Detailed Technique Report on XMLSnippet

Yiqi Lu, Sheng Huang, Yanghua Xiao
Fudan University

Abstract
XMLSnippet is a tool to recommend XML snippets automatically by mining structural patterns from application to assist programmer to generate proper XML configuration during product phase. These actions are realized by mining closed frequent tree patterns from existing projects without awareness of the syntax of the specific frameworks. XMLSnippet is built on two major techniques that are borrowed from database communities: closed frequent XML tree pattern mining and sub-tree query. In this paper, we will describe those major techniques used in XMLSnippet offline mining in detail.

1 The framework of XMLSnippet

In the design of XMLSnippet, we want to achieve following objectives.

1. The tool should be syntax neutral to support frameworks with different syntax. The tool should be able to extract generalized patterns over all kinds of frameworks.

2. The tool should be able to provide online intelligent coding suggestions with reusable XML Snippet when the programmer is editing the XML configuration files.

The architecture of XMLSnippet is shown in Figure 1. The architecture consists of three major phases: pre-processing, offline mining and online query. In phase 1, application repositories are first preprocessed, and then the result (XML Repository) will be fed to closed frequent sub-tree mining. Closed sub-tree patterns in the XML configuration files are of special interest to us since they contain enough information of editing rules in framework-based programming. All the resulting patterns are stored in XML tree pattern database, which will be indexed by prefix tree. Sample source codes will be indexed by inverted index that will be used to speedup CQ. The cores of XMLSnippet are closed frequent tree mining and prefix tree/source code indexing. Frequent tree pattern mining distinguishes our tool from other code recommendation tools that utilizes frequent item set patterns[3], or sequential patterns[4]. Online query phase is responsible for providing effective online assistance for programmers.

2 Preprocessing

It is hard to find meaningful patterns from frameworks for diverse proposes. And it is too heavy to use natural language methods just for identifying types of
Table 1: selected feature strings

<table>
<thead>
<tr>
<th>XML Doc Type</th>
<th>Feature String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>web-app</td>
</tr>
<tr>
<td>iBatis</td>
<td>iBatis.apache.org//DTD SQL Map Config 2.0</td>
</tr>
<tr>
<td>Hibernate Map.</td>
<td>Hibernate Mapping DTD</td>
</tr>
<tr>
<td>Hibernate Conf.</td>
<td>Hibernate Configuration DTD</td>
</tr>
<tr>
<td>Struts1.x</td>
<td>DTD Struts Configuration 1</td>
</tr>
<tr>
<td>Struts2.x</td>
<td>DTD Struts Configuration 2</td>
</tr>
</tbody>
</table>

XML documents. Fortunately, many typical features that can be identified from XML docs in framework programming. Therefore, we classify input XML configuration files into different categories manually according to the information in the head of these XML documents. The features we selected as identifying criteria are listed in table 1.

3 Closed frequent subtree pattern mining

XML docs are modeled as unordered labeled rooted trees, where each node in the tree is an element, or attribute. More formally, any attribute of an element will be a child of the element node; a sub-element will be a child of its parent element. In this way, we can minimize information loss of the original XML docs, which guarantees the quality of succeeding processing. As an example, Figure 3 shows the XML tree of the XML doc. We use a triple $T = (N, E, ft)$ to denote an XML tree extracted from an XML configuration file, where $N$ is the set of nodes (including two types of nodes, one is element node $N_{elem}$, while the other is attribute node $N_{attr}$), $E$ denotes the edge between parent and child, and $ft$ indicates the framework types of $T$. Since the nodes are distinguishable from each other, the tool could regenerate the original XML fragment from the XML tree by setting $N_{elem}$ as XML elements, the child $n_{attr}$ of $n_{elem}$ as the
attribute of \( n_{elem} \). Notice that the value of attributes will be ignored in the tree structures since they vary under different project context.

Let \( D \) be a database of XML trees. For a given pattern \( t \), we say \( t \) occurs in a XML tree \( s \in D \) if \( t \) is a subtree of \( s \). The support of a pattern \( t \) in the database \( D \) is defined as the number of XML tree \( s \in D \) such that \( t \) is a subtree of \( s \). A pattern \( t \) is called frequent if its support is greater than or equal to a minimum support (\( \text{minsup} \) is 10% of the XML docs for the same type of frame in this paper, based on the initial empirical experiments) specified by a user. The frequent subtree mining problem is to find all frequent subtrees in a given XML tree databases. One nice property of frequent subtrees is the apriority property, i.e., if \( t \) is frequent, all its subtrees are also frequent. A frequent tree \( t \) is and closed if none of the proper supertrees of \( t \) has the same support that \( t \) has.

Given the collections of XML trees constructed as above, reusable XML snippet for frameworks can be found by closed frequent subtree pattern mining[1, 2]. Notice that we are interested in mining closed frequent subtrees, instead of mining complete frequent subtree patterns due to the following reasons. The complete set of subtree patterns is redundant since subtrees of frequent tree patterns are also frequent. One way to reduce such redundancy is only keeping closed patterns. In general, there are fewer closed frequent subtrees compared to the total number of frequent subtrees and we can recover all frequent subtrees with their supports from the set of closed frequent subtrees with their supports. Hence, mining closed frequent subtrees is an effective way to mine informative yet non-redundant patterns from XML configuration repositories.

In general, different frameworks will have disparate syntax or semantic, which is one of the challenges in sample code recommendation for framework based applications. However, by modeling sample code recommendation as frequent tree mining, all syntax or semantics of different frameworks are modeled as a unified model: tree patterns, because the combination of attributes syntax and hierarchical relationship of elements are all preserved in frequent tree patterns. Hence, our tool can mine meaningful structural patterns in XML files independent of frameworks syntax. Most previous researches about coding assistance reduce the problem to frequent item set mining or frequent sequence mining[3, 4]. Compared to them, frequent closed tree mining preserves the structural information expressed by the XML configuration files, which provides
opportunities to find more interesting patterns and more hidden knowledge from framework based source repository.

We leverage the closed tree pattern mining approach used in [2] to extract the informative structural patterns required in expressing the logic controlled by XML configuration snippet. In the paper we use $fst = (N, E, f)$ to denote one frequent sub-tree pattern, where $fst.N$ denotes the nodes in $fst$, $fst.E$ denotes the edges in $fst$, and $fst.f$ denotes the frequency of this $fst$ in the same $ft$. Like $T$, $fst$ could also be restored to XML Snippet.

4 Prefix tree/source code indexing

Once the closed frequent tree patterns are extracted, we need to build an index to accelerating retrieve of these patterns, ensuring the online performance of coding assistance. Suppose a programmer has finished editing a node of XML tree, we will use location of current cursor, denoted by $c$, to indicate the node just been edited in following text. We call the root from the root of the XML tree to $c$ as the prefix of current cursor. Such a prefix essentially is an element/attribute sequence from the root the node just before the current cursor. Our tool needs to retrieve tree patterns that contain exactly the same prefix. For this purpose, we can extract all paths (i.e. element/attribute sequence) of XML trees in our repository. Then, build a prefix tree on all these sequences. We use $fst(p)$ to denote the function that get the set of frequent tree pattern contain element sequence $p$. The detailed algorithm is given in 1.

**Algorithm 1 Pattern tree indexing**

**Input:** frequent tree set $FST$

**Output:** Trie like path index tree $IDXT$

1: $IDXT \rightarrow \emptyset$

2: for each frequent subtree pattern $t \in FST$ do

3: for each path $p \in t$ do

4: if $IDXT$ has $p$ then

5: $p.C \leftarrow p.C \cup t$

6: else

7: insert path $p$ into $IDXT$ in a Trie-like way

8: $p.C \leftarrow t$

9: end if

10: end for

11: end for

As shown in Algorithm 1, the index of frequent pattern tree is a trie-like index. For every root-to-leaf path in every found frequent pattern tree $t$, algorithm 1 will first query if the path is already in the index tree $IDXT$. If the path is inserted before, then just add a link to frequent pattern tree $t$ at the leaf of this path in $IDXT$. Otherwise, insert this path into $IDXT$ and initialize the link frequent pattern tree set at the leaf of this path with $t$. After the $IDXT$ has been constructed, we can calculate the support of each $IDXT$ node bottom-up linearly, which will be used in SQ later. During the course of editing XML configuration file, XMLSnippet will generate a query path according to the existing XML document under development. Then XMLSnippet will query
the generated path on the index tree IDXT. Since the index tree is trie-like structure, the query progress is quite straight-forward. The detailed algorithm 2 is given below.

**Algorithm 2** XMLSnippet query on index tree

**Input:** A query path $p_q$, index tree IDXT

**Output:** snippet set $S$

1. if path $p_q$ is found in IDXT then
2. $S \leftarrow \emptyset$
3. for every child node $n$ of $p_q$ do
4. $S \leftarrow S \cup \{n\}$
5. end for
6. sort $S$ according to the support of each node
7. return $S$
8. else
9. print XMLSnippet has no proposal
10. end if

Then After obtaining a frequent tree pattern, we use function $sample(fst)$ to get the set of corresponding samples $s$ for frequent subtree pattern $fst$. Since the relationship between instances and patterns has already been built in the offline mining process, we can directly return samples (instance) for a tree pattern. Sample s is a triple of the form $< x, c, d >$ with $s.x$ denoting sample XML snippet, $s.c$ denoting the sample client source code and $s.d$ denoting the size of $s.x$ (i.e., number of XML nodes). In general, for a pattern, there exist multiple samples. We use $d$ to rank these samples. The smaller the $d$ is the more simple the sample is and consequently more interesting to programmers.

For a sample $s$, to online retrieve client source code for XML snippet, we use Lucene[5] to index all key words in client source codes. During the preprocessing step, we extract unordered XML tree from XML configuration files in the repository in one hand. On the other hand, we use Lucene to index all the source code files (including .jsp, .java, .xml and etc.) as documents. As mentioned before, in the process of building XML trees, we do not keep those attribute values as tree nodes. Instead, we regard these attribute values as search key words which will be used in the CQ later as Lucene query keywords. Those attribute values are stored in the form of triple $< type, elem, attr, v >$ with $type$ denoting the XML document type, $elem$ and $attr$ denoting which element and attribute this value belongs to and $v$ denoting the attribute value.

**References**


