constructs in a similar manner. Then the last two transformations insert `Element` $\langle\langle \text{College} \rangle\rangle$ along with its `Attribute` and its incoming `NestList`. Since there is no information relevant to the extents of these constructs in S , `extend` transformations are used, with `Void` as the lower-bound query. Note however that the upper-bound query generates a synthetic extent for both the $\langle\langle \text{College} \rangle\rangle$ `Element` and its incoming `NestList` (for the latter, the IQL function `generateNestLists` is used⁵); this is to make sure that if any following transformations attach other constructs to $\langle\langle \text{College} \rangle\rangle$, their extent is not lost (assuming that these constructs are not themselves inserted with `extend` transformations and the constants `Void` and `Any` as the lower-bound and upper-bound queries). At the end of

⁵Generally, function `generateNestLists` either accepts `Element` schemes $\langle\langle a \rangle\rangle$ and $\langle\langle b \rangle\rangle$, with equal size of extents, and generates the extent of `NestList` construct $\langle\langle a, b \rangle\rangle$; or, it accepts `Element` schemes $\langle\langle a \rangle\rangle$ and $\langle\langle b \rangle\rangle$, where the extent of $\langle\langle a \rangle\rangle$ is a single instance, and generates the extent of `NestList` construct $\langle\langle a, b \rangle\rangle$.

the growing phase, the transformations applied to schema IS_1 result in the intermediate schema shown in Figure 2.

The shrinking phase operates similarly. The transformations removing $\langle\langle\text{AcademicStaff, AcademicStaff.name}\rangle\rangle$, $\langle\langle\text{AcademicStaff, PCData}\rangle\rangle$ and $\langle\langle\text{AcademicStaff.name}\rangle\rangle$ specify the inverse of the element-to-attribute transformation of the growing phase. To support attribute-to-element transformations, the IQL function *skolemiseEdge* is used; it takes as input a *NestList* $\langle\langle e_p, e_c \rangle\rangle$, and an *Element* $\langle\langle e \rangle\rangle$, which have the same extent size, and for each pair of instances e of $\langle\langle e \rangle\rangle$ and $\{e_p, e_c\}$ of $\langle\langle e_p, e_c \rangle\rangle$ generates a tuple $\{e_p, e, e_c\}$.

The result of applying the transformations of Table 3 to schema IS_1 is IS_2 illustrated in Figure 1. There now exists a transformation pathway $S_1 \rightarrow IS_1 \rightarrow IS_2 \rightarrow S_2$, which can be used to query S_2 by obtaining data from the data source corresponding to schema S_1 . For example, if this is the *University.xml* document in Section 3.1, the IQL query

$$[\{n, p\} | \{s, n\} \leftarrow \langle\langle\text{staffMember, name}\rangle\rangle; \{s, o\} \leftarrow \langle\langle\text{staffMember, office}\rangle\rangle; \{o, p\} \leftarrow \langle\langle\text{office, PCData}\rangle\rangle]$$

returns the following result:

$$[\{\text{'Dr. Nicholas Petrou'}, \text{'123'}\}, \{\text{'Prof. George Lazos'}, \text{'111'}\}, \{\text{'Dr. Anna Georgiou'}, \text{'321'}\}]$$

We could also use the pathway $S_1 \rightarrow IS_1 \rightarrow IS_2 \rightarrow S_2$ to materialise S_2 using the data from the data source corresponding to S_1 — see [26] for details of this process.

The separation of the growing phase from the shrinking phase ensures the *completeness* of the restructuring algorithm: the growing phase considers in turn each node in the target schema T and generates if necessary a query defining this node in terms of the source schema S ; conversely, the shrinking phase considers in turn each node of S and generates if necessary a query defining this node in terms of T ; inserting new target schema constructs before removing any redundant source schema constructs ensures that the constructs needed to define the extent of any construct are always present in the current schema.

The restructuring algorithm has a theoretical complexity of $O(N_T \log(N_S) + N_S \log(N_T))$, where N_S and N_T are the number of element and attribute constructs of S and T , respectively, assuming a logarithmic cost of searching the source/target schema during the growing/shrinking phase and generating a defining query if necessary. We have implemented these algorithms using Java SDK 1.4.2 and empirical results confirm the scalability of the algorithm. For example, the graph in Figure 3 shows the time taken to restructure a number of source schemas into a number of target schemas. These schemas are derived from IS_1 and IS_2 in Figure 1 by replicating the schema structure under the root element as many times as necessary. The sizes of both the source and target schema are increased by a constant factor each time

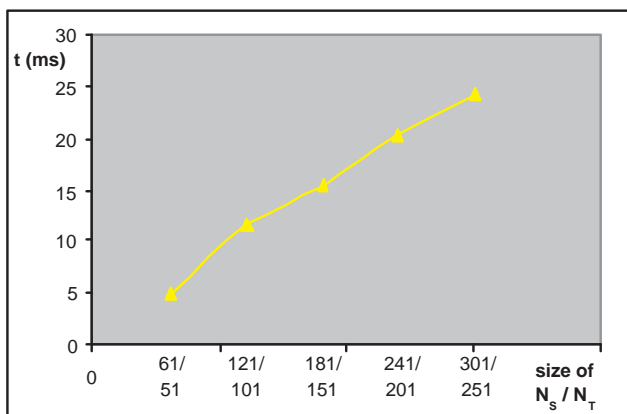


Figure 3: Performance graph for the schema restructuring algorithm.

(these sizes are shown along the x axis). Each restructuring was performed 1000 times and the resulting times averaged. Similar results are obtained for schemas whose structure is increased in a depth-wise rather than breadth-wise fashion.

We note that the times shown in this graph do not include the I/O cost of actually storing the transformations generated by the algorithm in the AutoMed repository. The number of transformations generated by the algorithm increases linearly according to $O(N_S + N_T)$ for the pairs of schemas shown in Figure 3 (these numbers are 222, 242, 662, 882 and 1102 respectively) and the I/O cost incurred increases accordingly. The time taken for I/O was typically 85% of the overall restructuring time for these particular schemas. The experiments were performed on a Pentium 4 with 1.5 Gb RAM running MS Windows XP, using Sun’s 1.4.2 JVM with 100 Mb of RAM, and PostgreSQL 8 as the database for the AutoMed repository.

In addition to its scalability, the schema restructuring algorithm has several other desirable properties: (i) It produces BAV pathways whose reversibility can then be exploited — this can be especially useful in peer-to-peer data integration settings [14]. (ii) The algorithm is able to use information that identifies an element/attribute in one data source to be equivalent to, a superclass of, or a subclass of an element/attribute in another data source. In our setting, this information is generated from correspondences between the data sources and ontologies. This allows more semantic relationships to be inferred between the data sources, and hence more information to be retained from a data source when it is transformed into a target format. (iii) The algorithm restructures a source schema into a target schema in an XML-specific manner, supporting a ‘move’ operation and attribute-to-element and element-to-attribute transformations.

Our own earlier work in [25, 26] also discussed the transformation and integration of XML data sources. However, this work was not able to make use of correspondences between the data sources and ontologies. It was however able to generate synthetic data to avoid the loss of information that may occur when parent-child relationships between elements are inverted. Another structural incompatibility which may lead to loss of data occurs when paths in the target schema contain **Element** constructs not present in the source schema, intermingled with **Element** constructs that are. This problem can also be tackled by generating new synthetic data using skolemisation. We are currently working on incorporating these further features into the schema restructuring algorithm presented here.

4.3 Schema integration

Consider now a setting where we need to integrate a set of XMLDSS schemas S_1, \dots, S_n all conforming to some ontology R into a global XMLDSS schema. The renaming algorithm of Section 4.1 can first be used to produce intermediate XMLDSS schemas IS_1, \dots, IS_n . We now need to integrate these intermediate schemas into a single global schema. This can be done as follows.

The initial global schema, GS_1 , is IS_1 . A **schema integration algorithm** (see below) is then applied to IS_2 and GS_1 , producing a new global schema GS_2 . IS_2 is then restructured to match GS_2 by applying the schema restructuring algorithm to IS_2 as the source and GS_2 as the target schema. Schemas IS_3, \dots, IS_n are integrated similarly: for each IS_i , we first apply the schema integration algorithm to IS_i and GS_{i-1} , producing the new global schema GS_i ; we then restructure IS_i to match the GS_i , by applying the schema restructuring algorithm to IS_i and GS_i .

The **schema integration algorithm** applied to each pair IS_i and GS_{i-1} has a growing and a shrinking phase. In the growing phase, the algorithm recreates the structure of IS_i under the root of GS_{i-1} by issuing **add** or **extend** transformations, thereby creating an intermediate global schema containing all constructs from IS_i and GS_{i-1} . In the shrinking phase, the algorithm removes any duplicate constructs now present in the intermediate global schema, using **delete** transformations, creating schema GS_i .

We note that the structure of the final global schema GS_n depends on the order of integration of IS_1, \dots, IS_n , in that it will be identical to IS_1 successively augmented with missing constructs appearing in IS_2, \dots, IS_n . If a different integration order is chosen, then the structure of the final global schema will be different, though containing the same **Element** and **Attribute** constructs.

4.4 Interacting with XML data sources

In our framework, XML data sources are accessed using the AutoMed toolkit’s XML Wrapper. This has three versions: a DOM wrapper supporting a subset of XPath a SAX wrapper to support large XML documents which could create memory problems for the DOM-based wrapper, and an XQuery wrapper over the eXist⁶ native XML repository. The latter translates IQL queries representing (possibly nested) select-project-join-union queries into (possibly nested) XQuery FLWR expressions.

The XML wrapper can be used in three different settings: (i) When a source XMLDSS schema S_1 has been transformed into a target XMLDSS schema S_2 , the resulting pathway $S_1 \rightarrow S_2$ can be used to translate an IQL query expressed on S_2 to an IQL query on S_1 , and the XML wrapper of the XML data source corresponding to S_1 can be used to retrieve the necessary data for answering the query. (ii) In the integration of multiple data sources with schemas S_1, \dots, S_n under a virtual global schema GS , AutoMed’s Global Query Processor can process an IQL query expressed on GS in cooperation with the XML wrappers for the data sources corresponding to the S_i . (iii) In a materialised data transformation or data integration setting, where the XML wrapper(s) of the data source(s) retrieve the data and the XML wrapper of the target schema materialises the data into the target schema format.

5 Handling Multiple Ontologies

We now discuss how our approach can also handle XMLDSS schemas that conform to different ontologies. These may be connected either directly via an AutoMed transformation pathway, or via another ontology (e.g. an ‘upper’ ontology) to which both ontologies are connected by an AutoMed pathway.

Consider in particular two XMLDSS schemas S_1 and S_2 that are semantically linked by two sets of correspondences C_1 and C_2 to two ontologies R_1 and R_2 . Suppose that there is an articulation between R_1 and R_2 , in the form of an AutoMed pathway between them. This may be a direct pathway $R_1 \rightarrow R_2$. Alternatively, there may be two pathways $R_1 \rightarrow R_{Generic}$ and $R_2 \rightarrow R_{Generic}$ linking R_1 and R_2 to a more general ontology $R_{Generic}$, from which we can derive a pathway $R_1 \rightarrow R_{Generic} \rightarrow R_2$ (due to the reversibility of pathways). In both cases, the pathway $R_1 \rightarrow R_2$ can be used to transform the correspondences C_1 expressed w.r.t. R_1 to a set of correspondences C'_1 expressed on R_2 . This is using the query translation algorithm mentioned in Section 3 which performs query unfolding using the `delete`, `contract` and `rename` steps in $R_1 \rightarrow R_2$.

⁶<http://exist.sourceforge.net/>

The result is two XMLDSS schemas S_1 and S_2 that are semantically linked by two sets of correspondences C'_1 and C_2 to the same ontology R_2 . Our algorithms described in Section 4 can now be applied. There is a proviso here that the new correspondences C'_1 must conform syntactically to the correspondences accepted as input by the renaming algorithm of Section 4.1 i.e. they conform to the syntax described in the first paragraph of Section 4.1. Determining necessary conditions for this to hold, and extending our approach to handle a more expressive set of correspondences, are areas of future work.

6 Concluding Remarks

This paper has presented algorithms for the transformation and integration of XML data sources that make use of known correspondences between them and one or more ontologies expressed as RDFS schemas. Our algorithms generate schema transformation/integration rules which are implemented in the AutoMed heterogeneous data integration system. These rules can then be used to perform virtual or materialised transformation and integration of the XML data sources.

The novelty of our approach lies in the use of XML-specific graph restructuring techniques in combination with correspondences from XML schemas to the same or different ontologies. The approach promotes the reuse of correspondences to ontologies and mappings between ontologies. It is applicable on any XML data source, be it an XML document or an XML database. The data source does not need to have an accompanying DTD or XML Schema, although if this is available it is straightforward to translate such a schema in our XMLDSS schema type. Although implemented using the AutoMed toolkit, our approach could readily be adapted to operate within data integration frameworks.

We envisage our approach as being particularly useful in peer-to-peer data sharing and data integration environments, and we are currently applying it in the context of biological data integration and biological workflows within the ISPIDER project⁷. In particular, for supporting workflow construction, we plan to use our methods in order to automatically transform the XML output produced by one web service into the correct format for input into another web service, making use of annotations provided on web service inputs and outputs.

For the future, we also plan to evaluate empirically the extensibility of our approach e.g. if a source or global XMLDSS schema evolves, or if an ontology and the set of correspondences to it evolve. Another issue is incrementally maintaining a materialised target or global XMLDSS schema.

⁷<http://www.ispider.man.ac.uk/>

References

- [1] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-based integration of XML web resources. In *Proc. International Semantic Web Conference 2002*, pages 117–131, 2002.
- [2] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
- [3] V. Christophides and et. al. The ICS-FORTH SWIM: A powerful Semantic Web integration middle-ware. In *Proc. SWDB'03*, 2003.
- [4] I. F. Cruz and H. Xiao. Using a layered approach for interoperability on the Semantic Web. In *Proc. WISE'03*, pages 221–231, 2003.
- [5] I. F. Cruz, H. Xiao, and F. Hsu. An ontology-based framework for XML semantic integration. In *Proc. IDEAS'04*, pages 217–226, 2004.
- [6] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proc. of the 16th National Conference on Artificial Intelligence*, pages 67–73. AAAI, 1999.
- [7] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistruc-tured databases. In *Proc. VLDB'97*, pages 436–445, 1997.
- [8] E. Jasper, A. Poulouvasilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. AutoMed Tech. Rep. 20, June 2003.
- [9] E. Jasper, N. Tong, P. Brien, and A. Poulouvasilis. View generation and optimisation in the AutoMed data integration framework. In *Proc. 6th International Baltic Conference on Databases & Information Systems, Riga, Latvia*, June 2004.
- [10] L. V. S. Lakshmanan and F. Sadri. XML interoperability. In *In Proc. of WebDB'03*, pages 19–24, June 2003.
- [11] P. Lehti and P. Fankhauser. XML data integration with OWL: Experiences and challenges. In *Proc. Symposium on Applications and the Internet (SAINT 2004), Tokyo*, 2004.
- [12] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246, 2002.

- [13] J. Madhavan and A. Halevy. Composing mappings among data sources. In *Proc. VLDB'03*, pages 572–583, 2003.
- [14] P. McBrien and A. Poulouvasilis. Defining peer-to-peer data integration using both as view rules. In *Proc. Workshop on Databases, Information Systems and Peer-to-Peer Computing (at VLDB'03), Berlin*, 2003.
- [15] P. McBrien and A. Poulouvasilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*. ICDE, March 2003.
- [16] L. Popa, Y. Velegrakis, R. Miller, M. Hernandez, and R. Fagin. Translating web data. In *Proc. VLDB'02*, pages 598–609, 2002.
- [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. Sirot, and D. Vodislav. Semantic integration of XML heterogeneous data sources. In *Proc. IDEAS*, pages 199–208, 2001.
- [19] P. Rodriguez-Gianolli and J. Mylopoulos. A semantic approach to XML-based data integration. In *Proc. ER'01*, pages 117–132, 2001.
- [20] H. Su, H. Kuno, and E. A. Rudensteiner. Automating the transformation of XML documents. In *Proc. WIDM'01*, pages 68–75, 2001.
- [21] W3C. Guide to the W3C XML specification (“XMLspec”) DTD, version 2.1, June 1998.
- [22] W3C. XML Schema Specification. <http://www.w3.org/XML/Schema>, May 2001.
- [23] W3C. RDF Vocabulary Description Language 1.0: RDF Schema. <http://www.w3.org/TR/rdf-schema/>, February 2004.
- [24] X. Yang, M. Lee, and T.W.Ling. Resolving structural conflicts in the integration of XML schemas: A semantic approach. In *Proc. ER'03*, pages 520–533, 2003.
- [25] L. Zamboulis. XML data integration by graph restructuring. In *Proc. BNCOD'04, LNCS 3112*, pages 57–71, 2004.

- [26] L. Zamboulis and A. Poulouvasilis. Using AutoMed for XML data transformation and integration. In *Proc. DIWeb'04 (at CAiSE'04), Riga, Latvia, June 2004*.