Xeditor: Inferring and Applying XML Consistency Rules

Chengyuan Wen

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Na Meng, Chair
Eli Tilevich
Francisco Servant

Dec 11, 2019
Blacksburg, Virginia

Keywords: Empirical, software merge, conflict detection, conflict resolution

Xeditor: Inferring and Applying XML Consistency Rules

Chengyuan Wen

(ABSTRACT)

XML files are frequently used by developers when building Web applications or Java EE applications. However, maintaining XML files is challenging and time-consuming because the correct usage of XML entities is always domain-specific and rarely well documented. Also, the existing compilers and program analysis tools seldom examine XML files. In this thesis, we developed a novel approach to XML file debugging called Xeditor where we extract XML consistency rules from open-source projects and use these rules to detect XML bugs. There are two phases in Xeditor: rule inference and application. To infer rules, Xeditor mines XML-based deployment descriptors in open-source projects, extracting XML entity pairs that frequently co-exist in the same files and refer to the same string literals. Xeditor then applies association rule mining to the extracted pairs. For rule application, given a program commit, Xeditor checks whether any updated XML file violates the inferred rules; if so, Xeditor reports the violation and suggests an edit for correction?. Our evaluation shows that Xeditor inferred rules with high precision (83%). For injected XML bugs, Xeditor detected rule violations and suggested changes with 74.6% precision, 50% recall. More importantly, Xeditor identified 31 really erroneous XML updates in version history, 17 of which updates were fixed by developers in later program commits. This observation implies that by using Xeditor, developers would have avoided introducing errors when writing XML files. Finally, we compared Xeditor with a baseline approach that suggests changes based on frequently co-changed entities, and found Xeditor to outperform the baseline for both rule inference and rule application.

Xeditor: Inferring and Applying XML Consistency Rules

Chengyuan Wen

(GENERAL AUDIENCE ABSTRACT)

XML files are frequently used in Java programming and when building Web application implementation. However, it is a challenge to maintain XML files since these files should follow various domain-specific rules and the existing program analysis tools seldom check XML files. In this thesis, we introduce a new approach to XML file debugging called Xeditor that extracts XML consistency rules from open-source projects and uses these rules to detect XML bugs. To extract the rules, Xeditor first looks at working XML files and finds all the pairs of entities A and B, which entities coexist in one file and have the same value on at least one occasion. Then Xeditor will check when A occurs, what is the probability that B also occurs. If the probability is high enough, Xeditor infers a rule that A is associated with B. To apply the rule, Xeditor checks XML files with errors. If a file violates the rules that were previously inferred, Xeditor will report the violation and suggest a change. Our evaluation shows that Xeditor inferred the correct rules with high precision 83%. More importantly, Xeditor identified issues in previous versions of XML files, and many of those issues were fixed by developers in later versions. Therefore, Xeditor is able to help find and fix errors when developers write their XML files.

Dedication

To my parents

To my wife

Acknowledgments

I would like to express my gratitude to my advisor, Prof. Na Meng. This thesis would not exist without her guidance and mentorship. When I met obstacles, her advice always pointed me to the right direction of moving forward. She also helped me improve my technical-writing skills. I would also like to express my gratitude to the members of my committee, Prof. Eli Tilevich, Prof. Francisco Servant. Their advice and support are always greatly appreciated.

I thank all the members in our group. The discussions with them in the group meeting broaden my views.

I also thank my roommates, including Xiangwen Wang, Weigang Liu, Wenchao Yang, and Wei Zhao. Their presence makes life much easier.

I thank Sharon for her help and assistance throughout my graduate life here.

Finally, I would like to express my gratitude to my parents and my wife for their endless love and unconditional support.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

2 background 4

2.1 Motivation 4

2.1.1 Deployment Descriptor (DD) 4

2.1.2 XML Syntax 5

2.1.3 Problem Statement 6

3 Methodology 8

3.1 Approach 8

3.1.1 Candidate Identification 9

3.1.2 Rule Extraction 11

3.1.3 Rule-Based Checking 12

4 Results 15

4.1 Evaluation 15

4.1.1 Data Sets . . . 15
4.1.2 Metrics . . . 16
4.1.3 Effectiveness of Rule Inference . . . 17
4.1.4 Effectiveness of Rule Application . . . 21
4.1.5 Comparison with Baseline . . . 29
4.2 Related Work . . . 32
4.2.1 Validation of Metadata . . . 32
4.2.2 Program Change Prediction . . . 33
4.2.3 Configuration Debugging . . . 34

5 Discussion **36**

5.1 Lessons We Learned . . . 36
5.2 Threats to Validity . . . 38

6 Conclusions **40**

Bibliography **41**

List of Figures

1.1 Xeditor consists of two phases: Phase I infers XML consistency rules from open-source projects, and Phase II uses the rules to check for incorrect or incomplete XML edits 2

4.1 The number of bugs Xeditor detected in the 71 buggy XML trees 22

List of Tables

4.1 Rule inference with different settings of ***supp*** 18

4.2 Rule inference with different settings of ***conf*** 19

4.3 Rule inference with different settings of ***vth*** 19

4.4 Rule inference with different settings of ***pth*** 20

4.5 The 17 real XML bugs fixed by developers 25

4.6 Rules inferred by the baseline approach 28

4.7 The three real XML bugs fixed by developers 32

Chapter 1

Introduction

When building enterprise applications or Web applications on top of software frameworks (e.g., Java EE platforms), developers usually edit XML-based deployment descriptors (e.g., web.xml) to configure deployment options [16]. Misconfigured XML files can trigger abnormal runtime behaviors [41] or confusing runtime errors [17]. Debugging XML files can be challenging and time-consuming for two reasons. First, software frameworks have various domain-specific XML usage rules that are undocumented or ambiguously described [18]. Second, existing compilers and tools mainly detect and/or fix errors in source code instead of XML files.

Prior research provides limited support for checking or transforming XML files [21, 22, 30, 40]. For instance, XQuery is a language for finding elements and attributes in XML documents [22]. To find particular XML errors, developers have to use XQuery to manually describe the pattern matching mechanism. Additionally, Song and Tilevich (1) defined a domain-specific language MIL to express constraints between source code and metadata (i.e., XML or Java annotations), and (2) developed an algorithm to automatically infer such constraints from software codebases [40]. However, their approach does not handle any constraint within XML files.

This paper presents Xeditor, our new approach to infer and apply XML consistency rules based on open-source projects. Figure 1.1 shows the architecture of Xeditor. The approach consists of two phases. Given software implementation, Phase I extracts pairs of XML entities

(i.e., elements and attributes) that (i) frequently co-occur in the deployment descriptors of Web applications, and (ii) commonly refer to the same string literal at least once. For each pair, Xeditor further conducts association rule mining [47] to decide (1) whether the two entities have any association relationship, and (2) whether they always refer to the same value. If two entities are associated with each other, Xeditor infers an **co-existence** rule of the format "$A \Rightarrow B$", meaning that "*if A exists, B should also exist*". Furthermore, if two associated entities always refer to the same value, Xeditor infers an additional **same-value** property, obtaining a rule like "$A \Rightarrow B$, *same-value*".

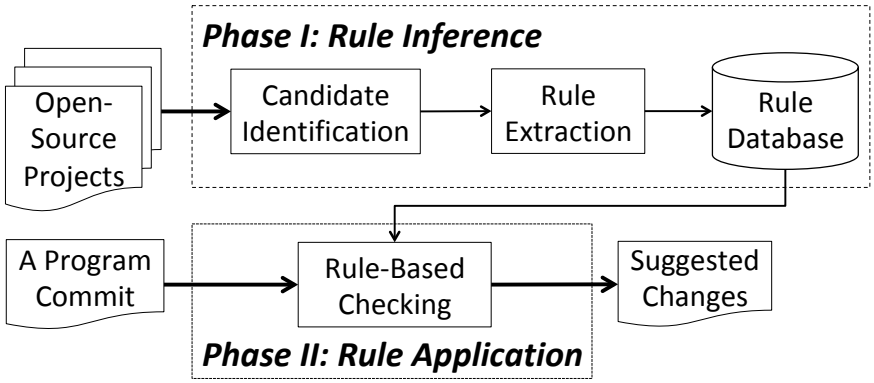

Figure 1.1: Xeditor consists of two phases: Phase I infers XML consistency rules from open-source projects, and Phase II uses the rules to check for incorrect or incomplete XML edits

Phase II takes in a new program commit submitted by developers and checks whether there is any XML file change. For every changed XML file Xeditor tentatively matches the new version of the file against all extracted rules. If there is a rule for which the XML file contains A without B, Xeditor recommends developers to insert B or delete A; furthermore, if the same-value property of any rule is violated, Xeditor also suggests the value correspondence between XML entities. In the Continuous Integration (CI) practices [24], we envision Xeditor to be used for correctness checking before a submitted commit is integrated into the software product. In this way, Xeditor helps developers correctly edit XML files and complement existing code-oriented program analysis techniques.

For evaluation, we applied Xeditor to the deployment descriptors of 1,137 projects. Xeditor inferred 48 rules, among which 40 rules are true positives. It means that our approach can infer rules with high precision (83%). Furthermore, we constructed a data set of 71 injected XML bugs to evaluate Xeditor's capability of rule application. Our experiment shows that Xeditor detected all injected bugs with high precision (100%), high recall (100%), and high accuracy (100%). More importantly, we applied Xeditor to the program commits in another 478 open-source projects, and detected 31 incorrectly updated XML files, 17 of which were later fixed by developers. This implies that Xeditor can help developers avoid introducing bugs when they modify XML files. Finally, we compared Xeditor with a baseline approach that infers rules from co-changed entities, and applied both approaches to the same software repositories. Our comparison shows that Xeditor works much better than the baseline by inferring rules with higher precision (83% vs. 43%) and revealing more real bugs.

In summary, this paper makes the following contributions:

- We developed a novel approach—Xeditor—to automatically infer and apply domain-specific XML consistency rules. Different from most prior work, Xeditor does not need users to prescribe any matching template or change pattern.
- We built Xeditor to infer rules from the co-existence of XML entities. Compared with a baseline technique that infers rules from co-changed entities, Xeditor worked better by identifying more rules with higher precision.
- We conducted a comprehensive evaluation on Xeditor. Our evaluation shows that (1) Xeditor could infer important rules because developers did make mistakes by ignoring such delicate constraints, and (2) Xeditor suggested useful corrective changes for buggy XML files.

Chapter 2

background

2.1 Motivation

This section first introduces deployment descriptors (Section 2.1.1) and XML syntax (Section 2.1.2). It then explains why it is challenging to configure deployment descriptors appropriately (Section 2.1.3).

2.1.1 Deployment Descriptor (DD)

A deployment descriptor is a configuration file that specifies how an artifact should be deployed. For instance, in a Web application *App* written in Java, the DD (e.g., web.xml) describes the component classes, resources, and configurations of *App*; it also specifies how a Web server uses these components to serve Web requests [11]. Similarly, in a Java EE application, the DD (e.g., application.xml) clarifies the configuration, container option, and security settings [12]. Generally speaking, XML is used for the syntax of DD files. Depending on the types of applications and modules, DDs may be located in various file folders and named divergently.

Listing 2.1: A simplified version of a web.xml file [4]

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="2.5" xmlns="http://java.sun.com/..."
```

```
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://java.sun.com/xml/ns/..." >
5     ...
6    <security-constraint>
7       ...
8       <auth-constraint>
9         <role-name>comm</role-name>
10      </auth-constraint>
11   </security-constraint>
12    ...
13   <security-role>
14      <role-name>comm</role-name>
15   </security-role>
16 </web-app>
```

2.1.2 XML Syntax

XML syntax defines how an XML file can be written [15]. According to the syntax rules, each XML file should include one or more **XML elements**, which elements are organized as a tree structure. Namely, there is only one **root element** in any XML file, while the root element can have one or more **child elements**. For the exemplar XML file shown in Listing 2.1, the `<web-app>` element is root; one of its child element is `<security-constraint>`.

Generally speaking, an XML document consists of **markup** and **data**. Markup is provided in the form of **tags** and **attributes**. Data is the text that goes in between the tags or is provided within their attributes. XML elements are represented by tags. An element usually consists of an opening tag (e.g., "`<role-name>`"), a closing tag (e.g., "`</role-name>`"), and data between the tags (e.g., "`comm`"); but it can also include just one tag (e.g., "`<tag/>`"). Attributes

can be added to XML elements to provide more information; they are represented as **name-value pairs** (e.g., "`version="2.5"`"). Here, values must be surrounded by double quotes. To facilitate discussion, this paper uses **XML entities** to refer to both XML elements and attributes.

2.1.3 Problem Statement

In deployment descriptors, developers need to declare specific XML entities in certain ways for their deployment requirements. For instance, as demonstrated in Listing 2.1, to define the access privileges to a collection of resources in a Web application, developers must specify a `<security-constraint>` element. Because this element is nested within the `<web-app>` element, the `<web-app>` element must always be included simultaneously, and it must indicate the version of the web application schema (2.4 or 2.5) [6]. Furthermore, to grant a group of users the defined access privileges, developers have to specify the `<security-role>` element, with its child `<role-name>` referring to a role name defined in the same document (e.g., "`comm`") [14]. When the data value of element `<role-name>` under `<auth-constraint>` is updated, the data value of element `<role-name>` under `<security-role>` should be also updated consistently.

It is always challenging and time-consuming to correctly define and maintain DDs. According to a recent study on StackOverflow, many developers ask various questions on how to correctly configure DDs and expressed frustrations with XML debugging [34]. This is mainly because the XML entity usage often varies with the type of applications being deployed and developers' deployment needs. Unfortunately, such delicate and complex domain-specific knowledge is rarely well documented in online tutorials of software platforms (e.g., Java EE) and application servers (e.g., WildFly [13]). This paper intends to explore an approach to automatically (1) infer the correct XML entity usage from open-source projects, and (2)

leverage the inferred knowledge to help developers debug erroneous XML files.

For this paper, we mainly focus on the **co-existence constraints** between XML entities. Our hypothesis is that when multiple XML entities are implicitly related, it is possible that developers use some of the entities but forget to always use the others simultaneously. Therefore, if an XML entity A exists in a DD, we were curious (1) whether there is any other XML entity B that must exist in the same file for completeness and correctness, and (2) whether developers actually ignored B when using A.

Chapter 3

Methodology

3.1 Approach

To explore the co-existence constraints between XML entities, we built a novel approach—Xeditor. As shown in Figure 1.1, Xeditor has two phases. This section first summarizes the steps in each phase and then describes each step in detail (Section 3.1.1-Section 3.1.3).

Phase I: Rule Inference

- Given a set of open-source projects, Xeditor identifies a candidate set of deployment descriptors $F = \{f_1, f_2, \ldots, f_m\}$, from which files Xeditor further extracts candidate XML entity pairs $C = \{c_1, c_2, \ldots, c_n\}$.
- For each candidate $c_i = (e_{i1}, e_{i2})$, Xeditor searches among all files F to find the occurrence of each entity; it further applies association rule mining to infer any co-existence relationship between the entities. Each mined rule is represented in the format of "$A \Rightarrow B$" or "$A \Rightarrow B$, *same-value*".

Phase II: Rule Application

- Given a program commit, Xeditor searches for any updated XML file f and examines the file's new version f_{new}. For each co-existence rule r, Xeditor checks whether A and

B co-exist in f_{new}. If A exists but B does not, Xeditor reports a bug. For any rule with the same-value property, Xeditor also examines the values of A and B when possible.

3.1.1 Candidate Identification

Given a set of open-source projects, we need to first identify the DDs. Although the DDs of different Java projects are all XML files, not every XML file is a DD. To efficiently locate candidate DDs among the available XML files, we leveraged a heuristic to extract the XML files that (1) are within the folder `WEB-INF`, or (2) contain any of the following keywords in its file path: "spring", "security", and "web". We defined this heuristic because based on our experience, DDs usually exist in specific folders.

For each located XML file, we applied GumTree [26] to generate a tree, where nodes represent XML entities or data and edges represent the parent-child containment relationship. To identify candidate pairs in the tree representation, a naïve approach can

- identify all XML entities $E = \{e_1, e_2, \ldots, e_n\}$, and
- create a pair out of any two identified entities (i.e., (e_i, e_j), where $i, j \in [1, n]$ and $i \neq j$).

However, since many entities are included into a DD for separate requirements, they are essentially irrelevant and the co-existence is accidental. Consequently, many of the candidate pairs generated by the above-mentioned naïve approach are actually useless or even misleading for constraint inference.

To overcome the challenge of noisy entity pairs and to identify promising candidates, we used a heuristic that "*if two XML entities refer to the same data value at least once, they are likely to be correlated*". As shown in Listing 2.1, the `<role-name>` element under

`<security-constraint>` refers to the data "`comm`" (line 9), while the `<role-name>` element under `<security-role>` refers to the same string literal (line 14). Therefore, Xeditor identifies a candidate pair based on the two elements. We noticed that in some scenarios, the common data shared between entities is as trivial as a digital number or a boolean constant (e.g., "`true`"). Such trivial data items can be so popularly used in various XML entities that the data usage does not necessarily imply any correlation between the container entities. Therefore, to further avoid unpromising pairs, Xeditor creates a candidate pair only if two entities commonly refer to the same non-trivial string literal.

Before generalizing rules from the concrete candidate pairs, we need to solve another challenge: how can we represent candidates in a non-ambiguous way? In Listing 2.1, the two elements referring to "`comm`" have the same tag `<role-name>`. If we simply use these tags to define a candidate pair `(<role-name>, <role-name>)`, the semantics is very confusing and we cannot tell the elements apart. To solve this problem, we decided to include the **context**, i.e., the fully qualified paths of both entities, into our representation for disambiguation. Suppose that an entity A has its parent entity as P, while P is contained by the root element R. Then our *context-aware* representation for A is: R_P_A, which corresponds to the XML path from R to A. Thus, for the candidate pair mentioned above, our context-aware representation is:

```
(web-app_security-constraint_auth-constraint_role-name,

 web-app_security-role_role-name).
```

By including the path information, our context-aware representation of entity pairs can (1) differentiate between same-tag entities, and (2) help with later processing (Section 3.1.2-Section 3.1.3).

3.1.2 Rule Extraction

After extracting a set of candidate pairs (i.e., $C = \{c_1, c_2, \ldots, c_m\}$) from candidate XML files F, Xeditor infers rules with **association rule mining (ARM)** [47]—a classical way to find patterns in data and detect couplings between data entities. An **assocation rule** between two entities e_1 and e_2 can have the format "$e_1 \Rightarrow e_2$" or "$e_2 \Rightarrow e_1$". In the notation "$e_1 \Rightarrow e_2$", e_1 is called the *antecedent*, and e_2 is called the *consequent*. The notation means that the occurrence of e_1 implies that of e_2. With such rules, we can predict the occurrence of e_2 whenever e_1 occurs.

ARM mines association rules in a probabilistic way. Intuitively, ARM infers the rule "$e_1 \Rightarrow e_2$" if the two entities co-occur for a sufficient number of times and whenever e_1 occurs, e_2 usually occurs. Formally, given the number of occurrences of entity e_1 (i.e., $freq(e_1)$) and that of entity e_2 (i.e., $freq(e_2)$), we represent the number of co-occurrences between the entities as $freq(e_1, e_2)$. The rule "$e_1 \Rightarrow e_2$" is derived if

1. $freq(e_1, e_2) \geq supp$, where $supp$ is the threshold for the number of co-occurrences, and
2. $Pr(e_2|e_1) = \dfrac{freq(e_1, e_2)}{freq(e_1)} \geq conf$, where $conf$ is the threshold for the probability.

In our research, for each candidate entity pair $c_i = (e_{i1}, e_{i2})$, Xeditor identifies the occurrences of e_{i1} and e_{i2} in all files; it then computes $freq(e_{i1}, e_{i2})$, $Pr(e_{i2}|e_{i1})$, and $Pr(e_{i1}|e_{i2})$ accordingly. To avoid the bias towards any project-specific rule, we also defined the following formula:

(3) $pfreq(e_1, e_2) \geq pth$, where pth is the minimum number of projects that support the co-occurrences.

We believe that the more projects supporting certain co-occurrences, the more likely that

such co-occurrences imply authentic and project-agnostic rules. Additionally, because associated entities are likely to refer to the same value, we also defined and used the following formula to ensure the quality of inferred rules:

(4) $Pr(same\_value) = \dfrac{freq(value(e_{i1}) = value(e_{i2}))}{freq(e_{i1}, e_{i2})} \geq vth$, where vth is the minimum rate of same-valued co-occurrences.

By default, Xeditor extracts the rule "$e_{i1} \Rightarrow e_{i2}$" if $supp = 8$, $conf = 0.9$, $vth = 0.9$, and $pth = 3$. We used these threshold values because our evaluation shows that Xeditor works most effectively with this setting (Section 4.1.3).

In a rule "$e_{i1} \Rightarrow e_{i2}$", if both entities hold the same values in all of their co-occurrences (i.e., $Pr(same\_value) = 1.0$), Xeditor attaches an extra same-value property to the rule, deriving "$e_{i1} \Rightarrow e_{i2}$, *same-value*". Actually in our research, to reveal more rules with the same-value property, we also examined the inferred rules with $Pr(same\_value) < 1.0$, and manually added that property when our examination showed that the property should hold.

3.1.3 Rule-Based Checking

Given a program commit in an open-source project, Xeditor uses the approach mentioned in Section 3.1.1 to check whether any updated XML file is a DD. If such an XML file is updated from f_{old} to f_{new}, Xeditor enumerates the inferred rules to detect bugs in f_{new} and suggests changes when possible. For instance, from the file shown in Listing 2.1 and several other files, Xeditor inferred the following rule:

`web-app_security-constraint_auth-constraint_role-name` $\Rightarrow$

`web-app_security-role_role-name`, *same-value*.

We applied Xeditor to the XML file shown in Listing 3.1, which file was produced by a program commit in the project bagh [3]. Xeditor found two matches for the antecedent `<role-name>` element under `<security-constraint>`, but found no match for the consequent `<role-name>` element. In this scenario, Xeditor reports two bugs and suggested the corresponding fixes as below:

"*A <role-name> entity should be inserted at <web-app><security-role><role-name>, with the data value 'prof'; or the <role-name> entity at <web-app><security-constraint><auth-constraint> with value 'prof' should be removed.*

A <role-name> entity should be inserted at <web-app> <security-role> <role-name>, with the data value 'stu'; or the <role-name> entity at <web-app><security-constraint><auth-constraint> with value 'stu' should be removed. "

Listing 3.1: A simplified version of another web.xml file [3]

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="2.5" xmlns="http://java.sun.com/..."
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/..." >
5    ...
6    <security-constraint>
7      ...
8      <auth-constraint>
9        <role-name>prof</role-name>
10     </auth-constraint>
11   </security-constraint>
12   ...
13   <security-constraint>
14     ...
15     <auth-constraint>
```

```
16          <role-name>stu</role-name>
17        </auth-constraint>
18    </security-constraint>
19 </web-app>
```

Chapter 4

Results

4.1 Evaluation

This section first introduces our data sets (Section 4.1.1) and evaluation metrics (Section 4.1.2). Next, it presents our evaluation on Xeditor's effectiveness of rule inference (Section 4.1.3) and rule application (Section 4.1.4). Finally, it expounds on the comparison between Xeditor and a baseline technique (Section 4.1.5).

4.1.1 Data Sets

We constructed two major data sets for evaluation. The first set (**D1**) contains 1,137 open-source projects, from which Xeditor identified 4,248 DDs. We mainly leveraged this data set to investigate three research questions:

RQ1: How effectively does Xeditor infer rules based on the co-existence of XML entities?

RQ2: How effectively does Xeditor apply rules to locate bugs and suggest corrective changes?

RQ3: How well does Xeditor compare with a baseline approach that infers rules from frequently co-changed entities?

The second data set (**D2**) contains another 478 open-source projects. From these projects,

Xeditor identified 5,811 versions of updated XML files, and leveraged all rules inferred from the first data set to find bugs. This data set is similarly used as the above-mentioned data for our investigation of RQ2. However, it also enables us to explore the following research questions:

RQ4: Do developers introduce any of the XML bugs that Xeditor can detect during XML file maintenance?

RQ5: How well does Xeditor's change suggestion match developers' actual fixes for those detected XML bugs?

RQ6: How well does Xeditor compare with the baseline in terms of rule application and change suggestion?

We mined the projects in both data sets on GitHub [5] using the heuristics described in Section 3.1.1. Namely, we crawled the website for any project that contains at least one XML file in specific folders, such as `WEB-INF/` and `web/`. We randomly put the crawled projects into the two data sets, such that Xeditor inferred rules from 70% of the projects, and applied rules to the other 30% of projects.

4.1.2 Metrics

We used the following metrics for evaluation.

Precision (P) measures among all reports generated by Xeditor, how many of them are true positives:

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of generated reports}} \times 100\%. \tag{4.1}$$

Precision can be used to evaluate the effectiveness of rule inference and rule application. For

rule inference, P measures how many reported rules are correct rules. For rule application, P measures among all reported XML bugs, how many of them are real bugs.

Recall (R) measures among all known true positives, how many of them are reported by Xeditor:

$$R = \frac{\text{\# of correct reports}}{\text{Total \# of true positives}} \times 100\%. \tag{4.2}$$

Since we do not have any prior knowledge on the true rules existing in XML files, we did not evaluate the recall of rule inference. However, based on the inferred rules and our manual inspection, we managed to construct a ground truth data set and evaluated the recall of rule application.

F score (F) combines P and R to measure the accuracy:

$$F = \frac{2 \times P \times R}{P + R} \times 100\%. \tag{4.3}$$

The F score measures the trade-off between precision and recall, thus we leveraged it to measure the accuracy of rule application.

4.1.3 Effectiveness of Rule Inference

There are four parameters in Xeditor that can influence its capability of rule inference: $supp$, $conf$, vth, and pth.

To determine the best configuration for each parameter, we enumerated multiple value settings, applied Xeditor to the first data set D1 with each setting, and then manually inspected the inferred rules to calculate precision. Before our manual inspection, we did not have any prior knowledge of the correct rules. Therefore, given an inferred rule, our manual inspection involved lots of online search for (1) the related library specifications or framework tutorials,

(2) XML examples that violate the potential rule but still get recommended as correct usage, or (3) relevant discussions on technical forums (e.g., StackOverflow [10]). We considered a rule to be correct if online documentation or discussion recommends it, or no XML example violates the rule.

Sensitivity to *supp*.

We explored three settings for the threshold of supporting instances *supp*. As shown in Table 4.1, when *supp* decreases from 9 to 7, Xeditor infers more rules and reveals more correct rules. This is understandable because with the other parameters unchanged, the fewer supporting instances are required, the more likely that Xeditor reveals correct rules while obtaining noise in its reports. Since our manual inspection to validate inferred rules is very time-consuming, we did not explore more settings for this parameter. However, we still observed a precision peak within these three settings. Therefore, we set $supp = 8$ by default.

Table 4.1: Rule inference with different settings of ***supp***

supp	**conf**	**vth**	**pth**	**# of Inferred Rules**	**# of Correct Rules**	**Precision (%)**
9	0.9	0.9	3	45	37	82
8	0.9	0.9	3	48	40	83
7	0.9	0.9	3	53	42	79

Sensitivity to conf.

We explored three settings for the threshold of confidence value $conf$. As shown in Table 4.2, when $conf$ decreases from 1 to 0.8, more correct rules are detected, while the precision increases first and then decreases. Thus, we set $conf = 0.9$.

Table 4.2: Rule inference with different settings of ***conf***

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
8	1	0.9	3	33	26	79
8	0.9	0.9	3	48	40	83
8	0.8	0.9	3	51	41	80

Sensitivity to vth**.**

We explored five settings for the threshold of same-valued co-occurrences vth. In Table 4.3, as vth changes from 1.0 to 0.6 with 0.1 stepwise decrements, the number of inferred rules increases significantly. Meanwhile, the inference precision decreases. Our observation shows that the number of same-valued co-occurrences can effectively indicate whether two co-occurring entities are related. When the threshold vth is too low (e.g., $vth \leq 0.6$), Xeditor can report many false rules.

Table 4.3: Rule inference with different settings of ***vth***

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
8	0.9	1.0	3	32	27	84
8	0.9	0.9	3	48	40	83
8	0.9	0.8	3	56	45	80
8	0.9	0.7	3	68	54	79
8	0.9	0.6	3	78	61	78

By default, we set $vth = 0.9$. Notice that we did not pick the setting $vth = 1.0$, which achieves the highest precision among all our explorations. This is because in addition to the high precision of rule inference, we also want to ensure that Xeditor can identify as many true rules as possible. When more true rules are identified, Xeditor is more likely to report real XML bugs in the rule application phase. We believe that $vth = 0.9$ achieves a good balance between precision and the number of correct rules identified.

Table 4.4: Rule inference with different settings of ***pth***

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
8	0.9	0.9	1	59	42	71
8	0.9	0.9	2	53	41	77
8	0.9	0.9	3	48	40	83
8	0.9	0.9	4	44	37	84

Sensitivity to *pth*.

We explored four settings for *pth*. In Table 4.4, as *pth* increases from 1 to 4, fewer rules are reported and fewer correct rules are revealed. When $pth = 1$, precision is 71%, much lower than the precision values produced by other settings. Two possible reasons can explain the observed trend. First, when the co-occurrences of certain entities only exist in one project or one XML file, such co-occurrences may be specific to the particular project or the coding styles of that project's owners. Consequently, many rules inferred in this way can be false positives. Second, when multiple projects support the co-occurrences of certain entities, the rule inference from these projects is more likely to avoid the bias introduced by a particular project or a few developers. Therefore, the precision rate is almost the same as *pth* increases from 3 to 4. By default, we set $pth = 3$.

Furthermore, among the 40 correct rules revealed by our default setting, there are 11 rules that have the same-value property. Based on our manual inspection for different parameters, we identified in total 71 correct rules, 16 of which have the same-value property.

Finding 1: *For rule inference, we configured* Xeditor *with* $supp = 8$, $conf = 0.9$, $vth = 0.9$, $pth = 3$. *This is because our sensitivity experiments show that with this configuration,* Xeditor *can reveal a decent number of true rules (i.e., 40) with reasonably good precision (i.e., 83%).*

4.1.4 Effectiveness of Rule Application

With the 71 true association rules inferred, we performed two experiments to evaluate Xeditor's effectiveness of rule application. The two controlled experiment assesses how well Xeditor detects and fixes manually injected bugs, while the uncontrolled one evaluates how well Xeditor detects and fixes real bugs.

First Controlled Experiment.

To assess whether Xeditor can apply rules as expected to detect XML bugs and provide suggestions, we constructed a set of manually injected bugs S. When we define the rule class, there is a field recording which files support this rule. Specifically among the 71 correct rules, We back track the supported files for each rule and randomly picked 10% files (597 files) f supporting the rules, generated a tree representation for f using GumTree, removed the entity B intentionally, and put the rest of the tree—t—into S. Some of the support files are duplicated but support different rules. The other 90% files go into the training set. We then extract rules from the training set and apply the rules detect bugs in the test set. Ideally, if Xeditor works as expected, it should be able to locate the injected bug in t inside S and provide suggestion accordingly. This is a controlled experiment because by manually removing certain entities from 71 correct XML files, we built a ground truth set of XML bugs, which can be used to evaluate Xeditor's precision, recall, and accuracy for rule application.

From the training set, we mined all the 71 true rules. Interestingly, Xeditor detected 991 bugs in the 597 buggy XML trees, revealing more bugs than the ground truth. As shown in Figure 4.1, Xeditor detected single bugs—the oracle bugs only—in 439 trees. In another 106 trees, Xeditor detected 2 bugs in each tree. There is one buggy tree in which Xeditor

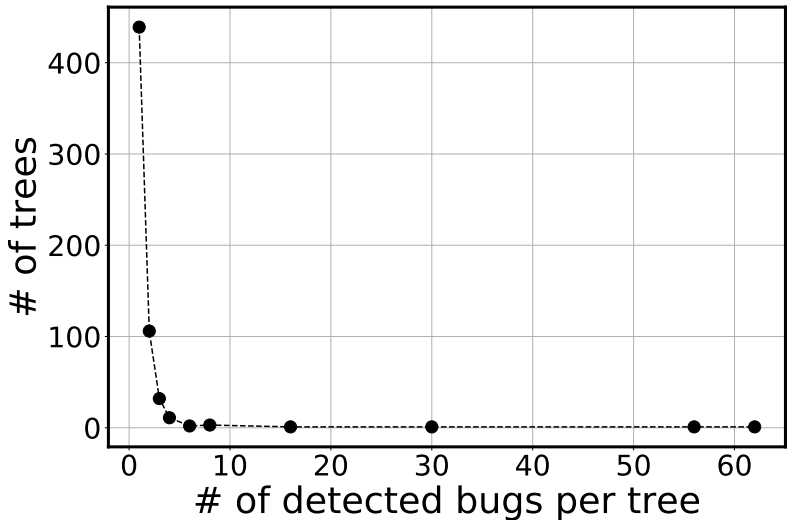

Figure 4.1: The number of bugs Xeditor detected in the 71 buggy XML trees

reported sixty two bugs. We manually compared the reported bugs and suggested fixes against the original 71 correct files, finding all 991 bugs to be correctly reported. The first reason is when I remove the entity B, I remove all the entities B in the file. Another reason is because some association rules share the same consequents but have different antecedents, as illustrated below:

"`beans_sec:authentication-manager_sec:authentication-provider_`

`sec:password-encoder_ref` $\Rightarrow$ `beans_bean_id`" vs.

"`beans_sec:http_sec:custom-filter_ref` $\Rightarrow$ `beans_bean_id`."

When both antecedents exist in a buggy tree, Xeditor can apply both rules to identify the same missing entity and provide suggestion accordingly. Finally, the reported bugs cover the 71 injected bugs in our data set S.

Second controlled Experiment

To evaluate how Xeditor performs in real environment, we split data set D1 to 10 folds. 9 folds of the files are combined together to be the training set to infer rules. Default parameters shown in Sec. 4.1.3 are used for the rule inferring. The left fold of files are converted to buggy XML trees and used as test data. The inferred rules applied to the buggy XML trees and provide bugs. The precision and recall can be calculated since we know the ground truth. This experiment is done 10 times due to each fold could be treated as test data once. The averaged precision is 74.6%, and the averaged recall is 50.8%.

Finding 2: *Knowing all correct rules and manually injected buggy files,* Xeditor *reported bugs with 100% precision, 100% recall, and 100% accuracy. It means that* Xeditor *can correctly detect bugs when an erroneous XML file violates any rule* Xeditor *inferred. In the real environment,* Xeditor *inferred rules with default parameters, and report bugs with* 74.6% *precision and* 50% *recall. It means that* Xeditor *can effectively infer rules and detect bugs*

An Uncontrolled Experiment.

To evaluate whether Xeditor can detect any real bug in updated XML files, we further applied Xeditor to the second data set D2. Specifically for each project, if a commit C in the version history modifies a deployment descriptor f, Xeditor scans the after-change version of f. If there is any reported rule violation, our manual inspection further examines versions after C to validate the tool's output.

In total, Xeditor reported 205 unique bugs, 31 of which are real bugs and 17 real bugs were fixed by developers in later commits. This observation indicates that developers did make

the mistakes that can be captured by Xeditor. For the 14 remaining real bugs that were not fixed in software version history, we also sent developers emails to ask for their opinion on the located bugs and suggested fixes. Up till this paper submission, we have not received much feedback from those developers.

Finding 3: Xeditor *located 17 real XML bugs that were later fixed by developers. It means that developers did introduce the XML bugs that* Xeditor *can detect, showing the usefulness of* Xeditor*'s bug detection capability.*

Given a rule "$A \Rightarrow B$", if an XML file only contains A without B, Xeditor can suggest two potential fixes: (1) inserting B or (2) removing A. We further examined the bug fixes applied by developers to investigate how well Xeditor's suggestions match developers' actual fixes. In Table 4.5, column **Bug Index** shows the index we assigned to each revealed bug, column **Fixing Strategy** describes developers' strategies to resolve those bugs, and **# of Commits in-between** measures the number of commits standing between the buggy version and fixed version.

As shown in Table 4.5, eight bugs (i.e., 4^{th}, 5^{th}, 8^{th}, 11^{th}, 12^{th}, 17^{th}) were fixed via the insertion of B. Developers fixed another six bugs (i.e., 6^{th}, 7^{th}, 9^{th}, 13^{th}, 14^{th}, and 15^{th}) by deleting A. Four bugs (i.e., 2^{nd}, 3^{rd}, 10^{th}, and 16^{th}) were fixed via updates to B. One bug (i.e., 1^{st}) was fixed when developers imported another XML file for the definition of A. Among all these 17 fixes applied by developers, 14 fixes correspond to the candidate solutions suggested by Xeditor. It means that Xeditor usually suggests helpful fixes.

Additionally, for 15 out of the 17 bugs introduced by a certain program commit, developers applied fixes in the immediately next commit. Among the remaining 4 bugs, developers fixed 2 bugs (i.e., 8^{th} and 17^{th}) after 10 commits, fixed 1 bug (i.e., 9^{th}) after 4 commits, and fixed 1 bug (i.e., 13^{th}) after 2 commits. These observations imply that if developers

Table 4.5: The 17 real XML bugs fixed by developers

Bug Index	**Fixing Strategy**	**# of Commits in-between**
1	Import an XML file	1
2	Update the incorrect string literal for B	1
3	Update the incorrect string literal for B	1
4	Insert B	1
5	Insert B	1
6	Delete A	1
7	Delete A	1
8	Insert B	10
9	Delete A	4
10	Update the incorrect string literal for A	1
11	Insert B	1
12	Insert B	1
13	Delete A	2
14	Delete A	1
15	Delete A	1
16	Update the incorrect XML tag in B	1
17	Insert B	10

had used Xeditor to examine their updated XML files before committing program changes, they should have avoided checking in the erroneous program changes, or even have fixed the introduced bugs earlier.

Finding 4: *Developers fixed 14 out of the 17 real bugs via inserting or deleting entities. The suggestions provided by* Xeditor *cover all these 14 fixes, showing great potential of helping developers correctly maintain XML files.*

There are 174 false positives reported by Xeditor for 3 reasons.

1. Unknown rules There are 141 false positives reported because the actual co-existence rules applied in XML files are not included in Xeditor's rule set. For instance, in Spring, a bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container [9]. A defined bean can refer to another bean as its property, and Xeditor infers a relevant rule as below:

`beans_bean_property_ref` ⇒ `beans_bean_id`, *same-value*,

With this rule, given the XML file shown in Listing 4.1, Xeditor locates a reference to a bean `contactRepository` but does not recognize the corresponding bean definition. Thus, Xeditor suggests that a bean entity should be inserted under `<beans>` to have the id as "`contactRepository`". Nevertheless, developers actually defined a different entity `<axon:event-sourcing-respos` with that id. Since Xeditor is unaware of such less popular data correspondence between `<beans><bean><property><ref>` and `<beans><axon:event-sourcing- repository><id>`, its suggestion is not applicable to the scenario.

Listing 4.1: A file follows a rule unknown to Xeditor [1]

```
1 ...
2 <beans xmlns="http://www.springframework.org/...>
3   <bean class="nl.envotation.addressbook...">
4     <property name="contactRepository" ref="contactRepository"/>
5   </bean>
6   <axon:event-sourcing-repository id="contactRepository" ...>
7     <axon:snapshotter-trigger .../>
8   </axon:event-sourcing-repository>
9 </beans>
```

In particular, 108 of these false positives are generated for the updates of `context-idgen.xml` in project `lushtext` [7]. It means that when a developer defines DDs quite differently from most other developers, Xeditor is less likely to provide useful suggestions. This is understandable

because Xeditor infers rules based on probabilities and the frequent occurrences of certain entities; it does not help when some entities are uniquely used in a few XML files. Xeditor shares this limitation with many probability-based rule/specification mining tools [27, 40, 44, 46, 49, 50].

2. Java-based entity association. There are 27 false alarms generated because related entities are defined in separate XML files and a Java file declares the file-level connection. As shown in Listing 4.2, the first XML file declares a bean to refer to another bean `dataSource`, the second XML file declares the `dataSource` bean, and a Java file annotates that both XML files should be considered for context configuration. However, because Xeditor currently analyzes each XML file independently, it does not relate different XML files or extract any XML relationship from Java files.

Listing 4.2: Two XML files are connected by a Java file [2]

```
1  // In eventstore-jpa-test-context.xml
2  ...
3  <beans ...>
4    <bean id=...>
5      ...
6      <property name="dataSource" ref="dataSource"/>
7    </bean>
8  </beans>
9  // In db-context.xml
10 ...
11 <beans ...>
12   ...
13   <bean id="dataSource" class=org.springframework...>
14     ...
15   </bean>
```

```
16 </beans>
17 // In JpaEventStoreTest.java
18 ...
19 @ContextConfiguration(locations={"classpath:/META-INF/spring/db-context.xml",
       "classpath:/META-INF/spring/eventstore-jpa-test-context.xml
20 })
21 public class JpaEventStoreTest {...}
```

Table 4.6: Rules inferred by the baseline approach

Rule Id	Content	Any corresponding rule by Xeditor?
1	`Change(b:property_ref)` ⇒ `Change(b:property_name)`, *same-value*	`b:property_ref` ⇒ `b:property_name`, *same-value*
2	`Change(aop:config_aop:advisor_pointcut-ref)` ⇒ `Change(aop:config_aop:pointcut_id)`, *same-value*	`aop:config_aop:advisor_pointcut-ref` ⇒ `aop:config_aop:pointcut_id`, *same-value*
3	`Change(web-app_servlet_servlet-name)` ⇒ `Change(web-app_servlet-mapping_servlet-name)`, *same-value*	`web-app_servlet_servlet-name` ⇒ `web-app_servlet-mapping_servlet-name`, *same-value*

3. Import-based entity association. In six scenarios, Xeditor reports missing entities although those entities are actually declared in imported XML files. As shown in Listing 4.3, the first XML file declares a bean with the id "`connectionParameterDBCHD`", while the second XML file imports the first XML file to refer to the defined bean. Because Xeditor currently processes each file independently, it does not model any relationship between files.

Listing 4.3: An XML file imports another XML file to refer to any entity defined by the latter [2]

```
1 // In connections.xml
2 ...
3 <beans xmlns="http://www.springframework.org/...
```

```
4    ...
5    <bean id="connectionParametersDBCHD" class=...>
6    ...
7    </bean>
8  </beans>
9  // In fpApplicationContext.xml
10 ...
11 <beans xmlns="http://www.springframework.org/...
12   <import resource="chd/connections.xml"/>
13   ...
14   <bean id=...>
15     ...
16     <property name="connectionParameters" ref="connectionParametersDBCHD" />
17   </bean>
18 </beans>
```

Finding 5: Xeditor *produced false positives mainly because (1) its rule set did not cover all co-existence rules, and (2) it does not model any connection between related XML files.*

4.1.5 Comparison with Baseline

Prior change suggestion tools mine software version history for co-change patterns, and use those patterns to identify any missing change [27, 28, 32, 37, 45, 50]. For instance, ROSE leverages association rule mining (ARM) to identify the co-change rules between program entities (e.g., "*if method A is changed, method B should also be changed*"), such that whenever an entity (e.g., A) is changed and related entities (e.g., B) are not, ROSE suggests the missing change(s) [50]. These tools are similar to Xeditor due to their adoption of ARM, but different

by relying on the *co-changes* instead of *co-existence* of program entities. We were curious how Xeditor compares with prior work, but no prior work extracts any project-agnostic co-change rules for deployment descriptors.

The Baseline Approach (Baseline). To facilitate the comparison between the two methodologies, i.e., co-existence-based vs. co-change-based, we built a ROSE-like baseline approach (named "Baseline" for short). Baseline mines frequently co-changed XML entities in software version history and uses ARM to infer association rules that have the format "$Change(A) \Rightarrow Change(B)$" or "$Change(A) \Rightarrow Change(B)$, *same-value*". Baseline then exploits these rules to check individual program commits (i.e., the program changes) for any erroneous XML update. Similar to Xeditor, Baseline extracts two entities as a candidate pair if (i) they were frequently co-changed within the same files and (ii) they refer to the same non-trivial string literal at least once. Baseline shares the same configuration with Xeditor: $supp = 8$, $conf = 0.9$, $vth = 0.9$, and $pth = 3$.

Comparison for Rule Inference. To facilitate fair comparison, in our experiment, we applied Baseline to the software repositories corresponding to D1 for rule inference, and applied it to D2 for XML bug detection and change suggestion. For rule inference, Baseline only reported 7 rules, while only 3 of these rules are true rules. Thus, the inference precision of Baseline is 43%, much lower than that of Xeditor (i.e., 83%). Two possible reasons can explain the low number of true rules revealed by Baseline. First, the tool infers rules from co-changed XML entities in version history. If two entities always co-occur but seldom get changed, Baseline cannot infer any association relationship between the entities. Second, co-changed entities are not necessarily related. When certain entities are accidentally co-changed in multiple commits, Baseline can infer false rules. Table 4.6 lists the three correctly inferred rules. Xeditor correctly inferred 40 rules, among which three rules correspond to the true rules inferred by Baseline. It means that our approach is more effective than Baseline.

Finding 6: *For rule inference,* Xeditor *is more effective than* Baseline *by inferring more rules with higher precision. This indicates that our rule mining based on co-existence is more effective than existing tools that are based on co-changes.*

Comparison for Rule Application. With the three rules correctly inferred, we applied Baseline to D2. In total, Baseline reported nine buggy program commits. Based on our manual inspection, four of these reports violate the rule shown below:

`Change(web-app_servlet_servlet-name)` $\Rightarrow$

`Change(web-app_servlet-mapping_servlet-name)`, *same-value*

Therefore, we consider all of them to be real bugs. In particular, as shown in Table 4.7, these four real bugs were all fixed by developers. For the particular rule mentioned above, Xeditor also revealed the same set of real bugs. According to this experimental comparison, it seems that Baseline works as effectively as Xeditor to detect bugs, given the corresponding set of inferred rules. However, theoretically, Xeditor can work more reliably than Baseline in certain scenarios. For instance, given a rule $Change(A) \Rightarrow Change(B)$, Baseline can interpret the rule in three ways: (a) "*if A is inserted, B should be also inserted*", (b) "*if A is deleted, B should also be deleted*", and (c) "*if A is updated, B should also be updated*" . However, (a) and (b) logically conflict with each other. The logical equivalence of (a) is "*if B is deleted, then A should be also deleted, but not vice versa*", which is totally opposite to (b). Consequently, if $Change(A) \Rightarrow Change(B)$ is purely inferred from the insertions of both entities, given a commit that deletes A, Baseline will erroneously suggest the deletion of B, producing false positives.

Table 4.7: The three real XML bugs fixed by developers

Bug Index	Fixing Strategy	# of Commits in-between	Detected by Xeditor?
1	Update the incorrect XML tag in B	2	✓
2	Insert B	1	✓
3	Insert B	1	✓
4	Insert B	1	✓

Finding 7: *Given the corresponding rule(s) for bug detection,* Baseline *worked as effectively as* Xeditor *to reveal the same set of bugs. However, theoretically,* Xeditor *can detect bugs and suggest changes more reliably.*

4.2 Related Work

The related work of this research includes metadata validation, program change prediction, and configuration debugging.

4.2.1 Validation of Metadata

Approaches were proposed to help check and/or fix the usage of metadata (i.e., XML and annotations) [21, 22, 23, 25, 30, 35, 40]. For instance, XQuery is a widely used query and functional programming language that queries and transforms collections of structured or unstructured data in XML documents [22]. Similarly, CDuce [21] and XDuce [31] are independently developed domain-specific languages (DSLs) for XML processing. To validate and transform XML files, users have to learn one of these DSLs and exploit the DSL to prescribe matching logic and change operations, which procedure can be tedious and error-prone.

To validate Java annotation usage, Eichberg et al. provided a DSL for users to define constraints [25]. To check user-specified constraints, the researchers automatically converted Java bytecode to XML documents, and converted constraints to XQuery path expressions. Similarly, Darwin [23] and Noguera et al. [35] separately defined DSLs for users to specify, and then validate the constraints on annotation usage. However, general developers may not have sufficient domain knowledge to properly utilize these languages.

Song and Tilevich built an approach to automatically infer and check invariants between metadata and program constructs, without requiring users to manually prescribe anything [40]. However, this approach focuses on the relations between metadata and code; it does not handle any consistency constraint within XML files.

4.2.2 Program Change Prediction

Researchers built tools to mine version histories for co-change patterns, and used those patterns to predict any missing change [27, 28, 45, 50]. Specifically, Gall et al. mined release data for the co-change relationship between subsystems [27] and classes [28]. Zimmerman et al. and Ying et al. further extracted the co-change relationship between finer-grained program entities (e.g., classes, methods, and fields) [45, 50]. However, these approaches predict changes *purely* based on entities' co-change frequencies, without considering any syntactic or semantic relationship between entities. When lots of irrelevant entities are accidentally co-changed multiple times, such tools may incorrectly infer rules and produce incorrect predictions.

Some hybrid approaches combine history-based association rule mining with information retrieval (IR) [29, 33, 46]. Given a software entity E, these approaches leverage IR-based techniques to (1) extract terms from E and any other entity and (2) rank those entities

based on their term overlapping with E. Meanwhile, as with the tools mentioned above, these approaches also mine history for co-changes and rank entities accordingly. Given a new commit, these approaches combine the two ranked lists in various ways to reveal any missing change. However, these approaches cannot effectively infer the change patterns imposed by software frameworks, because such tools do not use any domain knowledge to distill framework-related changes from the noisy change data.

Several researchers leveraged the syntactic and textual relationship between entities to predict changes [39, 42]. For instance, Shirabad et al. trained a machine-learning model to characterize any commonality between co-changed files, such as numbers of commonly used types/functions/variables [39]. Given a changed file, the model predicts what other files to be changed together. Wang et al. characterized the commonality between co-changed methods, whose changes were applied for the accesses of new fields [42]. Based on the characterization, Wang et al. built CMSuggester, a tool to predict methods for change given an added field and at least one method changed to access the field. However, the exploited syntactic information prevents such approaches from being applied to non-code artifacts, such as XML files.

4.2.3 Configuration Debugging

Several tools were built to diagnose or fix software configuration errors [19, 20, 36, 43, 48]. These approaches execute buggy software, gather execution profiles, compare the profiles, and conduct dynamic analysis to locate errors. For instance, ConfDiagnoser records program predicates that may be affected by each configuration option, and collects the execution profiles of a program's correct and undesired runs [48]. By comparing the behavioral differences between two types of runs in terms of recorded predicates, ConfDiagnoser identifies the can-

didate options with misconfigured values. Weiss et al. built an approach to generalize system configuration repairs for certain types of machines from the shell commands developers entered to update one machine [43].

The configuration files examined by these approaches are irrelevant to XML documents. None of these tools check for any update coupling between configuration options.

Chapter 5

Discussion

5.1 Lessons We Learned

In this paper, we exploited a probability-based method (i.e., association rule mining) to extract the constraints on DD definition with minimum domain-specific prior knowledge. With this approach, we explored (1) two methodologies that are either based on the co-existence or co-changes of entities within the same file, and (2) multiple parameter settings. Although we revealed various interesting rules and reported the real bugs that developers introduced when updating their XML files, we also learnt new research opportunities and challenges in this area.

First, *related entities do not always occur in the same deployment descriptor.* We observed that some associated entities are defined in separate XML files, while the association relationship is achieved when one XML file imports other XML files or a Java annotation describes that all those XML files are relevant. Our observation indicates that to flexibly infer more rules with a higher precision, we need to conduct cross-file association rule mining, by treating multiple XML files and Java files as a whole and capturing the relationship between files.

Listing 5.1: A file violates an inferred rule but is considered to be valid by its owner developer [8]

```
1 ...
```

```
<beans xmlns="http://www.springframework.org/...>
  <mongo:db-factory dbname="${mongo.database.name}" mongo-ref="mongo" />
  <bean id="mongoTemplate" class= ...>
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory" />
  </bean>
</beans>
```

Second, *some related entities may never refer to the same value.* Listing 5.1 presents a file that violates an inferred rule but is considered to be correct by the owner developer. Based on the rule

`beans_bean_constructor-arg_ref` $\Rightarrow$

`beans_bean_id`, *same-value*

Listing 5.1 has a rule violation and should include another bean defined with the id `mongoDbFactory`. However, according to the developer's feedback email and relevant documentation [38], we learnt that by default, the `<db-factory>` element enables Spring to create an instance of `MongoDbFactory` and to register the instance as a bean named `mongoDbFactory`. In other words, with the existence of `mongo:db-factory`, we should NOT define a bean with the id `mongoDbFactory`. To extract such delicate constraints, we may need to define more heuristics and exploit more domain-specific insights.

Third, *some related entities may not frequently co-occur.* As shown in Listing 4.1, even though the association relationship between `<beans><bean><property><ref>` and `<beans><axon:event-sourcing-repository> <id>` is not supported by statistics, the relationship can still be real and put constraints on the defined deployment descriptors. We believe that this limitation is commonly share among all probability-based rule inference approaches. More advanced and novel techniques are still needed to reveal such constraints.

Fourth, *the fixing strategies for rule violations can vary a lot.* Currently, when a rule $A \Rightarrow B$ is violated, Xeditor suggests developers to either remove A or insert B. However, in reality, developers' fixing strategies can be more diverse, such as modifying an existing entity to satisfy the constraint or importing an XML file with B defined. Without more domain knowledge about any particular project, it is hard to precisely predict which fixing strategy developers will adopt. Therefore, even though Xeditor can locate some real bugs and provide two applicable solutions, researches may need more sophisticated techniques to automate the whole process and repair XML bugs fully automatically.

5.2 Threats to Validity

Threat to External Validity We applied Xeditor to 1,137 open-source projects for rule inference, and applied it to another 478 open-source projects in order to detect real XML bugs. The inferred rule set and detected bugs are limited to our experiment data sets. The observations may not generalize well to close-source projects. In the future, we would like to include more projects into our evaluation, or even include close-source projects if possible, so that our findings are more representative.

Threat to Construct Validity Although we tried our best to manually inspect the inferred rules and reported bugs, it is possible that our manual analysis is subject to human bias and restricted by our limited domain knowledge. To mitigate the problem, we sent emails to developers who owned the open-source projects and asked whether a reported rule violation makes sense or not. So far, we have not received much feedback from those developers. As we gather more comments from these domain experts, we can further improve the quality of inferred rules and change suggestions.

Threat to Internal Validity We currently leveraged the heuristic that associated XML entities should (1) frequently co-exist in the same files and (2) refer to the same data value. However, some associated entities may not satisfy these criteria. For instance, certain related entities may not co-exist in many files due to the limited usage of their corresponding software frameworks. Furthermore, some relevant entities do not refer to the same value even though they co-occur a lot. Consequently, the extracted rules are limited by our heuristic. In the future, we will mitigate such problems by defining more heuristics.

Chapter 6

Conclusions

It is hard to create and maintain XML files due to the many domain-specific constraints, and seldom compilers detect/fix XML errors. In this paper, we built a novel approach—Xeditor—to automatically extract co-existence rules and applied those rules to detect bugs. Xeditor applied association rule mining to infer rules which is similar with prior work, however, Xeditor infers rules from the co-existence instead of co-changes between entities which was applied in prior work. Our evaluation shows that the co-existence-based methodology infer rules with higher precision than co-change-based methodology. The reason is some associated entities co-occur but are seldom modified. In this scenario, the co-change method will not infer the association rule relationship. Our results show that Xeditor infer rules with precision (83%) and report bugs with 74.6% precision and 50% recall. More importantly, our experiments show that Xeditor can detect the real bugs that developers actually introduced and later fixed in software version history. It implies that Xeditor can help developers detect and fix real XML bugs. In the future, we will explore more advanced approaches to accurately extract and apply more complicated rules.

Bibliography

[1] addressbook-sample-mongodb. https://github.com/yholkamp/addressbook-sample-mongodb/blob/master/contacts/src/main/resources/META-INF/spring/contacts-context.xml.

[2] AxonFramework. https://github.com/efemelar/AxonFramework.

[3] bagh. https://github.com/moghim/bagh/commit/cae2a77fbdefedc823291a5fa98a3f51bb0c4816#diff-411d1a9625fc3f5d5be34d116942f6ca.

[4] demo-web. https://github.com/agile-shark/demo-web/blob/master/src/main/webapp/WEB-INF/web.xml.

[5] Github. https://github.com.

[6] Introduction to Web Application Deployment Descriptors. https://docs.oracle.com/cd/E19226-01/820-7627/bncbj/index.html.

[7] lushtext. https://github.com/hunsang/lushtext/blob/master/lush/target/sht_webapp/WEB-INF/classes/spring/context-idgen.xml.

[8] poc-spring-data-mongodb. https://github.com/rodrigozrusso/poc-spring-data-mongodb/blob/master/src/main/resources/spring-mongodb.xml.

[9] Spring - Bean Definition. https://www.tutorialspoint.com/spring/spring_bean_definition.htm.

[10] StackOverflow. https://stackoverflow.com.

[11] The Deployment Descriptor: web.xml. https://cloud.google.com/appengine/docs/standard/java/config/webxml.

[12] Viewing deployment descriptors. https://www.ibm.com/support/knowledgecenter/en/SS7K4U_9.0.5/com.ibm.websphere.zseries.doc/ae/trun_app_deploymtdesc.html.

[13] WildFly. https://wildfly.org.

[14] Working with Security Roles. https://docs.oracle.com/cd/E19226-01/820-7627/bncav/index.html.

[15] XML Syntax. https://www.quackit.com/xml/tutorial/xml_syntax.cfm.

[16] Introduction to Web Application Deployment Descriptors. https://docs.oracle.com/cd/E19226-01/820-7627/bncbj/index.html, 2010.

[17] Securing REST urls with Spring. https://stackoverflow.com/questions/13836451, 2012.

[18] How to correctly manage feature configuration deployment in JBoss Fuse 6.2.1? https://stackoverflow.com/questions/39706237, 2016.

[19] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 281–286, Berkeley, CA, USA, 2008. USENIX Association.

[20] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 237–250, Berkeley, CA, USA, 2010. USENIX Association.

[21] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An xml-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63, New York, NY, USA, 2003. ACM.

[22] M. Brundage. *XQuery: The XML Query Language*. Pearson Higher Education, 2004.

[23] I. Darwin. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering*, 2010.

[24] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[25] M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *Proceedings of the 8th International Conference, Held As Part of the Joint European Conference on Theory and Practice of Software Conference on Fundamental Approaches to Software Engineering*, FASE'05, pages 237–252, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[27] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. ICSM*, pages 190–198, 1998.

[28] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. IWPSE*, pages 13–23, 2003.

[29] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 430–440, June 2012.

[30] H. Hosoya and B. C. Pierce. Regular expression pattern matching for xml. *J. Funct. Program.*, 13(6):961–1004, Nov. 2003.

[31] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

[32] M. A. Islam, M. M. Islam, M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. [research paper] detecting evolutionary coupling using transitive association rules. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 113–122, Sep. 2018.

[33] H. H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18:933–969, 2012.

[34] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty. Secure coding practices in Java: Challenges and vulnerabilities. In *ICSE*, 2018.

[35] C. Noguera and L. Duchien. *Annotation Framework Validation Using Domain Models*, pages 48–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[36] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 193–202, Washington, DC, USA, 2011. IEEE Computer Society.

[37] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212, March 2016.

[38] J. Sharma and A. Sarin. *Getting Started with Spring Framework: Covers Spring 5*. CreateSpace Independent Publishing Platform, USA, 4th edition, 2017.

[39] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *Proc. ICSM*, pages 95–104, 2003.

[40] M. Song and E. Tilevich. Metadata invariants: Checking and inferring metadata coding conventions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 694–704, June 2012.

[41] Spring security JDK based proxy issue while using @Secured annotation on Controller method. https://stackoverflow.com/questions/35860442/spring-security-jdk-based-proxy-issue-while-using-secured-annotation\protect\discretionary{\char\hyphenchar\font}{}{}on-control.

[42] Y. Wang, N. Meng, and H. Zhong. Cmsuggester: Method change suggestion to complement multi-entity edits. In *SATE*, 2018.

[43] A. Weiss, A. Guha, and Y. Brun. Tortoise: Interactive system configuration repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 625–636, Piscataway, NJ, USA, 2017. IEEE Press.

[44] S. Xu, Z. Dong, and N. Meng. Meditor: Inference and application of api migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 335–346, Piscataway, NJ, USA, 2019. IEEE Press.

[45] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept 2004.

[46] M. B. Zanjani, G. Swartzendruber, and H. Kagdi. Impact analysis of change requests on source code based on interaction and commit histories. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 162–171, New York, NY, USA, 2014. ACM.

[47] C. Zhang and S. Zhang. *Association Rule Mining: Models and Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2002.

[48] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 312–321, Piscataway, NJ, USA, 2013. IEEE Press.

[49] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[50] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. ICSE*, pages 563–572, 2004.