XMLValue: XML Configuration Attribute Value Recommendation

Yiqi Lu
Fudan University
Shanghai, China
Email: luyiqi@gmail.com

Jiaqing Liang
Fudan University, Shuyan Technology
Shanghai, China
Email: l.j.q.light@gmail.com

Yanghua Xiao
Fudan University
Shanghai, China
Email: shawyh@fudan.edu.cn

Sheng Huang
IBM Research-China
Beijing, China
Email: huangssh@cn.ibm.com

Deqing Yang
Fudan University
Shanghai, China
Email: yangdeqing@fudan.edu.cn

Wei Wang
Fudan University
Shanghai, China
Email: weiwang1@fudan.edu.cn

Haibo Lin
Search Technology Center Asia, Microsoft
Beijing, China
Email: haibolin@microsoft.com

***Abstract*—Frameworks are popularly used to reduce implementation complexity and improve productivity. Unfortunately, most frameworks are quite complex and not well documented. Hence, correctly and effectively programming with Framework is still a great challenge. One of the significant obstacles for us to smoothly use Framework is the complicated attribute value configuration of XML files. To overcome these difficulties, we present XMLValue to automatically recommend XML attribute values using association rules and NLP techniques. Experimental results show that our tool is efficient and effective for mining reusable configuration snippets, and has significantly shorten development time for framework based programming, and is general enough to support a variety of frameworks, and has real time performance for code assistance.**

I. Introduction

In today's software development, frameworks have been widely used to reduce implementation complexity and improve software productivity. Developers in general need to edit XML configuration files and client codes to implement the application logic in framework-based programming. We use Example 1 to show a typical procedure in a framework based programming. This kind of programming in general consists of two major steps: 1) developers first need to write the client code, such as the JSP web pages and the Java class files in Fig. 1(a); 2) then, they need to edit the XML configuration files, such as the file shown in Fig. 1(b). The second step usually is mandatory in framework based programming.

Example 1 (Framework based programming): In this example, we show how to implement the typical login logic by Struts [1], which is widely used to implement MVC in Java based web applications. The code snippet of this example is shown in Fig. 1. At the Presentation tier, Struts implements MVC by the following steps:

- 1) the user submits a Form through a JSP page;
- 2) a Controller class of Struts passes the Form to a specific Action class, which is guided by a configuration file entitled as struts-config.xml;
- 3) the Action class is executed and returns a result string (e.g. "success", "fail");

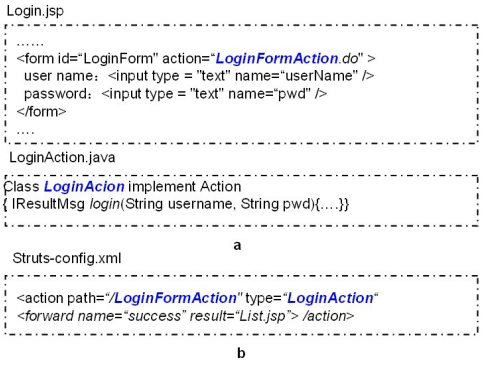

Fig. 1. Sample usage scenario

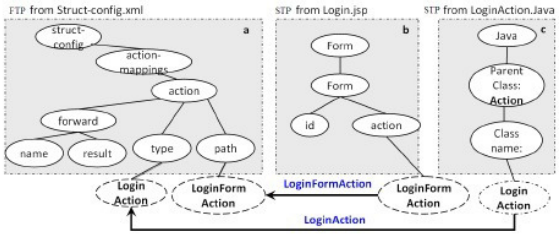

Fig. 2. An example of association rule

- 4) the Controller directs the control flow to the next JSP page specified in the same configuration file.

In this example (as shown in Fig. 1(a)), Login.JSP defines a LoginForm which will be submitted to the web server when a user logins. The struts-config.xml file has an <action...> element that relates this request to a Java class LoginAction, and also a <forward...> element that directs the "success" login to the JSP page List.jsp, where the users can do other operations.

Problem statement: With the help of IDEs and other recently developed tools [2], [3], [4], [5], [6], [7], [8], [9], [10], programmers nowadays can efficiently finish the client codes and delineate the structures of XML configuration files. But, it is not enough yet for a complete implementation of

the application logic. Continue with the previous example. To realize the login logic, we need to fill in the *"path"* and the *"type"* attributes of *"action"* element with the appropriate values and all these values can be found from the source codes. Developers still need to manually go through the whole project and find the proper literal values in the source codes as the attribute values in the XML configuration files. This step is still *error prone* and *time consuming* due to the following reasons. First, when developers manually input the attribute values, there may exist typos that will wreck the application logic. Second, there may exist many literal value candidates in the source code that can be filled as the XML attribute value. Developers may select a wrong candidate.

The above problem can be solved if an *XML attribute value recommendation tool* is available. Such a tool generally is expected to appropriately recommend a value or a set of candidates ranked by a certain confidence score. More formally, given an XML file D and the context c that the programmer is editing, we need to recommend a list of literal values: $\{(v, r(v))\}$, where each v indicates a candidate and $r(v)$ is v's ranking score. Given the ranked list, developers only need to select an appropriate value. As a result, the productivity and correctness can be improved.

Overview of XMLValue: In this paper, we propose such a tool, called XMLValue, which can be regarded as an extension of Eclipse XML Editor [7] and our previous work [6]. XMLValue aims to provide XML attribute value recommendation and defect detection features. We show our experience of using data mining and natural language processing techniques to build this tool. XMLValue can prompt candidate attribute values ranked by its confidence score. With XMLValue, we imagine that programmers can save time from typing all the elements, reduce the chance of typing incorrect attribute values.

XMLValue realizes XML attribute value recommendation by establishing the mapping between attribute values in XML configuration files and literal values in source code files first (shown in Figure 2). To achieve this, we first model these two kinds of files as trees and then mine association rules between trees. In this step, we fully utilize the syntax information implied in the tree structure of source codes and XML trees. But such a mapping may be only correct in syntax but invalid semantically. For example, we may also need to implement the logout logic in our running example. Thus, the attribute values like *"LogoutAction"* can also be mapped to the action element for the login logic. To solve this problem, we further suppress those semantically incorrect mappings according to the semantic distance between mapped value pairs. Before we can calculate the semantic distance, we first need to break words in literal values like *"LogoutAction"* into *"log out action."* This is known as *word breaking* problem in nature language processing (NLP) and we solve it by a probabilistic approach.

Having these mappings between attribute values and literal values as well as the semantic ranking mechanism, XMLValue can detect defects easily. XMLValue will get all the XML element sequences in the XML configuration files in the projects. After that, XMLValue will find all possible literal value candidates according to the rules already mined from repositories. XMLValue will throw a warning when it detects a candidate with smaller semantic distance or an error in an attribute value. Hence, in this paper, we mainly focus on describing XML attribute recommendation.

II. ARCHITECTURE

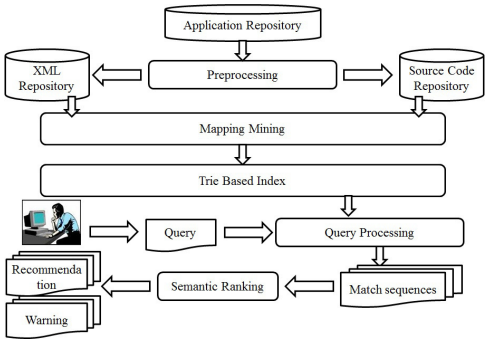

Fig. 3. XMLValue architecture

The architecture of XMLValue is shown in Fig. 3. The workflow of XMLValue consists of two major phases: offline and online. In the offline phase, we mined the patterns and semantics from the application repositories. In the online phase, the tool performs value recommendation and defect detection based on the knowledge mined in the offline phase.

In the offline phase, files in XML repositories and source code repositories are first parsed and modeled as trees. Then XMLValue mines mappings between sequences in XML tree and sequence in source code trees. These sequences are further organized as a Trie or a prefix tree, which serves as the index structure to accelerate the value recommendation and defect detection in the online phase.

There are two optional tasks in the online phase: XML attribute value recommendation and defect detection. These two tasks share a similar workflow. First, the context of the XML document are fed as the query to our tool. The query processing component of our tool will parse the query and retrieve candidate recommended values. In order to give developer high quality attribute value candidates, we further need a semantic ranking component, which is responsible for ranking the candidates according to its semantic similarity to the context under editing.

The following text is organized as follows. We present in Section III the detail of major techniques used in XMLValue, including: mapping mining and semantic ranking. Then, we present the experimental study in Section IV. Before our tool can run smoothly, a preprocessing step needs to be executed first. By this step, we clean the application repositories and classify XML repositories into corresponding categories. Next, we elaborate this component.

Preprocessing: In general, it is hard to find meaningful patterns from frameworks serving for different purposes. For

example, iBATIS and Struts generally will not share a meaningful frequent pattern. Hence, we only mine patterns within applications of the same framework. For this purpose, we first need to classify input XML configuration files into different categories. Many typical features that can be identified from XML docs in framework programming, such as "SQLMap" in iBATIS, "struct-config" in Struts, are sufficient for us to correctly classify XML documents into corresponding categories in terms of framework type. The mapping mining is processed over separate groups of framework XML documents one by one.

III. Online Phase

In this section, we will elaborate two major techniques used in XMLValue: mapping mining, semantic ranking.

A. Mapping mining

We first describe how we obtain mappings by combining XML repository and source code repository, to help online attribute value recommendation and defect detection. In order to guide attribute value recommendation, we need to build mapping between attribute values in XML configuration files and literal values in source codes. Therefore, we use tuples to represent both XML attribute values and source code literal values in basic rule obtaining phase.

1) Tree modeling: The input of our mining approach consists of two parts: XML configuration files and source codes. Both of these two parts can be modeled as rooted trees: XML trees and syntax trees. For a rooted tree, for each node *v* in the tree, there exists a path from the root to its parent. We call these paths for node *v* as prefix of *v* in the tree.

We model XML documents as unordered labeled rooted trees, where each node in the tree is an element, attribute name or attribute value. More formally, each attribute of an element will be a child of the element node; a subelement will be a child of its parent element; an attribute value node is the unique child of attribute name node.

Example 2 (XML tree): Fig. 2 shows the XML tree of the XML doc in Fig. 1(b). In the XML tree in Fig. 2(a), nodes in dashed line are attribute value nodes, such as "*LoginAction*" and "*LoginFormAction*" and other nodes are elements and attribute names.

Under this model, each attribute value is a leaf node in the XML tree. Each node of an attribute value corresponds to a unique path from the root of the XML tree to its attribute name node. Hence, we can represent an attribute value in an XML configuration file as a tuple, $< v_{attr}, s_{XML} >$, where v_{attr} is the attribute value and s_{XML} is the prefix of v_{attr} in the XML tree.

Example 3 (XML element sequence): As shown in Fig. 2(a), the attribute value node *"LoginAction"* corresponds to a prefix path s_{XML}= <*"struts-config", "action-mappings", "action", "type"*>.

Similarly, we can model the source code repositories as trees. Specifically, we model each source code file as an *abstract syntax tree* (AST), and model JSP files, SQL files as trees too. And therefore, for each attribute literal value, there also exists a corresponding path from tree root to the literal node. More formally, we use a tuple $< v_{literal}, s_{AST} >$ to represent a literal value in the source code file, where $v_{literal}$ is the literal value and s_{AST} is its prefix in AST tree.

Example 4 (AST sequence): In Fig. 2(c), for the literal value "*LoginAction*", there exists a prefix path s_{AST} as <"*TypeDeclaration*", "*class*", "*Name*", "*LoginAction*">.

2) Mining solution: We next propose our problem definition. If there exist a tuple pair $< v_{attr}, s_{XML} >$, $< v_{literal}, s_{AST} >$ such that $v_{attr} = v_{literal}$, then we can find a link between the two tuples. Such a link allows us to build a mapping between a XML node sequence s_{XML} in the XML configuration file and s_{AST} in the source code file for the attribute value v_{attr} or $v_{literal}$. In other words, we can build a new triple in the form of $< v_{attr} = v_{literal}, s_{XML}, s_{AST} >$. Then, our attribute value recommendation problem becomes: *given the s_{XML} in the current XML configuration file under editing, infer v_{attr}.*

Our basic idea to solve above problem is that, among all $< x, y, z >$ triples (where x is the attribute value, y is the XML sequence and z is the AST sequence) that can be mined from the repositories, if the AST sequence z is the most frequent pattern that co-occurs with the XML sequence y, the value corresponding to pattern z is the value to be recommended.

We regard each triple $< x, y, z >$ as a transaction. Let T be the collection of all transactions. An association rule $Z \rightarrow Y$ with confidence c on T means that if a transaction contains Z, it contains Y with probability of c. For a pair of s_{XML}, s_{AST}, we define the confidence of association rule by the following equation:

$$confidence(s_{AST} \rightarrow s_{XML}) = \frac{f(s_{AST}, s_{XML})}{f(s_{AST})} \qquad (1)$$

, where $f(s_{AST})$ is the frequency of s_{AST} in T.

Thus, our mapping mining problem will be: *given an XML sequence y and a threshold θ, find the AST sequence z whose confidence score from y to z is no less than θ.* For each XML sequence whose confidence score is larger than the given threshold, we create a corresponding association rule from the syntax node to an attribute value node.

Given the above association rules, XMLValue can recommend appropriate values for an XML attribute. Suppose that the programmer is editing the struts-config.xml shown in Fig. 1, and he is about to edit the attribute value of the attribute *"type"*. Our tool, XMLValue, accepts the XML prefix <*"struts-config", "action-mappings", "action", "type"*> as the input and then queries all the association rules mined from application repositories. After querying, XMLValue finds the AST sequence in the most confident association rule is <*"TypeDeclaration", "superclass", "Action", "class", "name"*>. And then XMLValue searches the current project for Java code files that contain such a pattern sequence and finally returns a candidate list which contains the value *"LoginAction."*

B. Semantic ranking

Previous mapping helps find some candidate attribute values. But this step is not good enough for attribute recommendation and defect detection. Because path based associations mining only utilize the syntax information implicated in the tree structure of AST tree and XML tree. Such a mapping cannot fully reflect the semantic relevance of correlated attributes between XML tree and AST tree. We still need to take advantage of semantic information contained in the literal values and attribute values to improve our solution. Example 5 clearly show the motivation to further use sematic information to improve the accuracy of value recommendation.

Example 5 (Motivation of semantic ranking): Assume that a programmer is editing a struts-config.xml shown in Fig. 1. After finishing the attribute *"path"* and entering *"LoginFormAction"*, he moves to the next attribute "type". If we only use path-based association rules to recommend attribute values, it is quite possible for XMLValue to recommend attribute values like *"LogoutAction"*, which is syntactically right but semantically wrong. If XMLValue takes a look at the sequence that shares enough prefix with the current sequence, XMLValue will prompt attribute values that relevant to *"Login"* instead of *"Logout."*

More formally, we need to solve the following problem in this section. Given an XML prefix y, let $Z = \{z_1, \ldots, z_k\}$ be the set of confident AST prefix sequences mined from application repositories. Suppose z_1 is the most confident AST prefix. According to the prefix, we may find multiple literal values to recommend. *Then, which one should we recommend?* Next, we propose a semantic based ranking mechanism to filter out those semantically incorrect mappings.

But, before we give the ranking scheme, we first need to solve the word breaking problem because usually programmers will concatenated several words into a single word to name variables or values. It is difficult to understand the semantics of these concatenated words. Hence, word breaking is necessary. Fortunately, experienced programmers tend to use high quality naming, which preserves the semantic of original words and is friendly for word breaking. It is a good practice to give meaningful class names, method names and variable names. These meaningful names will reveal the logic of the code, which makes it much easier for programmers to understand codes. This gives us a chance to use word breaking technique to reveal the semantic information hidden inside names and literal values in source codes. We illustrate the necessity of word breaking in Example 6.

Example 6 (Meaningful attribute values): Take fig. 1 as an example. There are many concatenated words used as variable names such as *"LoginAction"* and *"LoginFormAction"*. All the attribute values are meaningful. For instance, *"LoginAction"* means that it is a subclass of Action and is responsible for the login logic. *"LoginFormAction"* means that this subclass of Action deals with the information in the login form.

1) Word breaker: By word breaker, we expect to break the literal values at proper boundaries to extract the semantic information hidden in the literal words. For example, a good word breaker will break *"LoginAction"* into *"log in"* and *"action"*, by which we can understand the semantic of the literal value. Word breaking has been extensively studied and has become an essential component of many commercial products, such as Microsoft Office and SQL server. It is also critical for many NLP tasks such as entity recognition, information retrieval, and machine translation.

We adopt the word breaking solution used in [11] in our XML attribute recommendation scenario. The solution formulates the word breaking problem in a probabilistic framework: given a string u, the word breaker is expected to find the best segmentation s that maximizes the sum of the log likelihood of each word derived by the segmentation. We use the data set provided by Microsoft Ngram service [12] to get the probability of every possible word. As an example, the final output of breaking *"LoginAction"* is *"log in action"* since the log-likelihood of *"log in action"* is the largest in all possible results.

2) Ranking using taxonomy: After obtaining the words, we use a semantic knowledge base to calculate the proximity between pairs of literal values to recommend the most similar literal value. Before this, we first need to replace each acronym with its corresponding word. The words that produced by our word breaker may contain acronyms. For example *"lst"* may stand for *"list"* since programmers prefer short name for fast coding and easy remembering. We crawl all the acronyms from [13] and build a mapping from acronyms to its corresponding word.

Then we use the WordNet [14] taxonomy to calculate the semantic similarity between two literal values. WordNet is a large English lexical database. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. In this paper, we use the Wu and Palmer [15] metric. This similarity metric measures the depth of two given words in the taxonomy and the depth of the least common subsumer (LCS) concept and then combines these figures into a similarity score as follows:

$$sim(u, v) = \frac{2 * depth(LCS)}{depth(u) + depth(v)} \tag{2}$$

where u and v are two synsets in WordNet. Given this metric, we first convert each literal value into a set of split words (or phrases). For each split word, we find its corresponding synset in WordNet. Given a literal value L_1, let $S(L_1)$ be the set of all synsets in WordNet. Now for two literal values L_1, L_2, we first compute similarity by Equation 2 for each pair of $s_1 \in S(L_1)$ and $s_2 \in S(L_2)$. We finally use the mean of the all pairwise similarity as the final similarity score for the two literal values.

Continue with the previous example. Suppose two candidate literal values L_1 ="*LoginAction*" and L_2 ="*LogoutAction*" are found. Now we need to tell which is more similar to the

attribute value $A =$*"LoginFormAction"* that has already been entered. We first split these three literals into *"log in form action"*, *"log in action"* and *"log out action"*. Calculation shows that $sim(L_1, A)$ is significantly larger than $sim(L_2, A)$. Hence, L_1 should be ranked higher.

IV. Experimental results

In this section, we present the experimental study. The goal of our experiments is two-fold:

- (1) evaluate the effectiveness of XMLValue;
- (2) study the performance of the mining algorithm and word breaking solution.

First we begin with the description of the setup for our experiments. Next, we present and discuss our experimental results.

A. Setup

We evaluate our tool on a semi-manufactured Java web project. The test project is a simple book store management project. The project requires to implement the basic CRUD (Create, Retrieval, Update, Delete) functions. Except for the XML configuration codes, all other JSP, Java Code, CSS files are given. Hence, programmers only need to finish the coding work of XML configuration files. Table. I shows the basic statistics for the target application.

TABLE I
Statistics of the target application

Target Application	*Java Class*	*JSP*	*CSS*	*XML*
#file / #line	18/582	7/420	1/84	4/153

There are three tasks need to be accomplished. These tasks are designed for the same target applications with different difficulty. The three tasks are:

- Task 1: initializing web.xml. The functionality of this task is measured by whether the resulting application could be loaded and launched by the Tomcat server and the default page could be accessed by Internet Explorer. This is the easiest task in our evaluation.
- Task 2: constructing configuration file for MVC. Programmers need to configure six jumping points in struts-config.xml for form request of CRUD operations of the application.
- Task 3: constructing iBATIS configuration file for DAO. Programmers need to finish the dao.xml and SQLMap-config.xml of the target application.

Generally speaking, there are three typical cases when a programmer of an independent software vendor (ISV) builds applications with frameworks. 1) The programmer has no skills relating to the frameworks, but is still asked to use these frameworks to build applications due to customer requirement or urgency. In this case, the programmer tends to search for samples from Web and learns to use these frameworks to build applications from scratch. 2) Although the programmer has no framework relevant skills, the ISV will invite some instructors from professional training organizations or some experienced staff to give them enough training before they touch the code. 3) The ISV has already hired some experienced programmers for those frameworks. In order to align with real typical cases, we also organize three types of programmers and divide each type into two groups with or without the tool in the evaluation to see the benefit by the tool.

- **T1**: This group consists of 4 programmers without any experience of Struts, iBATIS and Java web application development.
- **T2**: This group accepted the training of related frameworks used in the test project.
- **T3**: We invited 4 professional programmers from IBM to form this group. They have the experience of Struts, iBATIS and Java web application development. Hence, they do not need the training about the frameworks.

For each group, we further divide them into two subgroups. One group didn't use any tools for programming and the other use XMLValue for coding assistance. All programmers were trained to understand the test project. The programmers used our tool will have an additional training about our tool. Table. II shows the grouping of all testers.

TABLE II
Groups of programmers under evaluation

Type	Group id	Experienced programmers	Training of related frameworks	XMLValue as assistant tool
T1	1	No	No	No
	2	No	No	**Yes**
T2	3	No	**Yes**	No
	4	No	**Yes**	**Yes**
T3	5	**Yes**	No	No
	6	**Yes**	No	**Yes**

Notice that all the Eclipse IDEs used by these programmers are configured with XML Editor. The XML Editor can generate XML nodes according to DTD rules. That is to say, for those without XMLValue Tool, they could also leverage DTD rule to generate sub node. However, they need to select the appropriate sub node based on their knowledge about the framework. In addition, they could not get suggestion for attribute values based on DTD rules because the XML Editor has no awareness of target application context.

B. Effectiveness

We give the results on different tasks in Table III. Table III shows the time that each tester spent on each task. '-' means that the tester failed to finish the task. Next, we give detailed analysis for each task one by one.

TABLE III
Effectiveness experiment time cost(minutes) (Each group contains two persons)

Task	Group 1		Group 2		Group 3	
	1	**2**	**3**	**4**	**5**	**6**
Task 1	180	85	6	7	37	15
Task 2	-	-	50	55	101	72
Task 3	-	-	56	50	162	125

Task	Group 4		Group 5		Group 6	
	7	**8**	**9**	**10**	**11**	**12**
Task 1	3	2	8	9	2	2
Task 2	30	42	57	65	8	13
Task 3	30	35	71	63	15	20

Task 1: From Table III we can see that all the programmers successfully finished this task. But groups using our tool

spent significantly less time than groups without our tool. For the programmers who accepted the training, those who used our tool spent less time than those without our tool, which can be observed from the comparison between Group 3 and Group 4. Even without the help of the tool, the professional programmers in Group 5 spent more time than programmers in Group 6. Above results show that the benefit brought by attribute value recommendation is quite remarkable and our tool is helpful for programmers with different degree of expertise.

Task 2: In this task, Group 1 failed to accomplish this task. In contrast, with the help of our tool Group 2 successfully finished the task. Group 3, although accepted the training about the framework, they still spent more time to finish the task than Group 2. The programmers of Group 4 achieved the best result among all inexperienced programmers, as expected. They spent less time and gained better correctness than those without the help of the tool in Group 3. Due to the benefits of our tool, the experienced programmers in Group 6 perform best. Note that the time saving was significant since Group 6 only consumed about 1/3 to 1/2 time spent by Group 5.

Task 3: For Task 3, Group 4 again did much better than programmers without the help of our tool. And as in Task 2, Group 6 finished the job in much less time than Group 5 without the assistance of the tool, because Group 5 still needed to spend time to search in the project to find the correct attribute values to finish the job.

The above results strongly suggest XMLValue is effective in improving the productivity and correctness for both inexperienced and experienced programmers. Such effectiveness of our tool is also independent on the difficulty of the tasks.

C. Performance

Next, we show the running time and accuracy of different components of XMLValue.

1) Time cost: We collect application repositories from public available codes on Web and CD attached to books about framework programming. We run our offline mining components on these repositories. The basic statistics about out dataset and the running time of our offline components are shown in Table IV. The mapping mining costs 34261ms, while the indexing costs 24182ms. Since mining and indexing are executed offline, such performances are acceptable. There are in total 138 mappings produced by our mapping mining component. Through manually checking by experienced programmers, we find that all these rules are meaningful.

TABLE IV
OFFLINE MINING STATISTICS

	Struts	***iBATIS***	***Spring***	***Hibernate***
# XML files	20	110	60	25
#Basic mappings	27	33	37	41
Mining time			34261ms	
Indexing Time			24182ms	

2) Accuracy: We give the accuracy result of our word breaker. From the repositories, we collect 500 attribute values as the data set to test the accuracy and performance of word breaking. We manually check the word breaking outcome to determine whether it is correct or not. We invite experts to evaluate whether the result of word breaking is plausible or not. The accuracy of our word breaking is 86.625%. And the average time cost is 522.52ms. In general, such an accuracy is acceptable and the time cost is tolerable.

V. CONCLUSION

In this paper, we propose a method to recommend XML attribute values and perform attribute value defect detection against current project to help programmers configure XML files better. Some key techniques such as close frequent itemset mining, word breaking, prefix tree based indexing are applied in attribute value snippet mining and defect detection. We have implemented the proposed method in a system. Evaluation results show that our tool can propose XML attribute values accurately, which significantly improves the efficiency in building framework based applications.

Acknowledgments The corresponding author is Yanghua Xiao (shawyh@fudan.edu.cn). This paper was supported by National Key Basic Research Program of China under No.2015CB358800, by the National NSFC (No.61472085, U1509213), by Shanghai Municipal Science and Technology Commission foundation key project under No.15JC1400900, by Shanghai Municipal Science and Technology project under No.16511102102, No.16JC1420401.

REFERENCES

[1] "Apache struts," http://struts.apache.org/.
[2] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *International Conference on Software Engineering*, 2005, pp. 117–125.
[3] N. Sahavechaphan and K. T. Claypool, "XSnippet: mining For sample code," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 413–430.
[4] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Automated Software Engineering*, 2007, pp. 204–213.
[5] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *European Conference on Object-Oriented Programming*, 2009, pp. 318–343.
[6] S. Huang, Y. Xiao, Y. Lu, W. Wang, and Y. Wang, "Xmlsnippet: A coding assistant for xml configuration snippet recommendation," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, july 2012, pp. 312 –321.
[7] "Eclipse xml editor," http://editorxml.sourceforge.net.
[8] "Xml editor in visual studio 2010," http://msdn.microsoft.com/en-us/library/ms255822\%28v=VS.100\%29.aspx.
[9] D. Mandelin, L. Xu, R. Bodłk, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 48–61.
[10] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *European Software Engineering Conference*, 2005, pp. 306–315.
[11] K. Wang, C. Thrasher, and B.-J. P. Hsu, "Web scale NLP: a case study on url word breaking," in *World Wide Web Conference Series*, 2011, pp. 357–366.
[12] "Microsoft n-gram service," http://research.microsoft.com/apps/pubs/?id=144355.
[13] "Acronym finder," http://www.acronymfinder.com/.
[14] G. A. Miller, "WordNet: a lexical database for English," *Communications of The ACM*, vol. 38, pp. 39–41, 1995.
[15] Z. Wu and M. S. Palmer, "Verb Semantics and Lexical Selection," in *Meeting of the Association for Computational Linguistics*, 1994, pp. 133–138.