Transforming XML Documents as Schemas Evolve

Marcin Kwietniewski Jarek Gryz Stephanie Hazlewood Paul Van Run

York University and IBM CAS
Toronto
Canada

{marcin,jarek}@cs.yorku.ca

IBM
Toronto
Canada

{stephanie,pvanrun}@ca.ibm.com

ABSTRACT

Database systems often use XML schema to describe the format of valid XML documents. Usually, this format is determined when the system is designed. Sometimes, in an already functioning system, a need arises to change the XML schemas. In such a situation, the system has to transform the old XML documents so that they conform to the new format and that as little information as possible is lost in the process. This process is called *schema evolution*.

We have implemented an XML schema transformation toolkit within IBM Master Data Management Server (MDM). MDM uses XML documents to describe products that an enterprise may be offering to its clients. In this work we focus on evolving schemas rather than on integrating separate or heterogeneous data sources. Our solution includes an extendible schema matching algorithm that was designed with evolving XML schemas in mind and takes advantage of hierarchical structure of XML. It also includes a data transformation and migration method appropriate for environments where migration is performed in an abstraction layer above the DBMS. Finally, we describe a novel way of extending an XSLT editor with an XSLT visualization feature to allow the user's input and evaluation of the transformation.

1. INTRODUCTION

Database systems which store XML documents often impose constraints on those documents to make certain the data they represent makes sense in the context of the database. A recommended way of doing that is the use of XML schema, by which the format of valid XML documents can be specified.

Usually, the format of XML data which a computer system will store is determined when the system is designed along with the whole database schema. However, it is possible that in an already functioning system, a need will arise to change the XML schemas. Perhaps the users need to store some additional data or need to describe phenomena that cannot be described in the old format. In such a situation, apart from adjusting all software that dealt with the old data model, there is a need to transform the old XML documents in such a way that they conform to the new format and that as little information as possible is lost in the process. This process is called *schema evolution*.

We have implemented an XML schema transformation toolkit within IBM InfoSphere Master Data Management Server (MDM). MDM is an enterprise application that works on top of a relational database and provides a solution for managing customer, account, and product data centrally. It uses XML documents to describe products that an enterprise may be offering to its clients. In this work we focus on evolving schemas rather than integrating separate or heterogeneous data sources. When two schemas to be mapped come from a single database and describe the same concept they will, most likely, have a big overlap. It is desirable to exploit this property and to create a schema matching and mapping tool geared towards evolving schemas rather than those coming from very different sources. Indeed, by restricting the domain of possible XML documents in this way we are able to provide semi-automatic (and often fully automatic) transformation of XML schemas.

This work provides a comprehensive solution to the problem of XML schema evolution. Our system includes an extendible schema matching algorithm that was designed with evolving XML schemas in mind and takes advantage of hierarchical structure of XML. It also includes a data transformation and migration method appropriate for environments where migration is performed in an abstraction layer above the DBMS. Finally, we extend an XSLT editor with an XSLT visualization feature to allow the user's input and evaluation of the transformation. Although this work focuses on the MDM Server environment, the results should be applicable to other similar systems that manage XML documents.

2. BACKGROUND

Numerous approaches to schema matching have been proposed [2]. The focus in this area is primarily on automatic or semi-automatic discovery of correspondences between attributes of matched schemas. Fully automatic and reliable matching is impossible to achieve because of ambiguity and imprecision of data model information. Therefore, researchers aim to help the user as much as possible, especially in the tedious analysis of large schemas, where most attributes have obvious matches. Successful, modern data matching tools use multiple matching methods and combine their results to obtain the best match. Ideas from such tools could potentially be reused in the environment considered here. However, the focus of research in this area is on integrating separate, heterogeneous data sources over the web.

Proceedings of the VLDB Endowment, Vol. 3, No. 2

Such sources may have dramatically different schemas with only small parts in common. When two schemas to be mapped come from a single database and describe the same concept they will, most likely, have a big overlap. It is desirable to exploit this property to create a schema matching and mapping tool geared towards dealing with evolving schemas rather than those coming from heterogeneous, hence very different sources.

The most popular among XML specific query and transformation languages are XQuery and XSLT. XQuery was designed to provide concise syntax and querying capabilities. XSLT stylesheets were initially used to specify how an XML document is to be presented. Currently, W3C recommends XQuery as a language to be used by human programmers, while XSLT is better suited to be used and generated by programs. Thus, in this work, we use the latter.

In the field of static XML transformation type checking, interesting XML transformation models (languages) have been proposed, like XSLT0 and k-pebble transducers. Those approaches are presented in [5] and [6]. However, only a few methods deal with popular XML transformation languages, like XSLT and XQuery. Notable examples of static XSLT code analysis can be found in [1] and [3].

3. TRANSFORMATION TOOLKIT DESIGN

The schema evolution transformation toolkit presented in this demo consists of four parts: schema matcher, transform generator, data migration module, and an XSLT visualizing editor. In this paper, we focus on the description of the XSLT editor which also provides a user interface for the demo. Schema matcher and transform generator are briefly described below. The data migration module is omitted for lack of space.

Consider an introduction of a new specification (spec) format of a document. Two scenarios are possible here. First, the spec format can be changed in such a way that all the existing documents will conform to the new XML schema. An example of such a change is when the user adds some optional elements. In this case, no spec value migration is necessary.

The second scenario is more interesting. When the new version of the schema is not compatible with old one, the existing XML documents need to be migrated to remain usable. The system tries to automatically generate a transform to apply to existing documents to bring them to the new format. If it does not succeed, then the user must provide an XSLT using external tools and upload it to the server.

Assuming that the XSLT transform is available, the system estimates the amount of work that needs to be performed during the migration. If it is below some configured threshold, then the whole migration can be performed immediately after updating the spec format and within the same application server transaction. On the other hand, if there are too many spec values to migrate, the migration cannot be executed immediately and needs to be scheduled later (this is handled by the migration module). It can also be split into several sub-tasks. The whole schema evolution scenario ends when all spec values that were in the old format have been updated.

3.1 Transform Generation

The automatic transform generation module attempts to match XML elements between source and target schemas and to generate an executable transformation (XSLT stylesheet). Clearly, it is not always possible to create a matcher that would work in all scenarios. In general, we cannot always relieve the user of the need to inspect the matching or to manually create a transformation. However, in many situations the user may benefit from an automatically created matching, even if it is incomplete.

The design of the automated transform generation module is based on following principles:

- The generator is a “best-effort” tool. It attempts to create a complete mapping but in some cases it will only be able to generate a partial mapping.
- The matching algorithm has to be simple so that the user can understand why some nodes were matched and some were not.
- The matching routine is modular. It consists of several matching rules that are applied to source and target schemas. Therefore, it is configurable and extendable.
- Due to our assumption that source and target schemas have a substantial overlap (as we are interested in schema *evolution*) the matcher uses strongest matching rules first and removes matched attributes from consideration. This assures correctness of the matching and speeds up processing.

The transform generator consists of two parts, a *Matcher* and a *XSLT Generator*. The matcher is given two XML trees representing the source and target schema and produces a matching between them. The matching rules (identity, linguistic similarity, etc.) are pretty standard for XML matching tasks. The routine is recursive and attempts to match sets of children of two already matched nodes (see Figure 1). Then, using this information, XSLT generator creates an XSLT file describing the transform.

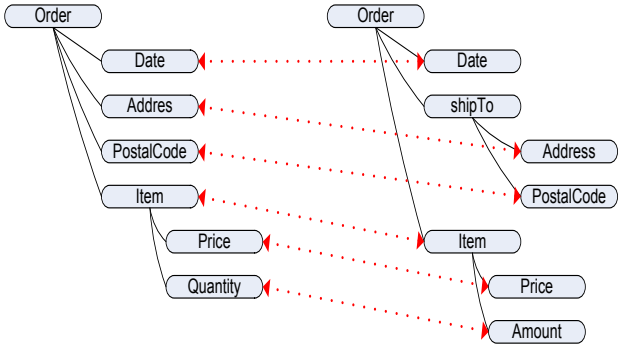

Figure 1 Source and target schema trees with final matching. Red dotted lines connecting two nodes represent they are matched.

XSLT generator is the second module of the transform generation tool. It works on the information provided by the schema matcher. The information includes source and target schema trees and matching information. The output of the generator is an executable XSLT stylesheet that when applied to a valid XML document in the source schema will output a valid XML document in the target schema that preserves all possible information contained in the input document.

3.2 XSLT Visualization

The last problem that we address in this work is that of static type checking or validation of XSLT stylesheets. This problem is important in the context of XML schema evolution, because creation of XSLT (or any other) transforms cannot be entirely automated with guarantees of their correctness. XSLT stylesheets are not particularly user-friendly, hence it may be hard for the user to follow the execution flow of more complex transforms. That is why it is desirable to know the format of outputs of particular XSLT. In other words, we would like to know the XML schema that defines the set of all possible output documents. In general, this problem, also known as static type checking for XSLT, is intractable [6][7]. However, we provide an approximate solution to it. Also, in the environment considered in this work the task is simpler than in the general case, because specification values in MDM system are more restricted than for arbitrary XML documents.

Assuming we have an algorithm for static type checking for XSL there still remains a question of how to present its result to the user. The simplest idea is just to compare the schema of the resulting document with the target schema and tell the user whether or not the former is contained in the latter (yes/no answer). Another idea is to show the user a graph representing the execution of an XSLT stylesheet, which gives the user an idea what the output format is (as in [1]). Finally, one may try to generate the schema constraining the output documents and let the user compare it to the desired one.

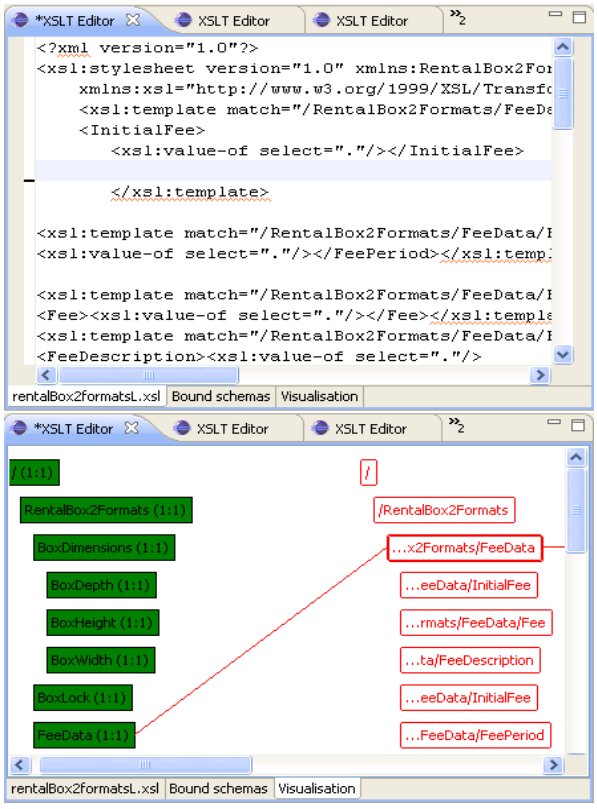

Figure 2 Visualizing editor. Top part of the figure shows the editor with the text editing tab active. The bottom part shows the editor with Visualization tab active. Both source schema tree and template execution tree are shown.

In our approach, we combine the last two ideas. We show both a representation of the execution flow and the resulting schema. Also, we combine the visualization with a simple text editor for transformations. The idea is to let the user navigate between the visual representation of the transformation and the code. Additionally, modifications to the transform are immediately reflected in the XSLT execution flow representation, as well as in the resulting schema. The editor is shown in Figure 2.

Both schemas and the structure of the execution flow of an XSL template are represented by trees. We chose "flattened" tree shape, which is frequently used to show file system directory structure as shown in Figure 3. For each node we include additionally the cardinality boundaries as defined in the schema, in the form "(*minimal # occurences*:*maximal # of occurences*)". For example, a mandatory node in a schema will be labeled with "(1:1)", while an optional node that can appear any number of times will be labeled with "(0:*)".

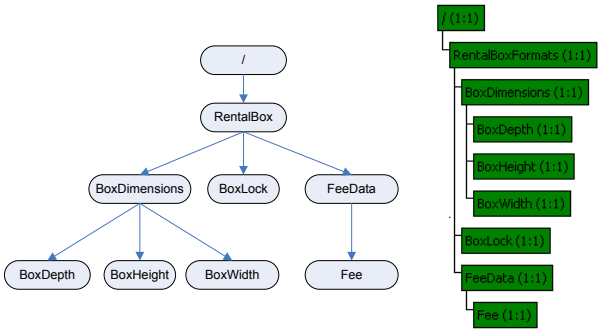

Figure 3 Two visual representations of the same XML schema tree. "Regular" representation on the left. "Flattened" representation on the right.

The complete Visualization tab contains all of the following:

- Source schema representation
- XSLT execution tree
- Transform result schema representation
- Target schema representation

An example of the visualization is shown in Figure 4.The user can click on any box representing a template to select it. Selected template box is highlighted (see "..x2Formats/FeeData" template in Figure 4). When a template box is selected the Visualizer displays lines connecting it with the element it matches in the source schema (line going to the left) and elements that it generates in the resulting schema (lines going to the right of the template). Additionally, the user may right-click on a template box to navigate to the template definition in the XSLT editor tab. This functionality should help the user quickly fix problems with a stylesheet. For example, when he sees that a template is generating wrong type of content the Visualizer will take him straight to that template.

Figure 4 XSLT visualization. From left to right: source schema, XSLT execution tree, resulting schema, target schema. Note: some template patterns are too long and don't fit entirely in a template box. In such case only suffix of the pattern is shown (…suffix)

The visualization tab has another feature that enables the user to immediately see potential problems with the transformation. Note that the resulting schema and the target schema are shown next to each other. If they were displayed in the same way as the source schema, that is, independently from each other, it could be hard for the user to see all differences between them. This problem is exacerbated when the schemas are huge, perhaps containing hundreds of attributes. To alleviate this problem the Visualizer aligns matching elements of both schemas. Figure 4 shows how this feature works. First five elements in the result and the target schema trees are identical, hence, they all are rendered next to each other. The only difference is that FeeData element of the resulting schema is optional (minOccurs = 0) while it is mandatory in the target schema. It means that the transform is not guaranteed to return this element and that indicates a potential problem with the transformation. The Visualizer uses a different color for optional and mandatory schema element so it is easy for the user to spot the difference.

Below the first five matching schema elements, there are five further elements that do not have a match: "FeeDescription", "InitialFee" and "FeePeriod" in the resulting schema and "Period" and "FeeDescr" in the target schema. Again, it is easy for the user to see that there is no corresponding element displayed next to any of those schema elements. The user can immediately identify the modifications in the code needed to fix the transformation. Finally, both the resulting and target schemas in Figure 4 include the "BoxDimensions''- elements and representations of those elements are displayed co-aligned. Their children, however, are not in the same order, which is clearly shown in the visualization tab.

4. DEMO PRESENTATION

Our demo presentation will be as follows. We will show a pair of XML schemas, representing an old schema, previously used in our database, and a new one that is being introduced to the system. We will show a graphical representation of the result of automatic matching of the two schemas and a generated XSL transform.

Then we will use the visualizing editor to show the execution flow of the aforementioned transform and how its output schema compares to the target schema.

Then we will proceed to another example. Again, we will present two subsequent versions of a schema with significant differences between them. We will also supply a partial, erroneous, transform between those schemas. We will show how it is possible to track and quickly fix bugs in the transform, using our visualizing editor. We will also fill in missing parts of the XSL stylesheet in several steps. As we add more templates to the stylesheet, we will be monitoring our progress using the visualizing editor.

We will show a fairly complex XML schema to represent an old version of a database. We will let the users make changes to the schema, reflecting what they think might be improved in it. Then, we will use the automatic transform generation tool to create an XSL transformation between the original and the updated schema. We will examine the resulting transform in the visualizing editor. If the stylesheet will not appear to produce correct output, we will use the editor to quickly fix the problems.

Finally, we will accept XML schema files from the audience and will demonstrate how our tools deal with them.

5. REFERENCES

[1] Moller, A., Olesen, M. O., Schwartzbach, M. I: Static validation of XSL transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29, 4 (2007), Article No. 21.

[2] Shvaiko, P., Euzenat, J.: A Survey of Schema-based Matching Approaches. *Journal on Data Semantics IV*, 146-171, Springer Berlin/Heidelberg, 2005

[3] Martens, W., Neven, F., Gyssens, M.: Typechecking Top-Down XML Tranformations: Fixed Input or Output Schemas *Information and Computation* 206 (7), pp. 806-827 (2008).

[4] Mong Li Lee, Liang Huai Yang, Wynne Hsu, and Xia Yang: XClust: clustering XML schemas for effective integration. In *Proceedings of the eleventh international conference on Information and knowledge management* (CIKM '02). ACM, New York, NY, USA, pp. 292-299.

[5] Tozawa, A. 2001. Towards static type checking for XSLT. In *Proceedings of the 2001 ACM Symposium on Document Engineering* (Atlanta, Georgia, USA, November 9-10, 2001). DocEng '01. ACM, New York, NY, pp. 18-27.

[6] Milo, T., Suciu, D., and Vianu, V. 2000. Typechecking for XML transformers. In *PODS* (Dallas, Texas, United States, May 15 - 18, 2000).. ACM, New York, NY, pp. 11-22.

[7] Martens, W. and Neven, F. 2004. Frontiers of tractability for typechecking simple XML transformations. In *PODS* '04 (Paris, France, June 14 - 16, 2004). ACM, New York, NY, pp. 23-34.