A Uniform System for Publishing and Maintaining XML Data

Byron Choi
University of Pennsylvania
kkchoi@gradient.cis.upenn.edu

Wenfei Fan[*]
University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

Xibei Jia
University of Edinburgh
x.jia@sms.ed.ac.uk

Arek Kasprzyk
European Bioinformatics Institute
arek@ebi.ac.uk

1 Introduction

XML has become the prime standard for data exchange on the Web. To exchange data currently residing in databases, one needs to *publish it in* XML, *i.e.,* to extract data from the database and transform the data into an XML format. In practice, data publishing is often done with a predefined "schema". A community agrees on a certain schema, and subsequently all members of the community exchange their data *w.r.t.* the predefined schema, by ensuring their published (*target*) XML data to conform to the fixed schema. This is called *schema-directed* XML *publishing*. The need for this is particularly evident in biological data exchange and services. However, it is nontrivial to ensure that the target XML data conforms to a given schema. The difficulty is introduced by, among others, recursion in a target schema, which is common in, *e.g.,* biological ontologies [7].

With XML publishing also comes the increasing need for *maintaining* target XML data. The underlying source data often changes and evolves, and the source updates should be reflected in its XML target accurately and efficiently. A naive approach would be to recompute the XML target from scratch in response to source data changes. This is not very realistic in many applications where XML publishing involves voluminous data and may take hours to complete. This suggests that one needs to deal with updates incrementally: propagate the updates from the source data to its XML target with minimal recomputation. While this is reminiscent of traditional database view maintenance, incremental updates are more challenging for hierarchical and possibly recursive XML views constrained by a predefined schema.

In response to the need we proposed a new approach for schema-directed publishing of relational data in XML, based on the novel notion of attribute transformation grammars (ATGs [3]). ATGs provide guidance on how to define views conforming to *target* schemas (DTDs) and better still, they automatically ensure schema conformance. We also developed an incremental algorithm for maintaining XML views produced by ATGs [4], based on new incremental computation techniques that capitalize on the hierarchical structure of XML data and unique features of ATGs.

Recently we have implemented a middleware system that supports both schema-directed XML publishing based on ATGs, and incremental updates of XML views created by ATGs. We have also been deploying and evaluating the system at the European Bioinformatics Institute (EBI). Our experimental results are promising: our system is capable of efficiently publishing real-life biological data (in the order of GigaBytes) and guaranteeing the XML views to conform to predefined (recursive) DTDs; moreover, our incremental update algorithm outperforms the recomputation approach by two orders of magnitude. The system will possibly be adopted by EBI in the near future. To the best of our knowledge, it is the first practical system that supports schema-directed XML publishing and incremental updates.

Taking real-life data from Gene Ontology [7] (GO), this paper demonstrates how this system can efficiently publish the GO data in XML *w.r.t.* a predefined recursive DTD, and how it incrementally updates the target XML data in response to changes to the underlying GO database.

Related work. Although a number of XML publishing systems have been developed (*e.g.,* [5, 8, 6, 9, 10]), none of these systems takes schema-conformance into account. Type-checking approaches to DTD conformance are impractical since type checking of transformations, even for simple DTDs, is computationally intractable for extremely restricted views and undecidable for realistic views [1]. Worse still, type-checking does not provide any guidance on how to repair an XML view that does not typecheck.

The notion of ATGs was proposed in [3] and the incremental update algorithm was developed in [4]. However, the work in this paper is the first effort to fully implement ATGs and incremental updates, combine them in a uniform system, and to verify the effectiveness of the techniques in real-life applications. Furthermore, in our implemen-

[*] Supported in part by NSF Career Award IIS-0093168, NSFC 60228006 and EPSRC GR/S63205/01.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

Source relational schema R:
```
  primary_terms(go_id, name, updated)
  terms(go_id, name, updated)
  ancestors(parent_id, child_id)
  main(protein_id, name, source)
  protein2go(protein_id, go_id)
```
Target DTD D_0:
```
 <!ELEMENT db       (term*)>
 <!ELEMENT term     (children, id, tname,
                     updated, proteins)>
 <!ELEMENT children (term*)>
 <!ELEMENT proteins (protein*)>
 <!ELEMENT protein  (pname, pid, source)>
/* #PCDATA is omitted here.  */
```

Figure 1: Example of a source schema and a target DTD

tation we developed new optimization ideas that were not explored by [3, 4]. Another extension of ATG was proposed in [2] for integration, which is not part of this demo system.

2 Schema-Directed Publishing

The problem of *schema-directed publishing* can be stated as follows: given a DTD D, to define a view σ for relational databases I such that $\sigma(I)$ is an XML document conforming to D. We conduct schema-directed publishing by means of Attribute Transformation Grammars (ATGs [3]).

Given an arbitrary target DTD D, an ATG defines a view 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 relational tuple 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 SQL queries; the query is treated as a function that may take $\$A$ as a parameter. Given a relational database I, the ATG is evaluated top-down: starting at the root element type of D, evaluate semantic rules associated with each element type 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 publishing (simplified) GO [7] data in XML. The GO data is stored in a relational database, which, as show in Fig. 1, consists of three relations for GO terms: `primary_term` stores the GO `id`, `name` and the date of the last update for each primary term; similarly for other `terms`; as a term may be composed of other terms, the composition hierarchy is given by the `ancestors` relation (keys are underlined). The database has also two relations for proteins: `main` specifies the `id`, `name` and `source` of each protein; and `protein2go` tells how `terms` and `proteins` are related.

Now one wants to construct a target XML document T that contains all the primary terms immediately under the root, along with their composition hierarchy and the proteins they are related to. Furthermore, T is required to conform to the DTD D_0 given in Fig. 1. Observe that D_0 is *recursive:* a `term` may have other `terms` as its children; this leads to XML trees of unbounded depths.

An ATG σ_0 specifying the publishing is shown in Fig. 3.

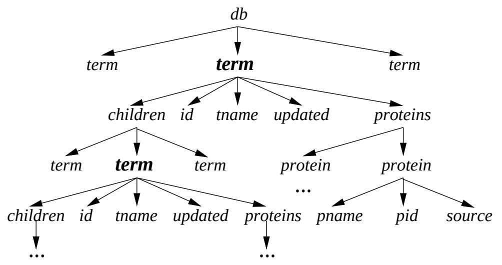


Figure 2: An XML tree conforming to D_0

When being evaluated over the GO database, σ_0 produces a target XML tree T as depicted in Fig. 2, as follows.

(1) It first creates the root element, `db`, and then triggers the rules associated with the production `db` $\rightarrow$ `term*`. Observe that the production contains a Kleene star; thus there is no bound on the number of the `term` children of the root. These children are determined by the evaluation of the SQL query Q_1 over the GO database, which returns all the `primary term` tuples. For each t of these tuples, a `term` element is created as a child of the root, carrying t as the value of its variable $\$term$. The operator "$\leftarrow$" in the rule denotes the iteration for generating the `term` children, corresponding to the Kleene star.

(2) At each `term` element t, the XML tree T is expanded by generating the children of t. In contrast to the last case, the production for `term` tells us that t has exactly five children: `children`, `id`, `name`, `updated` and `source`. The variables associated with these children are assigned values extracted from fields of the parent variable $\$term$, *e.g.,* $\$children$ inherits the value $\$term.go\_id$.

(3) At each `children` element c, the target tree T is further expanded as follows. The SQL query Q_2 finds the go_ids of all the children terms of c from the `ancestors` relation, by using $\$children.go\_id$ as a constant parameter; it then extracts tuples from the `term` relation using these go_ids. For each t' of these tuples, a `term` child of c is created carrying t' as the value of its variable, and the `term` node is in turn processed as described in (2).

(4) Similarly, at each `proteins` child p of `term` t, the SQL query Q_3 extracts all the `protein` tuples related to t from the `main` and `protein2go` relations, by using $\$proteins.go\_id$ as a constant. For each of these tuples a `protein` child of p is generated, whose children are in turn created as described in (2).

Steps (2) and (3) are repeated until the target tree T cannot be further expanded, *i.e.,* when all the `terms` at the leaves of T are no longer composed of other `terms`. At this point the evaluation of the ATG is completed.

ATG has several salient features. First, when the evaluation of an ATG terminates, the target XML tree generated is guaranteed to conform to its embedded DTD. Second, it adopts a *data-driven* semantics: the expansion of an XML tree in the recursive case are determined by the source data. Third, it is easy to use ATGs to specify schema-directed XML publishing. The DTD productions provide a guidance on how to write semantic rules to expand the tree that conforms to the DTD. There is no need to learn a new language: one can write ATGs as long as she/he knows SQL and DTD.

db → **term***
Q_1: \$term ← **select** go_id, name, updated **from** primary_terms

term → **children**, **id**, **tname**, **updated**, **proteins**
\$children = \$term.go_id, \$id = \$term.go_id,
\$tname = \$term.name, \$updated = \$term.updated,
\$proteins = \$term.go_id,

children → **term***
Q_2: \$term ← **select** t.go_id, t.name, t.updated
from ancestors a, terms t
where \$children = a.parent_id **and** a.child_id = t.go_id

proteins → **protein***
Q_3: \$protein ← **select** m.name, m.protein_id, m.source
from protein2go p, main m
where \$proteins = p.go_id **and**
p.protein_id = m.protein_id

protein → **pname**, **pid**, **source**
\$pname = \$protein.name, \$pid = \$protein.go_id,
\$source = \$protein.source

Figure 3: An example ATG σ_0

3 Incremental Updates

The *incremental update* problem for ATGs can be stated as follows: given an ATG σ, a relational database I, the XML view $T = \sigma(I)$, and changes ΔI to I, to compute XML changes ΔT to T such that $T \oplus \Delta T = \sigma(I \oplus \Delta I)$, where the operator $\oplus$ denotes the application of these updates. In contrast to recomputing the new view from scratch, incremental update of ATGs improves performance by applying only the changes ΔT to the old view T.

Our incremental algorithm [4] is based on a notion of ΔATG. A ΔATG $\Delta\sigma$ is statically derived from an ATG σ by deducing and incrementalizing SQL queries for generating edges of XML views. In response to relational changes ΔI to the source data, $\Delta\sigma$ computes XML changes ΔT via the incrementalized SQL queries, which yield a pair of relations (E^+, E^-), denoting the insertions (*buds*) and deletions (*cuts*) of the edges of the old XML view T, respectively. More specifically this is carried out in three phases: (1) a *bud-cut generation phase* that determines the impact of ΔI on *existing* parent-child (edge) relations in the old XML view T by evaluating a fixed number of incrementalized SQL queries; (2) a *bud completion phase* that iteratively computes newly inserted subtrees top-down by pushing SQL queries to the relational DBMS; and finally, (3) a *garbage collection* process that removes the deleted subtrees. It minimizes unnecessary recomputations via a novel caching strategy such that each new subtree in the XML view is computed at most once no matter how many times it occurs in the XML view, and moreover, it maximally reuses subtrees in the old XML view. The caching strategy is based on the *subtree property* of XML data and ATGs: each subtree in an XML view generated by an ATG is *uniquely determined* by the tuple-value of the variable associated with the subtree root. This allows us to efficiently identify and reuse existing subtrees via a hash table.

As an example, consider changes ΔI to the GO database I that modify the `ancestors` relations of `term`s, which can be understood as *group updates* consisting of insertions and deletions of multiple `ancestors` tuples. Referring to the XML view T of Fig. 2 generated by $\sigma_0(I)$, the corresponding XML changes ΔT are computed as follows. The bud-cut-generation phase generates a set of cuts to the (`children`, `term`) edges of T, as well as a set of buds $\mathsf{N}_{\texttt{term}}$ consisting of the newly inserted `term` tuples, in response to ΔI. It then deletes the edge of the cuts, and creates a `term` node for each tuple in $\mathsf{N}_{\texttt{term}}$, along with edges from the root `db` or `children` elements to these `terms`. Then, the bud-completion phase generates the subtrees for these new `term`s, maximumly reusing the subtrees that have been computed or are already in T by capitalizing on the subtree property. Finally, after the subtree are constructed, the garbage collection process runs in the background to remove the subtrees deleted by the cuts. Note that the physical deletion is delayed such that the removed subtrees can be reused in the bud-completion phase.

Another salient feature of the incremental algorithm is that it computes ΔT in parallel with the updating process of T with ΔT. More specifically, each iteration in the generation phase computes ΔT to a certain depth below newly added buds, and thus partial results of the new XML view can be returned to the users before the computation of ΔT is completed; this allows *lazy evaluation* that overlaps the view update process with client access.

4 System Architecture

Our middleware supports two evaluation modes: *publishing* and *incremental updates*. See Fig. 4 for its architecture.

In the ATG-based XML publishing mode, the system takes an ATG σ and a source relational database I as input and generates a target XML view $T = \sigma(I)$ that conforms to the DTD embedded in σ. Specifically, it parses σ, generates a query plan for evaluating the SQL queries embedded in σ, and pushes the SQL queries down to the underlying DBMS to extract data from the source I. The system has fully implemented the optimization techniques proposed in [3], including query composition to reduce communication cost between the middleware and the DBMS. The XML view T is stored in a subtree pool with a hash table built on top of it, leveraging the subtree property described early.

In the incremental update mode, the system accepts SQL updates and a handle (name) for an ATG σ (see also Fig. 5(a)); it then conducts the relational updates, derives $\Delta\sigma$, evaluates $\Delta\sigma$ using our incremental algorithm to compute XML changes to the corresponding XML view, and updates the subtree pool and the hash table accordingly, as described in the last section. Alternatively, the system allows the underlying DBMS to function independently and accept updates; an *update monitor* (not explored in [4]) detects source updates, and triggers our incremental algorithm to propagate the changes to XML views automatically.

The user interface of the system is shown in Fig. 5(a). In a window one can select and display a source schema and an ATG; the system also supports an interface for accepting SQL updates, and for the choice of ATG publishing or incremental updates. A graphic tool is provided to facilitate ATG design. Output XML data is browsed by popping up another window (Fig. 5(b)).

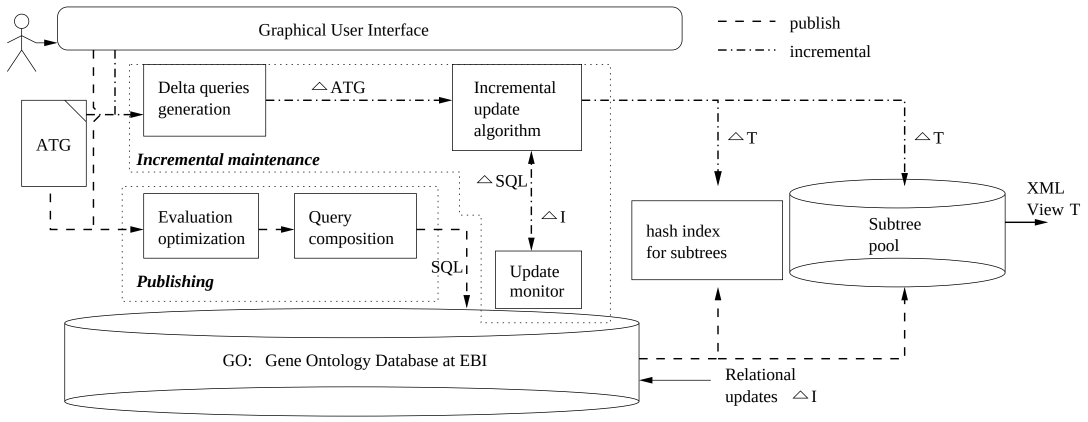
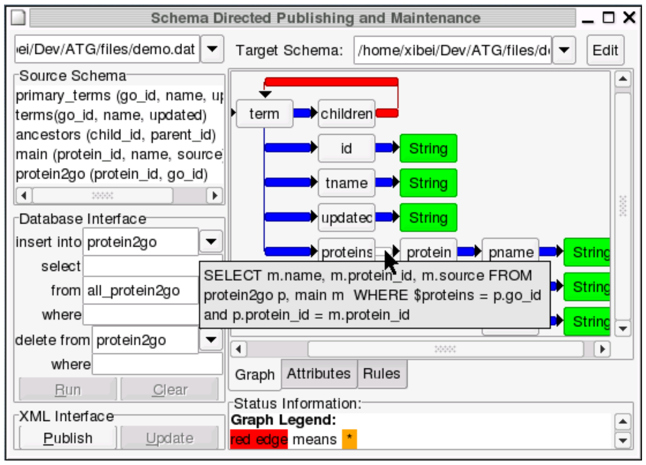


Figure 4: System architecture



ei/Dev/ATG/files/demo.dat Target Schema: /home/xibei/Dev/ATG/files/d

Source Schema
primary_terms (go_id, name, u
terms(go_id, name, updated)
ancestors (child_id, parent_id)
main (protein_id, name, source
protein2go (protein_id, go_id)

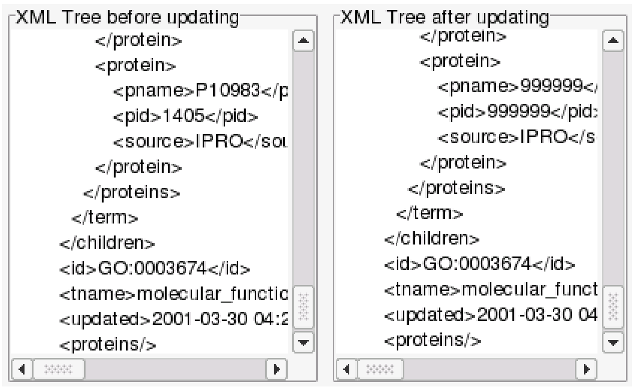
Database Interface
insert into protein2go
select
from all_protein2go
where
delete from protein2go
where

XML Interface

```
SELECT m.name, m.protein_id, m.source FROM
protein2go p, main m  WHERE $proteins = p.go_id
and p.protein_id = m.protein_id
```

Status Information:
Graph Legend:
red edge means *

(a) Visual user interface



XML Tree before updating

```
        </protein>
        <protein>
           <pname>P10983</p
           <pid>1405</pid>
           <source>IPRO</sou
        </protein>
      </proteins>
    </term>
  </children>
  <id>GO:0003674</id>
  <tname>molecular_functic
  <updated>2001-03-30 04:2
  <proteins/>
```

XML Tree after updating

```
        </protein>
        <protein>
           <pname>999999</
           <pid>999999</pid:
           <source>IPRO</s
        </protein>
      </proteins>
    </term>
  </children>
  <id>GO:0003674</id>
  <tname>molecular_funct
  <updated>2001-03-30 04
  <proteins/>
```

(b) XML data before and after updates

Figure 5: User Interface

5 Demonstration Overview

The demonstration will show the following.

Schema-directed publishing. To illustrate the main aspects of ATG-based XML publishing, we show how to publish Gene Ontology (GO) data in XML *w.r.t.* predefined recursive DTDs via ATGs. We demonstrate that our system is efficient when dealing with the real-life data.

Incremental updates. To verify the effectiveness of our incremental algorithm, we show how source updates can be efficiently propagated to XML views created by ATGs in contrast to the recomputation approach: the former takes seconds to evaluate while the latter takes minutes.

Aids to ATG specification. We provide a graphic user interface to facilitate ATG design (Fig. 5(a)). An ATG is depicted as a graph, in which each edge represents a parent-child relation in a production in the target DTD. Clicking on the edge, a text window displays the corresponding semantic rule and allows the user to display and edit the rule.

Aids to answer analysis. We also demonstrate graphic tools for viewing the published XML data, and for illustrating update propagation from source to target by comparing XML views before and after updates (Fig. 5(b)).

References

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. In *LICS*, 2001.

[2] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, 2003.

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

[4] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

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

[6] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, 2000.

[7] EBI. Gene Ontology. `http://www.geneontology.org/`.

[8] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.

[9] Intelligent Systems Research. XML from databases: ODBC2XML. `http://www.intsysr.com/odbc2xml.htm`.

[10] Oracle. Using XML in Oracle internet applications. `http://technet.oracle.com/tech/xml/`.