Constructing Maintainable Semantic Mappings in XQuery

Gang Qian
Dept. of Computer Science and Engineering
Southeast University, Nanjing 210096, China
qiangang@seu.edu.cn

Yisheng Dong
Dept. of Computer Science and Engineering
Southeast University, Nanjing 210096, China
ysdong@seu.edu.cn

ABSTRACT

Semantic mapping is one of the important components underlying the data sharing systems. As is known, constructing and maintaining such mappings both are necessary yet extremely hard processes. While many current works focus on seeking automatic techniques to solve such problems, the mapping itself is still left as an undecorated expression, and in practice it is still inevitable for the user to directly deal with such troublesome expressions. In this paper we address such problems by proposing a flexible and maintainable mapping model, where atomic mapping and combination operators are the main components. Conceptually, to construct global mapping for the whole target schema, we first construct the atomic mappings for each single target schema element, and then combine them using the operators. We represent such combined mappings as mapping trees, which can be incrementally constructed, and can be locally maintained. Also, we outline the main issues in combining our work with the current automatic techniques, and analyze the maintainability of the mapping tree. Though our discussion is applicable to other models, this paper limits the attention to the XML model and the XQuery language.

1. INTRODUCTION

Semantic mapping is one of the important components underlying the data sharing (e.g., data integration and data exchange) systems. For example, the mappings may be used to translate the user query over the target (mediated) schema into queries over the source schemas (e.g., [8]), or translate the data resided at different sources into the target database. To enable data sharing, the user has to first construct the semantic mappings between the target and the source schemas. Also, as the application requirements or the schemas change, the user has to maintain and modify the early constructed mappings.

As is known, constructing and maintaining such mappings both are extremely labor-intensive and error-prone processes. Trying to provide automated support, much recent literature has extensively studied the techniques like *schema matching*, *mapping discovery* and *mapping adaptation*. Given a pair of schemas, the technique of schema matching focuses on discovering semantic correspondences (*matches*) between schema elements (e.g., [11, 6, 18, 5]). Taking these matches as input, the tools like `Clio` [3] then are employed to further discover and generate the candidate semantic mappings between the schemas, e.g., in the form of a *naive* SQL or XQuery expression [10, 13]. When the schemas evolve, the technology of mapping adaptation is responsible for adjusting the mappings constructed originally and keeping them as consistent as possible [19].

Despite this progress, however, it is still inevitable for the user to directly construct and maintain the mappings. In practice there are many factors that may require to modifying and maintaining the mappings. For example, mapping construction is usually a process of repeated refinement. In most case, only semantically valid and partial mappings the automated techniques can discover. To obtain the *desired* one, the user may need to further refine the discovered mappings, or completely reconstructed it in the cases beyond the intelligence of the automated techniques. A detailed motivation appears in Section 2.

As the automated techniques could not completely solve these mapping problems, we are inspired to explore other *complementary* ways to alleviate the burden on the user. Currently, schema mappings are mainly represented as (query) expressions, which are troublesome for the user to deal with. In dynamic environment like the Web, schemas and application requirements may change frequently. We believe that a maintainable mapping representation would be more suitable than the undecorated expression. Further, as large, complicated schemas become prevalent on the Web, it may be more feasible to incrementally construct the whole schema mappings, e.g., starting with simple mappings, and then gluing them to formulate the globe ones.

In terms of the above observations, in this paper we propose a flexible and maintainable mapping model, where *atomic mapping* and *combination operators* are the main components. Specifically, we limit our attention to the XML model and the mappings expressed in XQuery, though our discussion is also applicable to others. In our model, two atomic mappings (say M_1 and M_2) may be combined using the *Nest*, the *Join* or the *Merge* operator, and the resulting mapping is called *combined mapping* (say M_3). We say that the combined mapping M_3 is *maintainable*, which means that it can be combined again with others, possibly using another combination operator, and it is also possible to reset the operator of connecting M_1 and M_2, or recover M_1 and M_2 from M_3.

With our model, to construct global mapping for the whole target schema, we begin to construct the atomic mappings for each single target schema element, and then incrementally combine them using the operators. Such *flexibility* in mapping construction makes our model adapt well to complicated applications. We represent the combined mapping as a mapping tree. To maintain and modify the schema mapping, we only need to adjust the corresponding nodes of the mapping tree, while other

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland

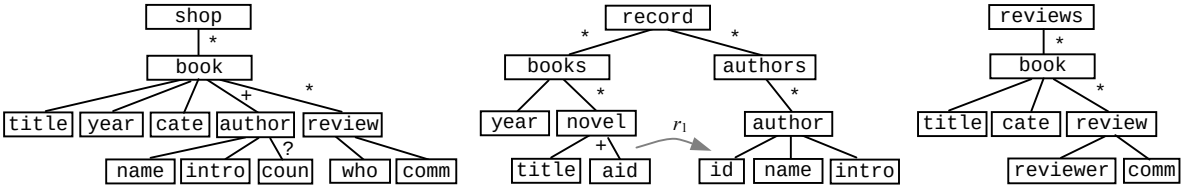


Figure 1. The target schema T (left), source schema S1 (middle) and S2 (right)

nodes and their relationships are reused. Note that our approach would not replace, but rather complement existing techniques to assist the user to manage the schema mappings. We analyze the maintainability of the mapping tree, and outline the main issues in combining our work with the current automatic techniques.

2. A MOTIVATING EXAMPLE

We start with a common example of sharing book information and illustrate the practical requirement for a flexible, maintainable mapping representation model. Suppose there is an online shop that wants to collect data from other sources. Figure 1 shows the schemas `T` of the shop and `S1` and `S2` of its two sources, which serve as our running example for discussing schema-to-schema mappings throughout the paper.

We model nested schemas as tree structures, where each tree edge denotes the structural constraint, the non-tree edge like r_1 of `S1` indicates the referential constraint, and the multiplicity label associated with the tree edge represents the cardinality constraint. In the source (of) `S1`, `books` are grouped by `year`, and then categorized by the styles such as `novel`. The source `S2` provides `reviews` of books. We suppose that the book is identified by its `title`. Note that it is uncertain that every `novel` instance of `S1` must have corresponding reviews in `S2`.

Using XQuery, we give an example of mapping expression as follows, which relates the source schemas `S1` and `S2` and the target schema `T`, and indicates the correspondences between schema elements, e.g., the `novel` of `S1` and the `book` of `T`.

```
<shop>
for $bs in doc("S1")//books, $n in $bs//novel
return <book>
    {$n/title, $bs/year}
    <cate>{"novel"}</cate>{
    for $a in doc("S1")//author
    where $n/aid=$a/id
    return <author>
        {$a/name, $a/intro}
    </author>}
    … … … … … …
    </book>
</shop>
```

Figure 2. An example mapping expression

In practice, there are many cases where the mappings have to be modified and maintained. First, constructing the mappings may well be a repeated refinement process, especially for complicated applications. For example, to construct the above mapping, the user at the beginning might have related the `book` elements of `S2` and `T`. In another case, if the referential constraint r_1 of `S1` did not hold, then the above mapping may need to be refined to define the target `author` instances by the `aid` of `novel`, while for those authors not stored in `S1`, the related target attributes like `name` may be filled with null values or Skolem functions. Second, when the application requirements or the schemas change, the mappings need to be maintained accordingly. For example, for some reason the shop may want to constraint the schema element `review` by "+". Again, the shop may want to alter to share reviews from other more economy sources.

The undecorated expression is troublesome for the user to deal with. In contrast, we propose to represent the schema mapping as a combined formulation, where *atomic mappings* and *combination operators* are the main constituents. The atomic mapping defines the local view of a single schema element. Using the combination operator, two atomic mappings can be connected, and the result is a *combined mapping*, which can be further combined with other combined or atomic mappings.

Example 2.1 For the single target elements `book` and `title`, we respectively construct the atomic mappings as follows.

$M_{\texttt{book()}}$:
```
for $n1 in doc("S1")//novel
return <book></book>
```

$M_{\texttt{title}}$:
```
for $n2 in doc("S1")//novel, $t1 in $n2/title
return $t1
```

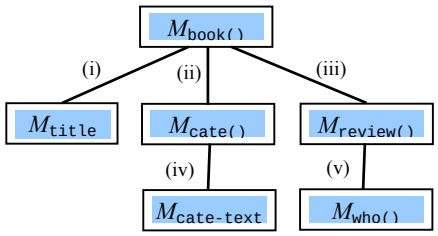
Using the *Nest* operator (see Section 3), we combine $M_{\texttt{book()}}$ with $M_{\texttt{title}}$ and obtain the mapping $M_{\texttt{book(title)}}$ as follows.

```
for $n1 in doc("S1")//novel
return <book>{
  for $n2 in doc("S1")//novel, $t1 in $n2/title
  where $n1=$n2
  return $t1}
  </book>
```

The above combined mapping represents a more significant view, where the initially separate `title` instances returned by $M_{\texttt{title}}$, now is structurally nested within the `paper` instances. Continuing to apply the operators in the same way, other instances also can be nested within the returned `paper` instances. □

A mapping should be semantically valid, i.e., conforming to the constraints contained in the target schema. Intuitively, ignoring its contexts (i.e., the associated constraints), we think of the single schema element as the simplest form of schema. Then the atomic mapping represents the semantic relationship between the source schemas and the simplest target schema. As the atomic mappings are combined, the separated schema elements are stitched up, and the ignored contexts are recovered. Thus, to construct the global mappings for the whole target schema, we begin to construct the atomic mappings for each single target schema element, and then combine them together by applying the operators.

The combined mapping possesses a tree structure, where the node contains a cluster of atomic mappings, and the edge denotes the applied operator. Figure 3.a shows an example of the mapping tree. We can insert other atomic mappings into the mapping tree



(i) (*Nest*, `$n1=$n2`) (iii) (*Nest*, `$n1/title=$b1/title`)

Figure 3. The global mapping between the target and the source schemas is represented as a mapping tree.

and construct the global mapping equaling to the one shown in Figure 2. Besides such flexibility, compared to the naive mapping expression, the combined mapping is also maintainable. For example, we can update the operator type to reflect the change of the cardinality constraint. Also, we can modify locally the atomic mappings contained in the mapping tree, while other parts are remained and reused.

3. MAPPING COMBINATION

Atomic mapping. We consider the `FLWR` [17] expression and define atomic mapping as a restricted query formulation. In contrast with the usual XQuery expression, which may contain arbitrary nested queries, an atomic mapping consists of only one `FOR`, one `RETURN` and one optional `WHERE` clauses, and, specifically, has the following general form.

```
for $v1 in SP1(), $v2 in SP2($v1), ……, $vn in SPn($vn-1)
where φ
return () | constant | SPn+1($vj) | <tag></tag>
```

Here, `SP` is a simple path expression with no branching predicates, and $SP_1()$ indicates that SP_1 must start at a schema root, while $SP_k(\$v_{k-1})$ denotes that SP_k is relative to the variable $\$v_{k-1}$. The filter φ is a conditional expression w.r.t. the variables of the atomic mapping. The `RETURN` clause indicates that the atomic mapping may be *empty*, *constant*, *copy*, or *constructor* type.

In Example 2.1, the atomic mapping M_{title} is copy type, and $M_{paper()}$ is constructor type, while the following atomic mapping $M_{cate\text{-}text}$ is constant type.

```
M_cate-text: for $n4 in doc("S1")//novel
             return "novel"
M_review(): for $b1 in doc("S2")//book, $r1 in $b1/review
            return <review></review>
```

We refer to $\$v_k$ ($1 \le k \le n$) as the *F-variable* of the atomic mapping, and $\$v_n$ as the *primary F-variable* (*PFV*) and others as the *prefix F-variable*. Besides binding tuples of instances, the F-variables may also be used to filter the binding tuples, copy the source fragments, or connect with other mappings. In the latter case, as will be seen in Section 4, the prefix F-variables such as `$n2` in M_{title} may be inserted dynamically. In other words, the user only needs to construct M_{title} as follows.

```
for $t in doc("S1")//novel/title return $t
```

In combining mappings, the F-variables sharing the same name will be renamed.

Combination operator. In the following, M_1 and M_2 denote atomic mappings with the general forms. We define a few basic operators to combine M_1 and M_2. The resulting mapping is called *combined mapping*, denoted by M_3. Different from mapping (or query) *composition* [7], where one query can be answered directly using the results of another query, *combining* two mappings is a "parallel" connection, which includes joining the bound sources instances and combining the constructed target instances.

The bound sources are related by combination path, which is a comparison expression w.r.t. the F-variables of M_1 and M_2, e.g., `$n1=$n2` in Example 2.1. In Section 4 we will discuss how to discover such combination paths in combining the mappings.

The operators are responsible for structurally relating the target instances returned by M_1 and M_2. At the same time, we respectively use the *Nest*, *Join* and *Merge* operator types to capture the cardinality constraints contained in the target schema. Note that [13] shows the techniques of generating mappings in the case of referential constraints, which also apply our context and are omitted here. We use the following example to explain the intuition behind the combination operators, while a bit more rigorous formalism will be given when defining the combined mappings.

Example 3.1 As another example, we use the *Nest* operator to combine $M_{book()}$ with $M_{review()}$, and get a combined mapping as follows.

```
for $n1 in doc("S1")//novel
return <book>{
  for $b1 in doc("S2")//book, $r1 in $b1/review
  where $n1/title=$b1/title
  return <review></review>}
  </book>
```

The above mapping indicates that for each `novel`, there will be a new `book` instance returned, no matter whether there are corresponding reviews in the source `S2`. In other words, the *Nest* operator captures the outer-join relationship between the binding tuples of the combined atomic mappings. With the constraints in the schemas of Figure 1, it is valid to apply the *Nest* operator to combine $M_{book()}$ with $M_{review()}$. However, in the shop schema, when the cardinality constraint "*" of `review` is replaced with "+", those books with no reviews should be filtered out. The *Nest* operator cannot satisfy such requirement. Instead, we use the *Join* operator to combine $M_{book()}$ with $M_{review()}$, and get a combined mapping as follows.

```
for $n1 in doc("S1")//novel
let $v:= for $b1 in doc("S2")//book, $r1 in $b1/review
         where $n1/title=$b1/title
         return <review></review>
where count($v)>0
return <book>{$v}</book>
```

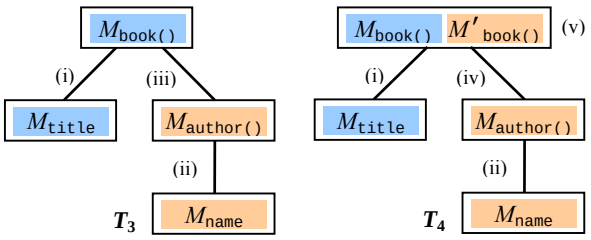
Let ψ denote the combination path. For both the *Nest* and the *Join* operators, we constrain M_1 to be constructor type. Figure 4 respectively shows the resulting combined mapping M_3, which specifies that the target instances returned by M_2 would be nested within those returned by M_1. Syntactically, for the *Nest* operator, the resulting mapping M_3 is obtained by nesting M_1 within the `RETURN` clause of M_2, while for the *Join* operator, M_3 is obtained by introducing a new `LET`-variable `$v` to bind the sequences returned by M_2, and a condition `count($v)>0` to filter out those unsatisfied binding tuples.

```
for $v1,1 in SP1,1(), ……, $v1,n in SP1,n($v1,n-1)
where φ1
return <tag>
   for $v2,1 in SP2,1(), ……, $v2,m in SP2,m($v2,m-1)
   where φ2 and ψ
   return exp </tag>
```

```
for $v1,1 in SP1,1(), ……, $v1,n in SP1,n($v1,n-1)
let $v:= for $v2,1 in SP2,1(), ……, $v2,m in SP2,m($v2,m-1)
          where φ2 and ψ
          return exp
where count($v)>0 and φ1
return <tag>{$v}</tag>
```

```
for $v1,1 in SP1,1(), ……, $v1,n in SP1,n($v1,n-1)
for $v2,1 in SP2,1(), ……, $v2,m in SP2,m($v2,m-1)
where φ1 and φ2 and ψ
return <tag></tag>
```

Figure 4. The *Nest, Join* and *Merge* operators



$M_{author()}$:
```
for $a1 in doc("S1")//author return <author></author>
```
$M'_{book()}$:
```
for $a3 in doc("S1")//author return <book></book>
```

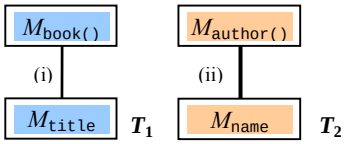
(i) (*Nest*, $n1=$n2) (ii) (*Nest*, $a1=$a2) (iv) (*Nest*, $a3=$a1)
(iii) (*Nest*, $n1/aid=$a1/id) (v) (*Merge*, $n1/aid=$a3/id)

Figure 5. Constructing the general mapping tree

In addition, the application may need to express the "product" relationship between the binding tuples of M_1 and M_2. When the target schema is a default XML view over the relational database [15], for example, the flattened instances may be required to be returned. We use the *Merge* operator to satisfy such demands. In this case, if M_1 is constant or copy type, then M_2 must be empty type; if M_1 is constructor type, then M_2 must also be constructor type and with the same <tag>. For the latter Figure 4 shows the general form of the resulting combined mapping M_3, where the returned instances are merged.

Combined mapping. In defining the above operators, we present the combined mapping M_3 as an equivalent expression. Now, to define the general combined mappings, we first model M_3 as a *mapping tree*. Specifically, the atomic mapping is considered as a node. If M_1 and M_2 are combined using the *Nest* or the *Join* operator, then M_2 is a child node of M_1, and the edge is labeled with (Op, ψ). If they are combined using the *Merge* operator, then the nodes are united into one, which contains both M_1 and M_2, and is associated with ψ.

For example, the following mapping tree T_1 corresponds to the combined mapping $M_{book(title)}$ (see Example 2.1), while the tree T_2 means a combined mapping obtained by combining $M_{author()}$ with $M_{name()}$ using the *Nest* operator.



Being a query, the atomic mapping returns a forest of data (DOM) tree. Specifically, for each binding tuple t, if the filter φ holds, then the atomic mapping will construct a data tree d. In this case we also say that $t \rightarrow d$ holds. Corresponding to the types, the data tree d may be an empty node, a text node, a copied subtree, or an element node. The *Nest* operator applied between $M_{book()}$ and M_{title} indicates that, for each binding tuple t ($t \rightarrow d$ holds) of $M_{book()}$, if there are n binding tuples t_n ($t_n \rightarrow d_n$ holds) of M_{title} satisfying the combination path (i.e., $n1=$n2), then the combined mapping tree T_1 will return the data tree d with d_n ($n \geq 0$) nested within its root node. In our running example, n=1.

Further, we use the *Nest* operator to combine the $M_{book()}$ of T_1 with the $M_{author()}$ of T_2, which means inserting T_2 into the root of T_1. Figure 5 shows the obtained mapping tree T_3, where the new edge "(iii)" is associated with the applied operator and combination path. In another case, we first combine $M'_{book()}$ (see Figure 5) with the $M_{author()}$ contained in T_2, and obtain a mapping tree (say T'_4). Then we use the *Merge* operator to combine the $M_{book()}$ contained in T_1 with the $M'_{book()}$ contained in T'_4, which means merging the root nodes of T_1 and T'_4. The obtained mapping tree T_4 is shown in Figure 5.

Let t_1 and t_2 respectively denote the binding tuples of $M_{book()}$ and $M'_{book()}$, and let d_1 and d_2 respectively denote the data trees returned by T_1 and T'_4. The *Merge* operator applied above indicates that, for each pair of the binding tuples (t_1, t_2), where both $t_1 \rightarrow d_1$ and $t_2 \rightarrow d_2$ hold, if the combination path holds, then the mapping tree T_4 will return a data tree which merges the root nodes of d_1 and d_2. Intuitively, the mapping tree T_4 defines flattening `author` instances.

As can be seen, in a general mapping tree, each node contains atomic mappings, which are combined using the *Merge* operator, and each edge corresponds to the *Nest* or the *Join* operator, which relates the atomic mappings contained in the parent and in the child nodes. Note that there are no the possibilities where both the *Nest* and the *Join* operators are simultaneously applied in the same edge, since the atomic mappings contained in the same node contribute to the same target instances. But now the combination path may be a conjunctive formulation.

Let a node in the mapping tree contain i atomic mappings. We can define the semantics of the general mapping tree in terms of the tuple (t_1, t_2, …, t_i), where each t_i denotes the binding tuple of the corresponding atomic mapping. We omit the details here. In terms of the rules of formulating the combined mapping M_3 as shown in Figure 4, we write the equivalent mapping expression for the mapping tree T_3 as follows.

```
for $n1 in doc("S1")//novel
return <book>{
  for $n2 in doc("S1")//novel, $t1 in $n2/title
  where $n1=$n2 return $t1}{
  for $a1 in doc("S1")//author
  where $n1/aid=$a1/id return <author>… … …</author>}
  </book>
```

Using the normalization rules such as shown in [8], the above mapping may be minimized into the expression fragment of the

one in Figure 2. The ways to construct the mapping tree are multiple. Alternatively, for example, to obtain the mapping tree T_4, we can first merge T_1 with $M'_{\texttt{book()}}$, and then insert T_2. We believe that such flexibility would be popular in constructing the global mapping, especially for the complicated applications. Note that the sibling nodes are order sensitive in the mapping tree.

4. AUTOMATED SUPPORT

Besides the flexibility, the combined mapping is also maintainable. In this section we further combine our work with the current automated techniques. Specifically, we show how to generate the atomic mappings and discover the combination paths. Also, we analyze the maintainability of the combined mappings.

Given the target and the source schemas, the atomic mappings are first generated in terms of the matches produced by a tool of schema matching (e.g., `LSD` [6]). Interestingly, due to the maintainability of the combined mapping, our model does not require that all the produced matches should be desired. Next, keeping the target schema in mind, the user incrementally combine the atomic mappings using the operators, i.e., inserting and merging the mapping trees. In this process, a tool may be used to suggest the candidate combination paths.

Maintainability. As motivated in Section 2, the requirements of modifying and maintaining the mappings may result from the way of incrementally constructing the mappings, the refinement of the mappings, or the evolution of the schemas. With our mapping model, they are all reduced to maintaining the mapping trees, e.g., inserting and merging subtrees. In the mapping trees, the atomic mapping may be related through the combination paths with other atomic mappings contained in the same, parent, or child nodes. As the modifications occur in the mapping tree, these combination paths would need to be discovered or adjusted, while other parts of the mapping trees would be remained and reused.

Modifying the combination operators (*Nest* and *Join*) will not affect the combination paths that relate the atomic mappings. For example, consider the Example 3.1 and the mapping tree in Figure 3, if in the target schema the cardinality constraint "*" of `review` is replaced with "+", then we only need to modify the operator type in the edge "(iii)" from *Nest* to *Join*.

On the other hand, for the following modifications, we find out those pairs of atomic mappings (M_1, M_2) between which the combination paths would need to be discovered or adjusted. In the following, S represents the set of (M_1, M_2), Tr corresponds to the mapping trees and r is its root node, and $atoms(n)$ denotes the set of atomic mappings contained in the node n of the mapping tree. We also use p and c_i to respectively denote the parent and the children nodes of the node n.

Inserting Tr_1 into the node n of Tr_2.
$S \leftarrow (M_1, M_2)$, where $M_1 \in atoms(r_1)$ and $M_2 \in atoms(n)$.

Merging Tr_1 into the node n of Tr_2.
$S \leftarrow (M_1, M_2)$, where $M_1 \in atoms(r_1)$ and $M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i)$.

Updating the atomic mapping M contained in the node n.
$S \leftarrow (M_1, M_2)$, where $M_1 \in \{M\}$ and $M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i) - \{M\}$.

Removing the atomic mapping M from the node n.
$S \leftarrow (M_1, M_2)$, where $M_1, M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i) - \{M\}$ and $M_1 \neq M_2$.

Consider the mapping tree T_4 shown in Figure 5. The atomic mappings $M'_{\texttt{book()}}$ and $M_{\texttt{author()}}$ contained in the tree are related by the combination path `$a3=$a1`. The modification of removing $M'_{\texttt{book()}}$ from the root of T_4 would affect the atomic mapping pairs $\{(M_{\texttt{book()}}, M_{\texttt{title}}), (M_{\texttt{book()}}, M_{\texttt{author()}})\}$, which are respectively taken to discover the candidate combination paths (discussed in a moment). Guided by these candidate paths, the user then may adjust the above path to `$n1/aid=$a1/id`.

Note that such adjustment is not additionally introduced by our mapping model. In contrast, such maintainability inherent in our model provides the opportunities for automating the process of mapping maintenance, be it caused by schema evolution, mapping refinement, or other factors. In our going work, we are developing methods to assign priorities to the discovered combination paths, and to heuristically reduce the amount of the potentially affected atomic mapping pairs. Additionally, our first experiment in the book domain shows that there are averaged 1.2 atomic mappings contained in each node of the mapping trees.

Schema matching. Schema matching [6, 18, 5] produces a set of semantic correspondences (matches) between the elements of the schemas, from which atomic mappings can be generated. For example, a 1-1 match would specify that the element `name` in `S2` match `who` in the shop schema `T`. Then, an atomic mapping $M_{\texttt{who}}$ could be generated via specifying its PFV (see section 3) by the element `name`. As to complex type (e.g., 1-*n*) matches, several atomic mappings may be generated, which then are combined using the operators. In practical mapping construction, it is often the case where the matches initially used to obtain the mappings are not the desired. Fortunately, as shown above, our model is able to make it local to update the corresponding atomic mappings.

Discovering combination path. Combination path ψ is used to relate the bound source instances of the atomic mappings, and can be heuristically discovered in terms of the semantic relationships between the source schema elements. As presented in [10, 13, 19], such relationships are captured by the *structural*, *user* and *logical associations*, which respectively describe a set of associated schema elements.

Consider the source schema `S1`. For example, the elements such as `novel` and `title` are in a structural association, while the elements such as `novel`, `title`, `author` and `name` are in a logical association. To relate the novel with the book reviews, the user may explicitly relate the `title` elements. Then the elements such as `novel` and `title` of `S1`, and `book` and `title` of S2 are in a user association.

Let $e1$ and $e2$ respectively denote the source schema elements specifying the PFVs of the atomic mappings M_1 and M_2. If the elements $e1$ and $e2$ are in a structural association, then ψ may be formulated in terms of their common path. Consider to combine the atomic mappings $M_{\texttt{book()}}$ and $M_{\texttt{title}}$. Their PFVs (i.e., `$n1` and `$t1`, see Example 2.1) are respectively specified by the elements `novel` and `title` of `S1`, which are in a structural association. In terms of the common element, i.e., `novel`, the path `$n1=$n2` is generated, where, as a prefix F-variable, `$n2` may be dynamically inserted into $M_{\texttt{title}}$, if it does not exist.

If $e1$ and $e2$ are in a structural but in a user association, then ψ may be formulated with the path assigned by the user. Lastly, if they are neither in a structural nor in a user, but in a logical association, then ψ may be formulated in terms of the referential path between the schema elements $e1$ and $e2$. For example, the combination path, `$n1/aid=$a1/id`, of relating $M_{\texttt{book()}}$ and $M_{\texttt{author()}}$ is generated in terms of the logical relationship between the elements `novel` and `author`. In a similar way, the prefix F-variables can also be introduced dynamically, if they are not defined in constructing the atomic mappings.

5. RELATED WORK

Schema mappings are extensively used in many modern applications such as the data integration and data exchange systems. To alleviate the burden on the user for constructing and maintaining such semantic mapping, many efforts have been made to pursue maximum automatic support, which can be classified into works on schema matching and mapping discovery. The former focuses on discovering semantic correspondences between the elements of a pair of schemas (e.g., [11, 6, 18, 5], see also [14] for a recent survey). Among these, [18, 5] discussed how to obtain complex type matches, based on the domain ontology and the multi-matcher mechanism. Under the assumption that the desired matches have been given, [10, 13] proposed to further discover candidate schema-to-schema mappings. In their approach, the matches are related using the chase technique [1, 12] to search the semantic relationships between the source or target schema elements. In terms of such semantic relationship, [19] proposed to compute the matches affected by schema evolution, and then re-employ the mapping discovery system to adjust the mappings.

In contrast, we propose a flexible and maintainable model to represent XML mappings. We think that the global mapping can be constructed in a piecemeal fashion, where, to some extent, the partial mappings resemble subgoals in datalog programs. Also, the XML-QL language [4] allow for defining partial mappings. Yet we provide a richer scheme for combining the results of the different partial mappings. In our model, mappings are considered as the first-class citizens that can be operated. Such idea also was used in [2, 9] to deal with the management of meta data. Yet the subjects these works focused on are not the mappings but the matches between the schema elements. Additionally, there have been many GUI-style tools developed to assist the user to construct the mappings (queries) in XQuery, where the queries are formulated in terms of the syntax ingredients such as the `FOR`, `LET`, `WHERE` and `RETURN` blocks [16]. Differently, our mapping model is based on the semantic relationships between mappings to be connected.

6. CONCLUSION

Semantic mappings are key for enabling a variety of data sharing scenarios. This paper described the flexible and maintainable mapping model, where the atomic mapping and the combination operator are the main components. Conceptually, to construct the global mapping for the whole target schema, we first construct the local atomic mappings for the single target schema element, and then combine them using the operators. We represented the combined mappings as the mapping trees. Then the mapping problem is reduced to the problem of constructing and maintaining the mapping trees. We analyzed the maintainability of the mapping tree, and presented how to combine our work with the current automated techniques.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their insightful comments.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995.

[2] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. of CIDR*, 2003.

[3] Clio. http://www.cs.toronto.edu/db/clio/

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *proc. of WWW*, 1999.

[5] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *proc. of SIGMOD*, 2004.

[6] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine learning approach. In *proc. of SIGMOD*, 2001.

[7] J. Madhavan and A. Halevy. Composing mappings among data sources. In *Proc. of VLDB*, 2003.

[8] I. Manolescu, D. Florescu, and D. Kossman. Answering XML Queries on Heterogeneous Data Sources. In *proc. of VLDB*, 2001.

[9] S. Melnik, E. Rahm, P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *proc. of SIGMOD*, 2003.

[10] R. Miller, L. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. of VLDB*, 2000.

[11] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *proc. of VLDB*, 1998.

[12] L. Popa and T. Val. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *proc. of ICDT*, 1999.

[13] L. Popa, Y. Velegrakis, R Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. of VLDB*, 2002.

[14] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. The *VLDB Journal*, 10(4): 334–350, 2001.

[15] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *proc. of VLDB*, 2001.

[16] Stylus Studio. http://www.stylusstudio.com

[17] XQuery. http://www.w3.org/XML/Query

[18] L. Xu and D Embley. Using domain ontologies to discover direct and indirect matches for schema elements. In *Proc. of the Semantic Integration Workshop at ISWC*, 2003.

[19] Y. Velegrakis, R. J. Miller, and L. Popa. Preserving mapping consistency under schema changes. The *VLDB Journal*, 13(3): 274-293, 2004.