TREX: DTD-Conforming XML to XML Transformations

Aoying Zhou Qing Wang Zhimao Guo Xueqing Gong Shihui Zheng
Hongwei Wu Jianchang Xiao Kun Yue
Fudan University, China
{ayzhou,qingwang,zmguo,xqgong,shzheng0,hwwu,jcxiao,kuny}@fudan.edu.cn

Wenfei Fan
Bell Laboratories, USA
wenfei@research.bell-labs.com

1. Overview

With the popularity of XML, it is increasingly common to find data in the XML format. This highlights an important question: given an XML document S and a DTD D, how to extract data from S and construct another XML document T such that T conforms to the fixed D? Let us refer to this as *DTD-conforming XML to XML transformation*. The need for this is evident in, *e.g.,* data exchange: enterprises exchange their XML documents with respect to a certain predefined DTD. Although a number of XML query languages (*e.g.,* XQuery, XSLT) are currently being used to transform XML data, they cannot guarantee DTD conformance. Type inference and (static) checking for XML transformations are too expensive [1] to be used in practice; worse, they provide no guidance for how to specify a DTD-conforming XML to XML transformation.

In response to the need we have developed *TREX* (TRansformation Engine for XML), a middleware system for DTD-conforming XML to XML transformations. TREX is based on the novel notion of *XTG* (XML Transformation Grammar), which extends a DTD by incorporating semantic rules defined with XML queries (expressed in Quilt [5]). This allows one to specify how to extract relevant data from a source XML document via the queries, and to construct a target XML document directed by the embedded DTD. TREX supports XTGs using Kweelt [6] as the underlying engine for XML queries (the reason for choosing Quilt rather than XQuery/XSL is that we could access the source code of Kweelt to incorporate our optimization algorithms). Given an XTG and a source document, it provides two evaluation modes: (1) in the *batch* mode, it generates a complete XML document, which is guaranteed to conform to the DTD embedded in the XTG; (2) in the *lazy* mode, it constructs a partial XML (DOM) tree, interacts with users, and expands the tree upon users' requests. As observed by [3], the lazy mode allows users to generate partial XML documents based on their interest; it also reduces resource utilization and presents more opportunities for optimization. TREX evaluates XTGs efficiently by implementing several optimization techniques: query composition, XPath simplification and graph reduction (a technique borrowed from functional programming for identifying repeated queries and reusing their results).

To our knowledge, TREX is the first attempt to deal with DTD-conforming XML transformations. Close to our work is [2], a DTD-directed publishing system for relational data. But XML transformations present new challenges, and thus demand new solutions, both at the conceptual level (XTG) and at the implementation level (TREX); these are beyond the issues investigated by [2] in the relational context.

With a prototype of TREX, the demonstration is to show that XTGs present a novel approach to handling DTD-conforming XML transformations, and that the optimization techniques of TREX are effective in practice. Our ultimate goal is to provide a systematic method and a practical system to support DTD-conforming XML transformations.

2. XTG: XML Transformation Grammar

We first briefly describe XTGs, the backbone of TREX.

Given a target DTD D, an XTG specifies a transformation as follows: (1) For each element type A in D, it defines a variable $\$A$; intuitively, each A element in an XML tree is to have a variable $\$A$, which contains a single XML element as its value. (2) For each element type definition (production) $A \rightarrow \alpha$ in D, where α is a regular expression, it specifies a set of semantic rules such that for each element type B in α, there is a rule for computing the values of $\$B$ via Quilt queries; the query is treated as a function that may take $\$A$ as a parameter. (3) For each attribute $@l$ of A, denoted by $A \Rightarrow @l$, the XTG also defines a variable $\$l$ and a semantic rule as above, treating $\$A$ as a parameter. Given a source XML document, the XTG is evaluated top-down: starting at the root element type of D, evaluate semantic rules associated with each element type/attribute encountered, and create nodes following the DTD to construct the target XML tree. The values of the variable $\$A$ are used to control the construction.

As an example, consider DBLP XML data, which collects records about papers (`inproceedings`). For each paper, it provides information about its authors, title, year, url, citation (`cite`), key, etc. Suppose that one wants to construct a target XML document T that contains all the papers co-

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

```
<!ELEMENT dblp        (paper*)>
<!ELEMENT paper       (title, url | nourl, citation)>
<!ATTLIST paper        key        CDATA #REQUIRED
                       year       CDATA #IMPLIED>
<!ELEMENT citation    (paper*)>
/* #PCDATA is omitted here.  */
```

Figure 1: Example of a target DTD

authored by Vardi and published in 2002, along with all the papers that are cited directly or indirectly by these papers and published in or after 1995. Furthermore, it is required that T conforms to the DTD D_0 given in Fig. 1. Observe that D_0 is *recursive:* `paper` is defined in terms of itself. That is, below each paper, the papers cited by it must be presented; this leads to an XML tree of an unbounded depth. The DTD is also *nondeterministic:* a `paper` may have either `url` or `nourl`, but not both; moreover, it may have an optional attribute `@year`. To do this transformation one might want to use an XQuery or XSLT query to generate an XML tree and then check whether the tree conforms to D_0. The problem is that if the query does not type check, then one has to start all over again. In other words, one can get a transformation that type checks only after repeated failures.

An XTG σ specifying the transformation is shown in Fig. 3. When being evaluated on a source XML document S containing DBLP records, σ produces a target XML tree T of the form depicted in Fig. 2 as follows.

(1) It first creates the root element, `dblp`, and then triggers the rules associated with the production `dblp` $\rightarrow$ `paper*`. Observe that the production contains a Kleene star; thus there is no bound on the number of the `paper` children of the root. These children are determined by the evaluation of the Quilt query Q_1 over S, which returns all the `inproceedings` elements (representing papers in S) that are co-authored by Vardi and published in 2002. For each p of these elements, a `paper` element is created as a child of the root, carrying p as the value of its variable $\$paper$. The operator "$\leftarrow$" in the rule denotes the iteration for generating the `paper` children, corresponding to the Kleene star in the DTD production.

(2) At each `paper` element p, T is expanded by generating the children for p. In contrast to the last case, the production for `paper` tells us that p has exactly three children: one `title` child, one `citation` child and either a `url` child or a `nourl` child. The query Q_2 extracts `title` from $\$paper$. The choice of `url` or `nourl` is made by a condition query Q_3 on the data in $\$paper$: p has a `url` child if and only if the value of the variable $\$url$ is not the special value `#NULL#`; similarly for `nourl`. For `citation`, Q_4 collects all the papers cited by p, which are put in a single element c_0. A `citation` child is then created, carrying c_0 as the value of $\$citation$. Note that Q_4 uses $\$paper$ as a parameter.

The attributes of p are generated similarly, by extracting the relative text data. The optional attribute `@year` is created only if the information exists in $\$paper$, as specified by the condition query Q_6. In contrast, the `@key` attribute is treated differently by Q_5 as it is *required* in the DTD D_0.

(3) At each `citation` element c, the target tree T is expanded by generating `paper` children for c. Specifically, for each (`inproceedings`) element c_0 in $\$citation$ of c, it creates a `paper` child carrying c_0 as the value of its variable $\$paper$. Each `paper` element is then processed as described in (2).

(4) For a `title` element t, the query Q_8 extracts the text data from $\$title$ as the PCDATA of t; similarly for `url` and `nourl`. If Q_8 returns multiple string values, then their concatenation (with a default ordering) is treated as PCDATA.

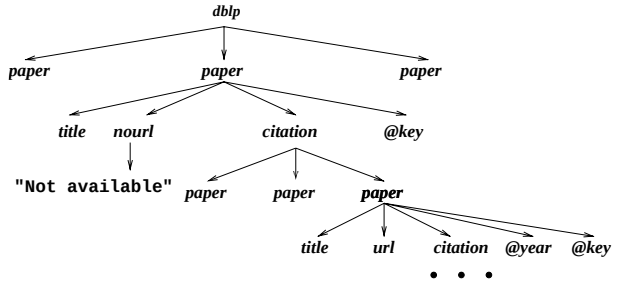

Figure 2: An XML tree conforming to D_0

dblp $\rightarrow$ **paper***
Q_1: $paper $\leftarrow$ LET $doc:=document("dblp.xml")
FOR $p IN DISTINCT ($doc//inproceedings)
WHERE CONTAINS($p/author,"Vardi")
AND $p/year = 2002
RETURN $p

paper $\rightarrow$ **title**, **url** + **nourl**, **citation**
Q_2: $title = RETURN $paper/title
Q_3: ($url, $nourl) = RETURN
IF (EXISTS($paper/url))
THEN ($paper/url, #NULL#)
ELSE (#NULL#,
<nourl>"Not available"</nourl>)
Q_4: $citation = <citation>
LET $doc:=document("dblp.xml")
FOR $cite IN ($paper/cite)
FOR $cpaper IN ($doc//inproceedings)
WHERE $cpaper/year .>=. 1995
AND $cpaper/@key = $cite/text()
RETURN $cpaper
</citation>

paper $\Rightarrow$ **@key**
Q_5: $key = RETURN $paper/@key
paper $\Rightarrow$ **@year**?
Q_6: $year =LET $yr := $paper/year
RETURN IF (EXISTS($yr))
THEN $yr/text()
ELSE #NULL#

citation $\rightarrow$ **paper***
Q_7: $paper $\leftarrow$ FOR $p IN $citation
RETURN $p

A $\rightarrow$ PCDATA /* A is one of `title`, `url`, `nourl` */
Q_8: $PCDATA = RETURN $A/text()

Figure 3: Example of an XTG

Steps (2) and (3) are repeated until the target tree T cannot be further expanded, *i.e.,* when all the `papers` at the leaves of T no longer cite papers published in or after 1995. At this point the evaluation of the XTG is completed.

XTG has several salient features. First, when the evaluation of an XTG terminates, the target XML tree generated is guaranteed to conform to its embedded DTD. Second, it adopts a *data-driven* semantics: the decisions on the choice of a nondeterministic production and on the expansion of an XML tree in the recursive case are made based on the source data. Third, it is fairly easy to use XTGs to specify XML transformations. Comparing to the grammar-based formalism of [2], XTGs are more involved: XTG variables carry XML (trees) elements rather than simple tuples.

3. TREX: A middleware system

We next give an overview of TREX, a middleware system supporting XTG evaluation. The system is built on top of Kweelt [6], which is a query engine for the Quilt XML query

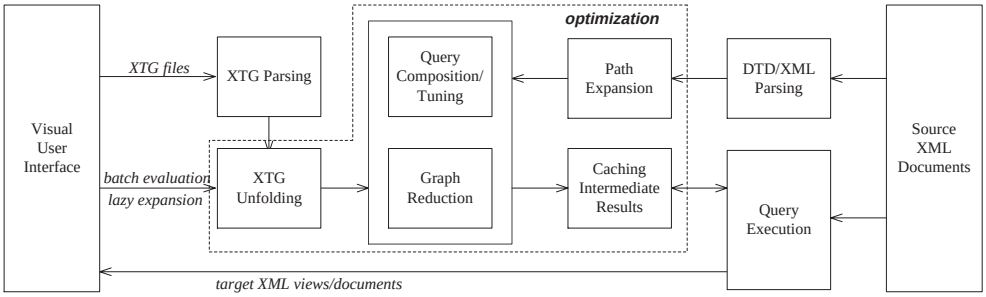

Figure 4: System architecture

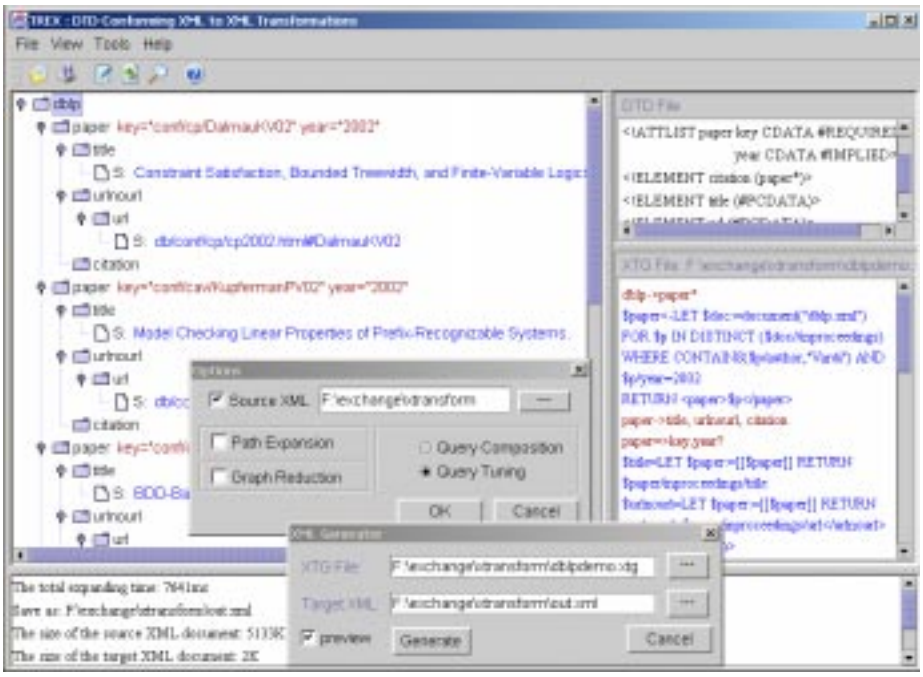

Figure 5: Visual user interface

language. TREX is currently implemented in Java.

As depicted in Fig. 4, TREX takes an XTG σ and a source XML document as inputs and generates a target XML document T that conforms to the DTD embedded in σ. It consists of a parsing phase, an optimization phase and a generation phase. In the *parsing phase*, an XTG is loaded and parsed into a graph representation, called an *XTG graph*, which is a DTD graph with nodes labeled with XML queries. The graph is cyclic if the DTD is recursive. The source document is also loaded and parsed in this phase. In the *optimization phase*, the XTG graph is first unfolded to a certain depth, which yields a partial XTG tree (the sub-graph down to the unfolding depth); then, a query plan is generated for the XTG tree by applying several optimization techniques. In the *generation phase* the query plan is submitted to the underlying Kweelt engine; using the query results TREX expands the target document T. The second and third phases are repeated until the construction of T is completed.

The user interface of the system is shown in Fig. 5. In a window it provides the DOM tree of the partial target document generated at each stage. A user can click on any node in the DOM tree and choose between two evaluation modes to generate its subtree. In the *batch* mode, the entire subtree is constructed. In the *lazy* mode, the subtree is expanded for one level, *i.e.*, only the children of the node are created; the user can then decide whether further expansion is needed. Thus, TREX is quite flexible: one can use it to produce just the interested parts of a document instead of the entire document, along the same lines as [3].

Below we focus on the optimization techniques.

Query composition and tuning. TREX has implemented several techniques for optimizing middleware XML queries [4, 2]. One is *query composition:* to reduce traffic to the underlying Kweelt engine, TREX extracts connected queries from a partial XTG tree, composes them into a single query and submits it to Kweelt. This is commonly used in batch evaluation. Another is *caching:* intermediate query results are cached and reused at later stages of the evaluation. As opposed to [2], to improve the response time we keep the intermediate results as DOM trees in the main memory; the buffer is maintained by a swapping algorithm. The lazy mode typically involves *query tuning:* after a query at a node is evaluated, we substitute its result for the parameters in the queries associated with the children of the node; furthermore, repeated queries are identified and their cached results are used to rewrite the subsequent queries.

Graph reduction. To reduce unnecessary repeated computations, TREX treats a parameterized query as a function and caches its evaluation results. If the function is invoked again with the same parameter, TREX reuses the cached

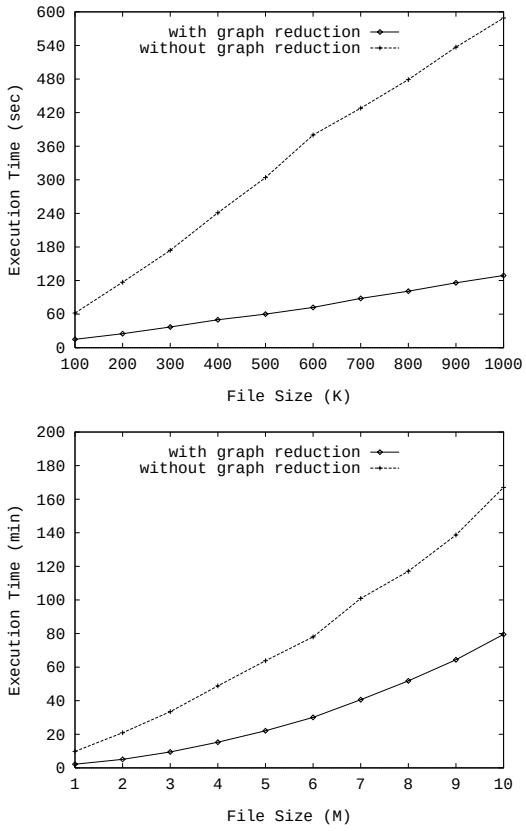

Figure 6: Benefits of graph reduction

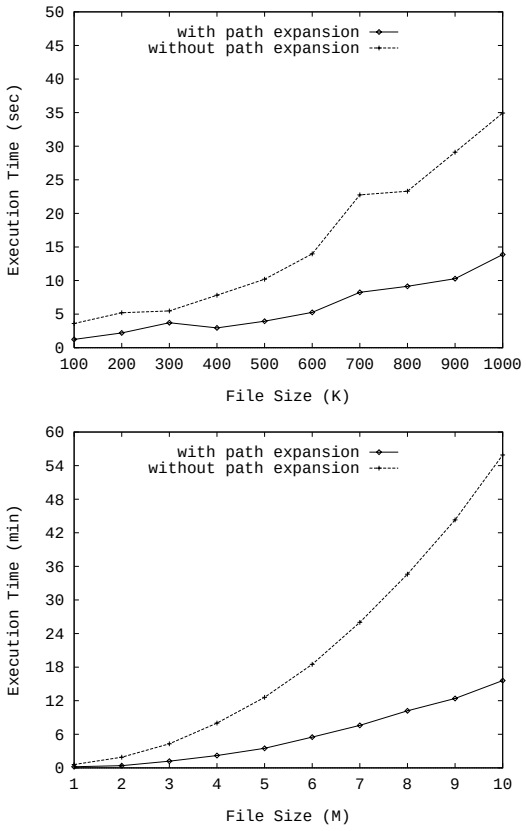

Figure 7: Benefits of path expansion

result instead of re-evaluating it. The analysis is conducted along the same lines as graph reduction extensively studied for functional programming, by extending the XTG graph with an auxiliary indexing structure (for parameters). This strategy is effective when deep XML trees are constructed, especially when recursive DTDs are involved.

Path expansion. Quilt queries heavily use XPath expressions, which are a major cost of XTG evaluation. To reduce the cost, TREX parses the source document S, extracts concrete (simple) paths from S, and rewrites XPath expressions in XTG queries by substituting concrete paths for expansive traversals such as "//" (descendant) and "*" (child). This is done at compile time for all the queries in the XTG. When the DTD of the source document is available, TREX uses the DTD for certain expansions ("*" and even "//" for non-recursive DTDs) without looking into the source document.

4. Performance

We next present some preliminary experimental results, which demonstrate that our optimization techniques – graph reduction and path expansion – are effective. The benefits of query composition are not presented here as there have been extensive experiments conducted for it [4, 2].

Our experiments were conducted on a 1.8GHz Pentium 4 machine with 40G of hard disk and 512MB of main memory running Windows 2000. We adopted DBLP XML records as source XML documents. We used an XTG similar to (yet more complicated than) the one described in Section 2; the evaluations were conducted in the batch mode.

Fig. 6 depicts the impact of graph reduction as a function of the source document size. We evaluated the XTG both with and without graph reduction. The execution time measures the time from loading the XTG file until the target XML document is generated. The experimental results indicate that graph reduction can speed up the evaluation by a factor of up to 9.6. The benefit of the technique is more evident when the size of the source document increases.

The next experiment demonstrates the benefit of path expansion: the XTG was evaluated both with and without path expansion. The results, shown in Fig. 7, tell us that path expansion can reduce the traversing time and improve the performance by a factor of up to 4.5. Better still, as the size of the source document increases, the benefit of path expansion becomes more significant.

We are currently exploring other optimization techniques for TREX, such as indexing, partitioning large XTG graphs, and and more sophisticated caching strategies.

Acknowledgment: Aoying Zhou is supported in part by NSFC 60228006. Wenfei Fan is supported in part by NSF Career Award IIS-0093168 and NSFC 60228006.

5. References

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *PODS*, 2001.

[2] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *VLDB*, 2002.

[3] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, and P. Shenoy. Optimizing view queries in rolex to support navigable result trees. In *VLDB*, 2002.

[4] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD*, 2001.

[5] J. Robie, D. Chamberlin, and D. Florescu. Quilt: an XML query language. `http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html`.

[6] SourceForge. Kweelt. `http://kweelt.sourceforge.net`.