Structure–Conforming XML Document Transformation Based on Graph Homomorphism[*]

Tyng–Ruey Chuang
Institute of Information Science
Academia Sinica
Nangang 115, Taipei, Taiwan

Hui–Yin Wu[†]
Program in Digital Contents and Technologies
National Chengchi University
Wenshan 116, Taipei, Taiwan

ABSTRACT

We propose a principled method to specify XML document transformation so that the outcome of a transformation can be ensured to conform to certain structural constraints as required by the target XML document type. We view XML document types as graphs, and model transformations as relations between the two graphs. Starting from this abstraction, we use and extend graph homomorphism as a formalism for the specifications of transformations between XML document types. A specification can then be checked to ensure whether results from the transformation will always be structure–conforming.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*Markup languages*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph labeling*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Design, Theory, Verification

Keywords

Document Transformation, Graph Homomorphism, XML

1. INTRODUCTION

In XML document processing, one often faces the problem of ensuring the correctness of the structure of a program–generated document. As an example, Figure 1 shows the structural constraints of two XML document types (named DocBook Tiny and XHTML Tiny). Each of the two graphs show what kinds of elements can appear as the children of other kinds of elements. If an element of type `parent` can have elements of type `child` as its children, then there is an edge from node `parent` to `child` in the graph.

For the graphs in Figure 1, it is natural for us to find mappings between nodes that are semantically close. For example, we may transform `orderedlist` elements in DocBook Tiny into `ol` elements in XHTML Tiny, and `listitem` elements into `li` elements. It too seems fitting to map `para` elements into `p` elements. However, such a straightforward mapping could transform DocBook Tiny documents into ill–structured XHTML Basic documents. Take the following DocBook Tiny document fragment as an example.

```
<orderedlist>
 <orderedlist>
  <para>A preface here and ... </para>
 </orderedlist>
 <listitem><para>...A list item</para></listitem>
</orderedlist>
```

By the proposed straightforward mapping, this structure–conforming DocBook Tiny document fragment will be transformed into the following XHTML document fragment which is ill-structured.

```
<ol>
 <ol>
  <p>A preface here and ... </p>
 </ol>
 <li><p>...A list item</p></li>
</ol>
```

The output is ill–structured because according to the parent–child constraints for XHTML Tiny element types, `ol` element cannot have `ol` or `p` elements as children.

In this paper, we propose to use and extend graph homomorphism as a formalism for the specifications of mappings between XML document types, so that the results can be ensured to be always structure–conforming.

2. RELATED WORKS

There is a wealth of research on the modeling of XML document types, and on the techniques and languages for expressing XML document transformations. Since the late 1990s, there have been many works on using formal languages for modeling SGML/XML document editing and transformation [2, 4, 6, 7]. This paper continues these efforts by using graph homomorphisms, a formalism relatively new to the document engineering community.

[*]An extended version of this paper is freely available from the authors' websites under a Creative Commons License.
[†]Hui–Yin Wu is also a research student at the Institute of Information Science, Academia Sinica.

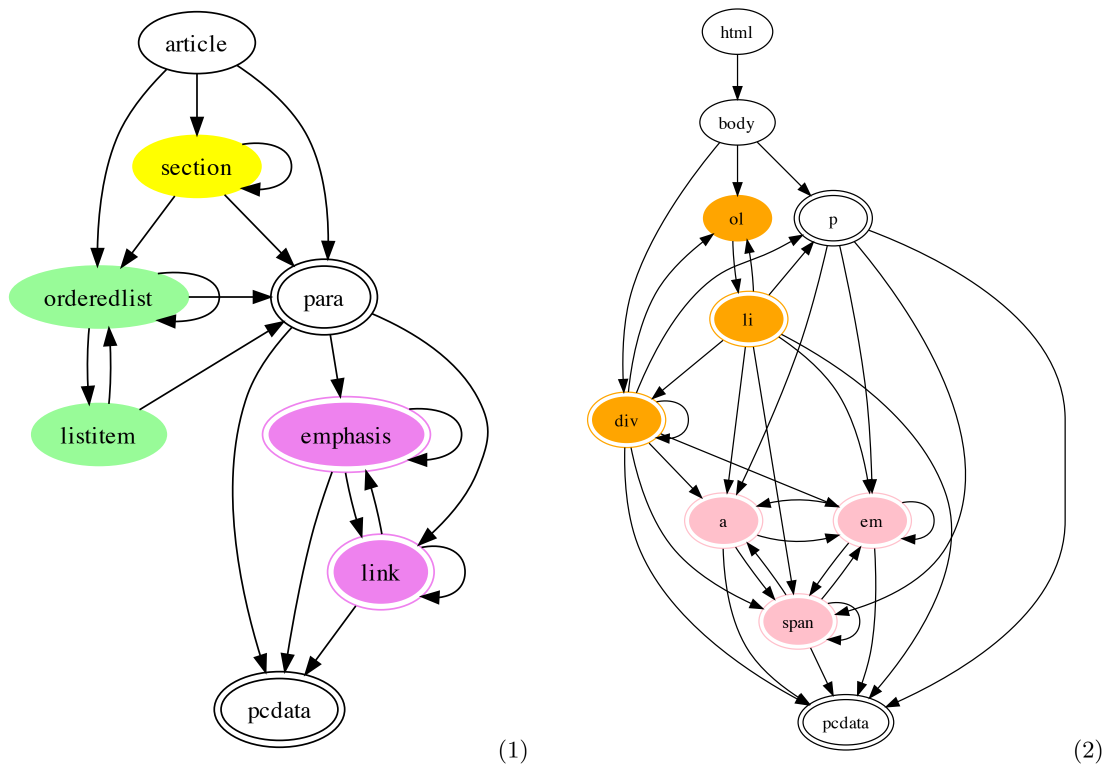


Figure 1: Parent-child constraints among DocBook Tiny (1) element types and among XHTML Tiny (2) element types. Double–circled nodes denote types whose elements may appear as leaves in document trees.

Programming languages have been designed and implemented to accommodate XML content models as native data types [1]. Many of these languages are high–level languages with expressive type systems to help detect type errors at compile–time. XML query and transformation languages such as XPath, XSLT, and XQuery have long been developed, standardized, and put into use; but these languages do not have a notion of document types for the input and output documents. Our method models XML document transformation before the transformation itself is programmed, and is not specific to particular programming languages.

Researchers in the database community have used graph homomorphism for matching data schema and for matching navigation paths [3]. These works focus on the similarity among graphs; ours is on expressing transformation using graphs. Some works analyze web applications in order to verify if they produce valid HTML/XML documents [9]. These works seem to concentrate on the target document types, but not the source document types.

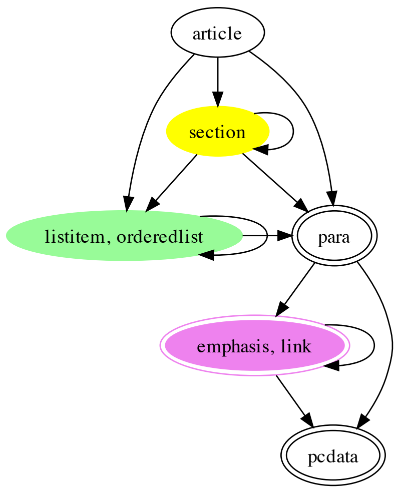
3. GRAPH HOMOMORPHISM

This section provides a brief introduction to graph homomorphism and some basic graph theory [8]. We use the notations that $g, h, \ldots$ are graphs, V_g is the set of nodes in graph g, and E_g is the set of edges in graph g. The graphs are directed and, if not noted otherwise, without multiple edges. That is, $E_g \subseteq V_g \times V_g$ is a relation on V_g. We write $u \rightarrow v$ to denote an edge $(u, v) \in E_g$. An edge $u \rightarrow v$ is a loop if $u = v$. A function $f : V_g \rightarrow V_h$ is a *graph homomorphism* from graph g to graph h if $f(u) \rightarrow f(v)$ is an edge in E_h for all edges $u \rightarrow v$ in E_g. That is, a graph homomorphism is a node–to–node function that preserves edge connectivity. A graph is strongly connected if there is a path from each node to each other node in the graph. The *strongly connected components* (*SCCs*) of a graph are its maximal strongly connected subgraphs.

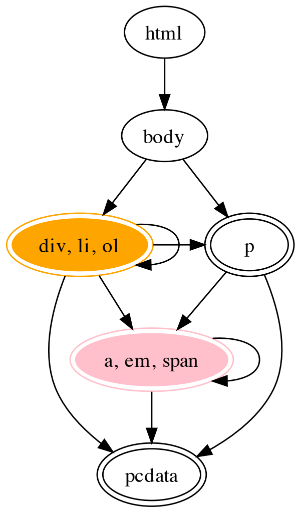
We extend the notion of graph homomorphism by the following. A function $f : V_g \rightarrow V_h$ is an *extended graph homomorphism* from g to h if, for all edges $u \rightarrow v$ in E_g, there is a *path* $f(u) \rightsquigarrow f(v)$ connected by edges in E_h. That is, an extended graph homomorphism preserves path connectivity. One can see that a function $f : V_g \rightarrow V_h$ is an extended graph homomorphism from g to h if and only if f is a graph homomorphism from g to the transitive closure of h.

In the context of XML document transformation, often we are mapping documents from a particular document type (the *source*) to those of another document type (the *target*). In this paper, we focus on the parent–child constraints among element types as imposed by their content models, using graph homomorphism as a formalism to guide and specify document transformations. It leads to a general method for mapping elements types from the source document type to those in the target document as the following.

- Produce the source graph g and target graph h respectively from the source and target document types. Nodes in the graphs are element types. Edges are parent–child constraints.
- Decompose g and h, respectively, into their SCCs. Find a graph homomorphism from the condensation of g to the condensation of h. Note that,
  - Each SCC node in the condensation of g must be



(1)



(2)

Figure 2: The condensations, and the related strongly connected components, of the two graphs.

mapped to a SCC node in the condensation of h (to preserve cycles inside the source SCC);

– One–to–one function is preferred when mapping SCC nodes in the condensation of g to those in h (to preserve structural information).

- For each pair of source and target SCCs, find a graph homomorphism for nodes in the source SCC to those in the target SCC.

- A loop in the source SCC must be mapped to a loop in the target SCC.

Figure 2 shows the condensations of the two graphs in Figure 1. The SCCs in DocBook Tiny (1) are colored in *yellow* = {`section`}, *green* = {`listitem`, `orderedlist`}, and *violet* = {`emphasis`, `link`}. Those in the XHTML Tiny (2) are colored in *orange* = {`div`, `li`, `ol`} and *pink* = {`a`, `em`, `span`}. For a graph homomorphism from the condensation of DocBook Tiny to the condensations of XHTML Tiny, we use the following mapping f:

$$\begin{aligned}
f(\texttt{article}) &= \texttt{body}\\
f(yellow) &= orange\\
f(green) &= orange\\
f(\texttt{para}) &= \texttt{p}\\
f(violet) &= pink\\
f(\texttt{pcdata}) &= \texttt{pcdata}
\end{aligned}$$

It can be verified this is indeed a graph homomorphism. For the mapping from the yellow SCC to the orange SCC, one has no choice but to map `section` to `div` as node `section` forms a loop and `div` is the only looping node in the target SCC. The same goes for the mapping from `orderedlist` to `div`. From the above, `listitem` can only be mapped to `div` too. For the mapping from the violet SCC to the pink SCC, we map `emphasis` to `em`, and map `link` to `span` instead of `a` because both `link` and `span` are looping nodes but `a` is not.

Also, since all documents we are mapping from the DocBook Tiny document type to the XHTML Tiny document type are tree–shaped, we must make sure that whatever can appear as leaf elements in the input, will only be mapped to leaf elements in the output, and that the root element is mapped to the root element. That is, we will be transforming an XML document tree to a complete XML document tree, not just to some fragments of a tree.

For leaf nodes in the graph for DocBook Tiny (that is, `para`, `emphasis`, `link`, and `pcdata`), indeed they are mapped to leaf nodes in the graph for XHTML Tiny (that is, `p`, `em`, `span`, and `pcdata`). The root node `article` is mapped to `body`. But the root node in the target graph is `html`, not `body`. This is easy to fix in an extended graph homomorphism, as there is a path from `html` to `body`. That is, we map the node `article` to the subgraph consisting of the edge `html` $\rightarrow$ `body`.

4. INDUCTIVE TRANSFORMATION

The solution we arrive at in the previous section is not satisfactory as much information is lost. The element types `orderedlist` and `listitem` are mapped to the element type `div`. We also do not get to use element `a` in the target graph to express element `link` in the source. In an extended graph homomorphism, an edge is mapped to a path (as illustrated by mapping the edge `article` $\rightarrow$ `section` to the path `html` $\rightarrow$ `body` $\rightarrow$ `div`). In this section, we aim to develop refined mappings for element types in which paths in the target graph are used to connect the parent–and–child pairs of nodes mapped from the source graph.

In the following, we show we can still map `orderedlist` to `ol`, and `listitem` to `li`. Node `orderedlist` has children `listitem`, `orderedlist`, and `para` in the source graph. We map them respectively to `li`, `ol`, and `p` in the target graph. But in the target graph node `ol` only has `li` as its child. What do we do with `ol` and `p`? A solution is to connect node `ol` to `ol` by a path `ol` $\rightarrow$ `li` $\rightarrow$ `ol`, and to connect node `ol` to `p` by a path `ol` $\rightarrow$ `li` $\rightarrow$ `p`. Note that by insisting nodes in a SCC in the source graph are mapped to nodes in a SCC in the target, for any edge $u \rightarrow v$ in the source SCC, we are ensured there is a path $u \rightsquigarrow v$ in the target SCC.

We use the following notation to describe such a mapping:

```
orderedlist{@li, @ol, @p} = ol{@li, li/@ol, li/@p}
```

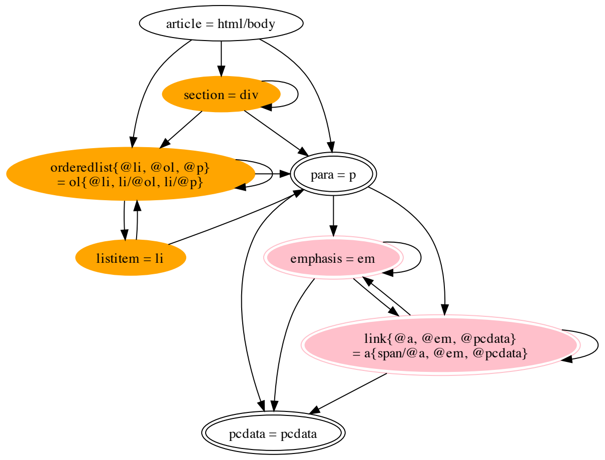


Figure 3: A structure–conforming mapping from DocBook Tiny to XHTML Tiny.

In the above, *@element* on the left hand side of the equation is a pattern that matches a node labeled with *element* (in the target); on the right hand side, it is the matched node. The pattern *element*{...} on the left hand side represents a mapping for the node labeled *element* (in the source) and for its transformed children (matched by using patterns inside the bracket). The expression *element*{...} on the right hand side represents the result in the target, which is a node labeled *element*. The node's children are inside the bracket. The notation p/q expresses a node p with a single child q. That is, p/q is a subgraph connecting two nodes p and q. This notation can be easily extended to express a subgraph consisting of a longer path, such as $u_1/u_2/\ldots/u_n$.

Figure 3 shows a mapping from DocBook Tiny element types to XHTML Tiny, based on an extended graph homomorphism. This mapping is more refined than the one given at the end of Section 3.

This mapping describes an inductive transformation on XML document trees. It leads naturally to a bottom–up transformation of DocBook Tiny documents: It maps leaf elements in DocBook Tiny to leaf elements in XHTML Tiny, and it specifies how to convert any (non–leaf) DocBook Tiny element to an XHTML Tiny element depending on the type of the element and its converted child elements. It is also a typed specification; the mappings are expressed in terms of element types (and their expressions) in the source and target document types. Further, the specification can be checked to see if it will always transform documents of the source type to documents of the target type.

The specifications are checked as the following. For each node u in the source graph, one first looks into the mapping equation

$$v_i\{\ \ldots\ \} = y_i\{\ \ldots\ \}$$

that is associated to each node v_i to which $u \to v_i$ is an edge. Nodes $y_1, y_2, \ldots, y_n$ are in the target graph, and they are u's children when u is being transformed. Then, one checks if each of the nodes $y_i, 1 \le i \le n$, appears as a pattern $@y_i$ in *pattern* in the mapping equation associated to node u:

$$u\{\ \mathit{pattern}\ \} = x\{\ \mathit{expression}\ \}$$

If so, the pattern–matching is exhaustive, and one proceeds to check if each of the corresponding expressions in *expression* leads to an edge or a path from x to y_i in the target graph. In cases where the mapping equation associated to node u is just $u = x$, one simply checks if $x \to y_i$ is an edge in the target graph.

This method of describing an inductive transformation based on an extended graph transformation can be further applied to cases where a node in the source graph is mapped to multiple nodes in the target graph. This is often necessary when one needs to break a SCC in the source graph into several subgraphs which are then mapped to different SCCs in the target graphs. In these scenarios, a node in the source SCC, depending on its already transformed child nodes, can be mapped to nodes in different SCC in the target graph.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a method for transforming XML documents that takes into account both the source and target document types. We shall end this paper by mentioning that our method can be further generalized. Until now we use only paths in the target graphs as the results from inductive mappings. Actually we can use any subgraph in the target graph as an inductive outcome. That is, from the already transformed child elements, we can assemble any XML document fragment as the inductive result, as long as it is a subgraph in the target graph.

6. REFERENCES

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML–centric general-purpose language. In *Int'l Conf. on Functional Programming*, pp. 51–63, 2003.

[2] T.–R. Chuang and J.–L. Lin. On modular transformation of structural content. In *ACM Symp. on Document Engineering*, pp. 201–210, 2004.

[3] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *Proc. of the VLDB Endowment*, 3:1161–1172, Sept. 2010.

[4] E. Kuikka, P. Leinonen, and M. Penttonen. Towards automating of document structure transformations. In *ACM Symp. on Document Engineering*, pp. 103–110, 2002.

[5] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and bared wire. In *Functional Programming Languages and Computer Architecture*, pp. 124–144, Aug. 1991.

[6] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing*, pp. 153–169, 1996.

[7] E. Pietriga, J. Vion–Dury, and V. Quint. VXT: A visual approach to XML transformations. In *ACM Symp. on Document Engineering*, pp. 1–10, 2001.

[8] A. Shapira and N. Alon. Homomorphisms in graph property testing — a survey. *Electronic Colloquium on Computational Complexity*, 12(085), 2005.

[9] R. Stone. Validation of dynamic web pages generated by an embedded scripting language. *Software: Practice & Experience*, 35(13):1259–1274, 2005.

[10] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Int'l Conference on Functional Programming*, pp. 148–159, Sept. 1999.