# Information Sharing for the Semantic Web — a Schema Transformation Approach

Lucas Zamboulis and Alexandra Poulovassilis

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

**Abstract.** This paper 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. 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 an XML-specific approach to the automatic transformation and integration of XML data, making use of RDFS ontologies as a 'semantic bridge'.

## 1 Introduction

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 rules implemented in the AutoMed heterogeneous data integration system (`http://www.doc.ic.ac.uk/automed/`). 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/integration may be virtual or materialised.

There are several 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: known semantic correspondences between data sources and domain and other ontologies can be utilised for transforming or integrating the data sources; the correspondences from the data sources to the ontology do not need to perform a complete mapping of the data sources; and changes in a data source, or addition or removal of a data source, do not affect the other 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 are linked to the same ontology, while Section 5 extends this to the more general case of data sources being linked 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.

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, 18–20, 23]. Our own earlier work in [24, 25] 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. The approach we present here 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. This information is generated from the 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.

## 3 Overview of AutoMed

AutoMed is a heterogeneous data transformation and integration system which offers the capability to handle virtual, materialised and 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 $\langle\langle \ldots \rangle\rangle$. 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 rest of the constructs. Each `extend` and `contract` transformation takes a pair of queries that specify a lower and an upper bound on the extent of the construct. These bounds may be `Void` or `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 while the `delete` and `contract` steps correspond to Local-As-View (LAV) rules. 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 for LAV views. 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 Data-Source Schema* (XMLDSS) which summarises the structure of an XML document. XMLDSS schemas consist of four kinds of constructs:

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

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

**NestList:** NestLists are parent-child relationships between two elements $e_p$ and $e_c$ and are identified by a scheme $\langle\langle \mathsf{e_p}, \mathsf{e_c}, \mathsf{i} \rangle\rangle$, 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 $e_p$.[1]

**PCData:** In any XMLDSS schema there is one construct with scheme $\langle\langle \mathsf{PCData} \rangle\rangle$, 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 $elementName$ 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.

To illustrate XMLDSS schemas, consider the following XML document:

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

```
<university>
    <school name="School of Law">
        <academic>
            <name>Dr. G. Grigoriadis</name>
            <office>123</office></academic>
        <academic>
            <name>Prof. A. Karakassis</name>
            <office>111</office></academic>
    </school>
    <school name="School of Medicine">
        <academic>
            <name>Dr. A. Papas</name>
            <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` ($\langle\langle \mathsf{e} \rangle\rangle$,*query*), `addAttribute` ($\langle\langle \mathsf{e}, \mathsf{a} \rangle\rangle$, *query*), `addNestList` (($\langle\langle \mathsf{e_p}, \mathsf{e_c}, \mathsf{i} \rangle\rangle$, *query*), and similar transformations for the `extend`, `delete`, `contract` and `rename` of `Element`, `Attribute` and `NestList` 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 show how these correspondences can be used 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 show how a transformation pathway from $IS_1 \rightarrow IS_2$ can then be automatically generated. An overall transformation pathway from $S_1$ to $S_2$ can finally be obtained by composing the three 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 is structured according to $S_1$ to be structured according to $S_2$, and an XML document structured according to $S_2$ can be output.

In Section 4.3 we discuss the automatic integration of a number of XML data sources described by XMLDSS schemas $S_1, \ldots, S_n$, each semantically linked to a single RDFS schema by a set of correspondences. This process extends the approach of Sections 4.1 and 4.2 to integrate a set of schemas into a single global XMLDSS schema.

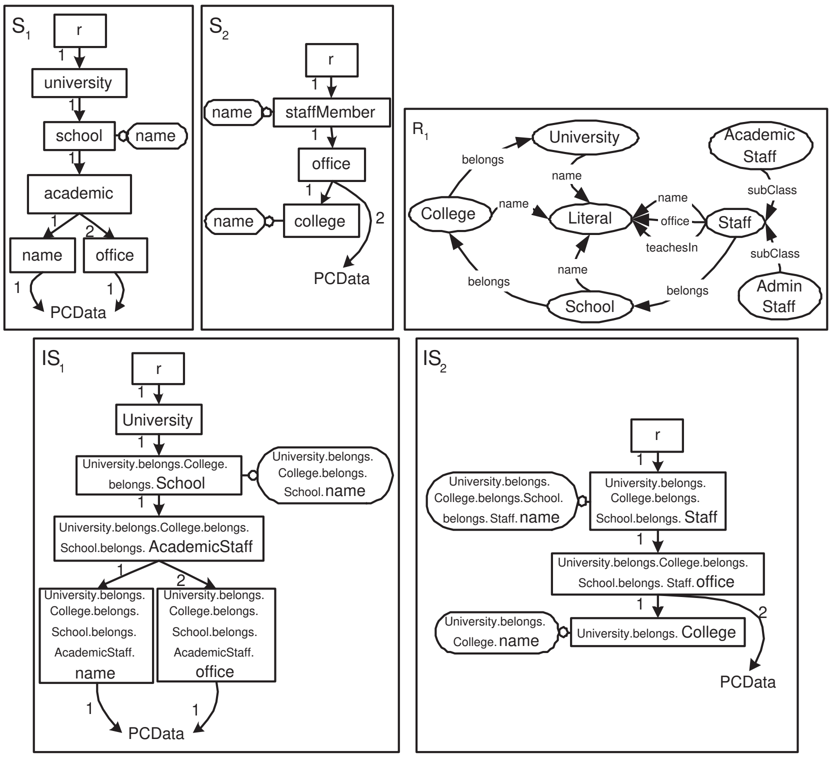


**Fig. 1.** XMLDSS schemas $S_1$ and $S_2$, RDFS schema $R_1$ and conformed XMLDSS schemas $IS_1$ and $IS_2$.

**Table 1.** Correspondences between XMLDSS schema $S_1$ and $R_1$

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

### 4.1 Schema Conformance

In our approach, a *correspondence* defines an `Element` or `Attribute` of an XMLDSS schema by means of an IQL *path query* over an RDFS schema[2]. 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 $\langle\langle \mathsf{p, c_1, c_2} \rangle\rangle$, or to a path ending with a literal-valued property of the form $\langle\langle \mathsf{p, c, Literal} \rangle\rangle$; 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 $\langle\langle$university$\rangle\rangle$ to class $\langle\langle$University$\rangle\rangle$. The 2nd correspondence states that the extent of element $\langle\langle$school$\rangle\rangle$ corresponds to the instances of class School derived from the join of properties $\langle\langle$belongs, College, University$\rangle\rangle$

[2] An RDFS schema can be represented in the HDM using five kinds of constructs: Class, Property, subClassOf, subPropertyOf, Literal. See [26] for details.

**Table 2.** Correspondences between XMLDSS schema $S_2$ and $R_1$

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

and ⟨⟨belongs, School,College⟩⟩ on their common class construct, College.[3] In the 4th correspondence, element ⟨⟨academic⟩⟩ corresponds to the instances of class Staff derived from the specified path expression and that are also members of AcademicStaff. In the 5th correspondence, the IQL function `generateElemUID` generates as many instances for element ⟨⟨name⟩⟩ as specified by its second argument i.e. the number of instances of the property ⟨⟨name, Staff, Literal⟩⟩ in the path expression specified as the argument to the *count* function. The remaining correspondences in Tables 1 and 2 are similar.

The conformance of a pair of XMLDSS schemas $S_1$ and $S_2$ to equivalent XMLDSS schemas $IS_1$ and $IS_2$ that represent the same concepts in the same way is achieved by renaming the constructs of $S_1$ and $S_2$ using the sets of correspondences from these schemas to a common ontology.

For every correspondence $i$ in the set of correspondences between an XMLDSS schema $S$ and an ontology $R$, a `rename` AutoMed transformation is generated, as follows:

1. If $i$ concerns an `Element` $e$:

[3] The 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 ⟨⟨belongs, College, University⟩⟩ and the pair of values $\{s, c\}$ is in the extent of construct ⟨⟨belongs, School, College⟩⟩". 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-92 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.

(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` $\langle\langle \mathsf{p}, \mathsf{c}, \mathsf{Literal} \rangle\rangle$, **`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` $\langle\langle \mathsf{p}, \mathsf{c}, \mathsf{Literal} \rangle\rangle$, and it is renamed as `Element` $e$ in step 1c.

Note that not all the constructs of $S_1$ and $S_2$ need be mapped by correspondences to the ontology. Such constructs are not affected and are treated as-is by the subsequent schema restructuring phase.

Figure 1 shows the schemas $IS_1$ and $IS_2$ produced by the application of the renamings to $S_1$ and $S_2$ arising from the sets of correspondences in Tables 1 and 2.

### 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 schema 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. We note that this algorithm is an extension of our earlier schema restructuring algorithm described in [25], which could only handle equivalence information between elements/attributes and could not exploit superclass and subclass information. The extended algorithm allows more semantic relationships to be inferred between $S$ and $T$, and hence more information to be retained from $S$ when it is transformed into $T$. The restructuring algorithm consists of a "growing phase" where $T$ is traversed in a depth-first fashion and $S$ is augmented with any constructs from $T$ that it is missing, followed by a "shrinking phase" where the augmented $S$ is traversed in a depth-first fashion and any contruct present in $S$ but not in $T$ is removed.

The AutoMed 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 $\langle\langle \mathsf{Staff} \rangle\rangle$ of $IS_2$. This element is inserted in $IS_1$ using `Element` $\langle\langle \mathsf{AcademicStaff} \rangle\rangle$, which corresponds to a class that is a subclass of the class $\langle\langle \mathsf{Staff} \rangle\rangle$ corresponds to in

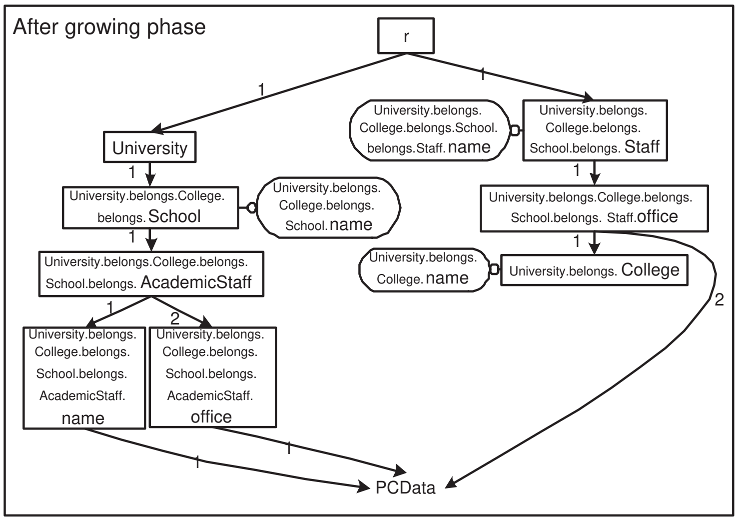


**Fig. 2.** Applying the growing phase to schema $IS_1$.

the RDFS ontology; the *ren* IQL function is used here to rename the instances of `Element` ⟨⟨AcademicStaff⟩⟩ appropriately. After that, a NestList is inserted, linking ⟨⟨Staff⟩⟩ to its parent, which is the root `r`, using the path from `r` to `AcademicStaff`. ⟨⟨Staff⟩⟩ in $T$ is not linked to the `PCData` construct, and therefore its attribute is handled next. The `addAttribute` transformation performs an element-to-attribute transformation by inserting `Attribute` ⟨⟨Staff, name⟩⟩ using the extents of ⟨⟨AcademicStaff, name⟩⟩ and ⟨⟨name, PCData⟩⟩. The following three transformations insert `Element` ⟨⟨Staff.office⟩⟩ along with its incoming and outgoing `NestList` constructs in a similar manner. Then the last two transformations insert `Element` ⟨⟨College⟩⟩ 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 ⟨⟨College⟩⟩ `Element` and its incoming `NestList` (for the latter, the IQL function `generateNestLists` is used[4]); this is to make sure that if any following transformations attach other constructs to ⟨⟨College⟩⟩, 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 the growing phase, the transformations applied to schema $IS_1$ result in the intermediate schema shown in Figure 2.

[4] Generally, function *generateNestLists* either accepts `Element` schemes ⟨⟨a⟩⟩ and ⟨⟨b⟩⟩, with equal size of extents, and generates the extent of `NestList` construct ⟨⟨a, b⟩⟩; or, it accepts `Element` schemes ⟨⟨a⟩⟩ and ⟨⟨b⟩⟩, where the extent of ⟨⟨a⟩⟩ is a single instance, and generates the extent of `NestList` construct ⟨⟨a, b⟩⟩.

**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($\langle\langle$Staff$\rangle\rangle$,$[ren\ a\ 'Staff' \mid a \leftarrow\ \langle\langle$AcademicStaff$\rangle\rangle]$) |
| addNestList($\langle\langle$r, Staff, 2$\rangle\rangle$,$[\{r,s\} \mid \{r,u\} \leftarrow \langle\langle$r, University, 1$\rangle\rangle$;<br>$\{u,s\} \leftarrow \langle\langle$University, School, 1$\rangle\rangle$; $\{s,a\} \leftarrow \langle\langle$School, AcademicStaff, 1$\rangle\rangle]$) |
| addAttribute($\langle\langle$Staff, name$\rangle\rangle$,<br>$[\{o,p\} \mid \{a,p\} \leftarrow [\{a,p\} \mid \{a,n\} \leftarrow \langle\langle$AcademicStaff, name, 1$\rangle\rangle$;<br>$\{n,p\} \leftarrow \langle\langle$name, PCData, 1$\rangle\rangle]$; $o \leftarrow [ren\ a\ 'Staff']]$) |
| addElement($\langle\langle$Staff.office$\rangle\rangle$,$[ren\ o\ 'Staff.office' \mid o\ \leftarrow\ \langle\langle$AcademicStaff.office$\rangle\rangle]$) |
| addNestList($\langle\langle$Staff, Staff.office, 1$\rangle\rangle$,<br>$[\{s, o2\} \mid \{a, o1\} \leftarrow \langle\langle$AcademicStaff, AcademicStaff.office$\rangle\rangle$;<br>$s \leftarrow [ren\ a\ 'Staff']; o2 \leftarrow [ren\ o1\ 'Staff.office']]$) |
| addNestList($\langle\langle$Staff.office, PCData, 2$\rangle\rangle$,<br>$[\{o2,p\} \mid \{o1,p\} \leftarrow \langle\langle$AcademicStaff.office, PCData, 1$\rangle\rangle$; $o2 \leftarrow [ren\ o1\ 'Staff.office']]$) |
| extendElement($\langle\langle$College$\rangle\rangle$,`Void`,<br>$[c \mid c \leftarrow\ generateElemUID\ \ 'College'\ \langle\langle$AcademicStaff.office$\rangle\rangle]$) |
| extendNestList($\langle\langle$Staff.office, College$\rangle\rangle$,`Void`<br>$[\{s,c\} \mid \{s,c\} \leftarrow\ generateNestLists\ \langle\langle$Staff.office$\rangle\rangle\ \langle\langle$College$\rangle\rangle]$) |
| extendAttribute($\langle\langle$College, College.name$\rangle\rangle$,`Void,Any`) |
| **Shrinking phase:** |
| deleteNestList($\langle\langle$r, University$\rangle\rangle$,$[\{r\$1\&1, University\$1\&1\}]$) |
| contractNestList($\langle\langle$University, School$\rangle\rangle$,`Void,Any`) |
| contractElement($\langle\langle$University$\rangle\rangle$,`Void,Any`) |
| contractNestList($\langle\langle$School, AcademicStaff$\rangle\rangle$,`Void,Any`) |
| contractAttribute($\langle\langle$School, name$\rangle\rangle$,`Void,Any`) |
| contractElement($\langle\langle$School$\rangle\rangle$,`Void,Any`) |
| contractNestList($\langle\langle$AcademicStaff, AcademicStaff.name$\rangle\rangle$,`Void`,<br>$[\{(ren\ o_1\ 'AcademicStaff'), o_2\} \mid \{o_1, o_2, o_3\} \leftarrow$<br>$skolemiseEdge\ \langle\langle$Staff, Staff.name$\rangle\rangle\ \langle\langle$AcademicStaff.name$\rangle\rangle]$) |
| contractNestList($\langle\langle$AcademicStaff.name, PCData$\rangle\rangle$,`Void`,<br>$[\{o_2, o_3\} \mid \{o_1, o_2, o_3\} \leftarrow\ skolemiseEdge\ \langle\langle$Staff, Staff.name$\rangle\rangle\ \langle\langle$AcademicStaff.name$\rangle\rangle]$) |
| contractElement($\langle\langle$AcademicStaff.name$\rangle\rangle$,`Void`,<br>$[o_2 \mid \{o_1, o_2, o_3\} \leftarrow\ skolemiseEdge\ \langle\langle$Staff, Staff.name$\rangle\rangle\ \langle\langle$AcademicStaff.name$\rangle\rangle]$) |
| contractNestList($\langle\langle$AcademicStaff, AcademicStaff.office$\rangle\rangle$,`Void`,$\langle\langle$Staff, Staff.office$\rangle\rangle$) |
| contractNestList($\langle\langle$AcademicStaff.office, PCData$\rangle\rangle$,`Void`,$\langle\langle$Staff.office, PCData$\rangle\rangle$) |
| contractNestList($\langle\langle$AcademicStaff.office$\rangle\rangle$,`Void`,$\langle\langle$Staff.office$\rangle\rangle$) |
| contractNestList($\langle\langle$AcademicStaff$\rangle\rangle$,`Void`,$\langle\langle$Staff$\rangle\rangle$ |

The shrinking phase operates similarly. The transformations removing $\langle\langle$AcademicStaff,AcademicStaff.name$\rangle\rangle$, $\langle\langle$AcademicStaff, PCData$\rangle\rangle$ and $\langle\langle$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` $\langle\langle \mathsf{e_p}, \mathsf{e_c} \rangle\rangle$, and an `Element` $\langle\langle \mathsf{e} \rangle\rangle$, which have the same extent size, and for each pair of instances $e$ of $\langle\langle \mathsf{e} \rangle\rangle$ and $\{e_p, e_c\}$ of $\langle\langle \mathsf{e_p}, \mathsf{e_c} \rangle\rangle$ 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 XML document of Section 3.1, the IQL query

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

returns the following result:

$[\{\text{'}Dr.\ G.\ Grigoriadis\text{'},\text{'123'}\}, \{\text{'}Prof.\ A.\ Karakassis\text{'},\text{'111'}\}, \{\text{'}Dr.\ A.\ Papas\text{'},\text{'321'}\}]$

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 [25] 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.

### 4.3 Schema integration

Consider now the integration of a set of XMLDSS schemas $S_1, \ldots, S_n$ 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 $IS_1, \ldots, IS_n$. The initial global schema, $GS_1$, is $IS_1$. $IS_2$ is then integrated with $GS_1$ producing $GS_2$. The integration of $IS_i$ with $GS_{i-1}$ to produce $GS_i$ proceeds until $i = n$. This integration consists of first an expansion of $GS_{i-1}$ with the constructs from $IS_i$ that it is missing (again via a growing and a shrinking phase) and then a restructuring, using the algorithm of Section 4.2, of $IS_i$ with the resulting schema $GS_i$.

### 4.4 Interacting with XML data sources

In our framework, XML data sources are accessed using an XMLDSS wrapper. This has SAX and DOM versions for XML files, supporting a subset of XPath. There is also a wrapper over the eXist XML repository which 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 $S_1, \ldots, S_n$ 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 are linked 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 $R_1 \rightarrow R_2$.

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 approach described for a single ontology 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 schema conformance process of Section 4.1 i.e. their syntax is as 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 discussed the automatic transformation and integration of XML data sources, making use of known correspondences between them and one or more ontologies expressed as RDFS schemas. 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.

The schema conformance algorithm handles 1-1 mappings between XMLDSS and RDFS constructs, enriched with containment relationships through the use of subclass/superclass and subproperty/superproperty RDFS constraints. This semantic reconciliation of the data source schemas is followed by their structural reconciliation by the schema restructuring algorithm, which handles 1-1 mappings between XMLDSS schemas, utilising the constraints defined in the correspondences. Extending our approach to be capable of utilising 1:n, n:1 and more complex mappings, is a matter of ongoing work. To this end, and at the same time aiming to maintain the current separation of semantic and structural schema reconciliation, we are currently extending the schema conformance algorithm.

## 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 middleware. 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 semistructured databases. In *Proc. VLDB'97*, pages 436–445, 1997.
8. E. Jasper, A. Poulovassilis, 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. Poulovassilis. 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.Poulovassilis. 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. Poulovassilis. 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. 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.
24. L. Zamboulis. XML data integration by graph restructuring. In *Proc. BNCOD'04, LNCS 3112*, pages 57–71, 2004.
25. L. Zamboulis and A. Poulovassilis. Using AutoMed for XML data transformation and integration. In *Proc. DIWeb'04 (at CAiSE'04), Riga, Latvia*, June 2004.
26. L. Zamboulis and A. Poulovassilis. Information sharing for the Semantic Web — a schema transformation approach. AutoMed Tech. Rep. 31, February 2006.