Inferring and Applying Def-Use Like Configuration Couplings in Deployment Descriptors

Chengyuan Wen
wechyu88@vt.edu
Virginia Tech
Blacksburg, Virginia

Yaxuan Zhang
yaxuan93@vt.edu
Virginia Tech
Blacksburg, Virginia

Xiao He
hexiao@ustb.edu.cn
University of Science and Technology
Beijing, China

Na Meng
nm8247@vt.edu
Virginia Tech
Blacksburg, Virginia

ABSTRACT

When building enterprise applications on Java frameworks (e.g., Spring), developers often specify components and configure operations with a special kind of XML files named "**deployment descriptors (DD)**". Maintaining such XML files is challenging and time-consuming; because (1) the correct configuration semantics is domain-specific but usually vaguely documented, and (2) existing compilers and program analysis tools rarely examine XML files. To help developers ensure the quality of DD, this paper presents a novel approach—XEDITOR—that extracts configuration couplings (i.e., frequently co-occurring configurations) from DD, and adopts the coupling rules to validate new or updated files.

XEDITOR has two phases: coupling extraction and bug detection. To identify couplings, XEDITOR first mines DD in open-source projects, and extracts XML entity pairs that (i) frequently coexist in the same files and (ii) hold the same data at least once. XEDITOR then applies customized association rule mining to the extracted pairs. For bug detection, given a new XML file, XEDITOR checks whether the file violates any coupling; if so, XEDITOR reports the violation(s). For evaluation, we first created two data sets with the 4,248 DD mined from 1,137 GitHub projects. According to the experiments with these data sets, XEDITOR extracted couplings with high precision (73%); it detected bugs with 92% precision, 96% recall, and 94% accuracy. Additionally, we applied XEDITOR to the version history of another 478 GitHub projects. XEDITOR identified 25 very suspicious XML updates, 15 of which were later fixed by developers.

KEYWORDS

Configuration coupling, deployment descriptor, rule mining

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416577

ACM Reference Format:
Chengyuan Wen, Yaxuan Zhang, Xiao He, and Na Meng. 2020. Inferring and Applying Def-Use Like Configuration Couplings in Deployment Descriptors. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416577

1 INTRODUCTION

When building enterprise applications on top of software frameworks (e.g., Java EE platforms), developers usually create deployment descriptors (e.g., `web.xml`)—a special kind of XML files—to configure deployment options [11]. Erroneous DD can trigger abnormal runtime behaviors [44] or confusing errors [13]. Debugging such XML files can be challenging and time-consuming for three reasons. First, frameworks have domain-specific rules to define or specify **deployment options as XML entities (i.e., elements and attributes)**, and developers have application-specific ways to configure DD for distinct needs. Unfortunately, the domain-specific rules and application-specific configurations are usually vaguely documented [14]. Second, it is tedious and error-prone for developers to memorize all DD-related rules. Third, existing compilers and tools examine source code instead of XML files. Even though XML file validators can be built to validate syntax based on XML Schemas or DTDs, the validators do not examine DD semantics.

Existing research provides quite limited support for checking or transforming XML files [20, 22, 30]. For instance, XQuery is a domain-specific language for finding elements and attributes in XML documents [22]. To find particular XML errors, developers have to learn XQuery, and then use XQuery to manually describe the pattern matching mechanism. The learning curve of XQuery may be long to some developers, while the pattern definition procedure can be also tedious and error-prone. We believe that *with an automatic approach to (1) infer rules from correct XML configuration files and (2) apply those rules in order to locate erroneous XML files, we can provide quality assurance for XML files without requiring much human effort.*

This paper presents XEDITOR, our novel approach to infer and apply XML rules based on open-source projects. Because different frameworks define diverse formats of XML files and the DD semantics vary a lot, it is almost infeasible to infer arbitrarily complex XML rules in a domain-agnostic way. To ensure the generalizability of our approach, we designed XEDITOR to mainly focus on one type of rules that commonly exist in distinct frameworks: **def-use**

```
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance …>
 <bean id="OutputHelper" class="com.mkyong.output.OutputHelper">
  <property name="outputGenerator">
   <ref local="CsvOutputGenerator"/>
  </property>
 </bean>

 <bean id="CsvOutputGenerator"
class="com.mkyong.output.impl.CsvOutputGenerator"/>
 …
</beans>
```

Figure 1: An exemplar deployment descriptor

```
    <bean id="validator"
class="org.springframework...FactoryBean">
      <property name="validationMessageSource"
ref ="messageSource"/>
    </bean>
+ <!--
    <bean id="messageSource" class=...>
      …
    </bean>
+ -->
```

Figure 2: The incorrectly maintained `dispatcher-servlet.xml` file in project `addressbook-sample-jpa` that violates $ref \rightarrow id$ [12]

like couplings "$A \rightarrow B$" between XML entities. Here, "$A \rightarrow B$" means "if entity A refers to an identifier or a string literal, then there must be another entity B that defines meaning for that identifier."

For instance, in Spring [5], a bean is an object that is instantiated, assembled, and managed by a Spring container. In DD, a reference to a bean identifier is always coupled with the identifier definition, or any bean reference is valid only when there is a definition for the bean. Figure 1 presents an example to demonstrate this constraint. In this figure, entity `ref` is a reference to `CsvOutputGenerator`, which is coupled with the bean `id` declaration of `CsvOutputGenerator`, i.e., $ref \rightarrow id$. Our research intends to reveal DD semantic rules similar to such def-use couplings. Notice that although $ref \rightarrow id$ is easy to understand, based on our experience, many def-use like rules are not so obvious (see Listing 1). More importantly, developers sometimes violated such rules when maintaining DD. Figure 2 shows an incorrectly updated deployment descriptor in the open-source project `address-sample-jpa` [12], where developers commented the bean `id` declaration of `messageSource` but kept a `ref` to that bean.

We believe that "*if two entities frequently coexist in the same file and often hold the same data, they are correlated*". With this insight, we designed XEDITOR to have two phases, as illustrated by Figure 3. Given DD or an XML corpus from open-source projects, Phase I extracts candidate pairs of XML entities that (i) frequently co-occur in XML files, and (ii) hold the same data or string literals at least once. For each candidate pair, XEDITOR identifies the longest common XML path C shared by both entities on the XML parsing tree, and then contextualizes the representation of both entities based on C. For every candidate (A, B), XEDITOR tentatively extracts couplings by applying our customized association rule mining technique. In particular, XEDITOR identifies all occurrences of each entity in the XML corpus, together with the corresponding string literals. If (1) the occurrence of one entity (e.g., A) is usually coupled with that of the other entity (e.g., B), and (2) both entities usually hold identical string literals, then XEDITOR infers a rule (e.g., $A \rightarrow B$).

Phase II takes in an XML file f that developers newly created or updated from an existing file, XEDITOR tentatively matches f

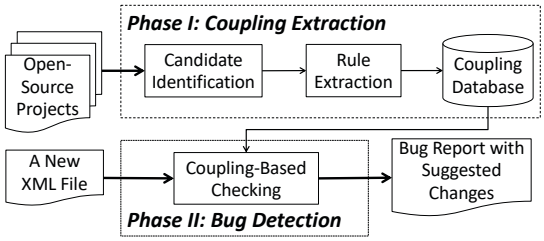

Figure 3: XEDITOR consists of two phases: Phase I extracts configuration couplings from open-source projects, and Phase II uses the couplings to check for bugs in DD

against all extracted couplings. If there is a coupling rule for which the file contains A without B, XEDITOR recommends developers to (1) insert B or delete A and (2) ensure both entities hold the same string literal. 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 DD and complement existing code-oriented program analysis techniques.

For evaluation, we first applied XEDITOR to the 4,248 DD from 1,137 projects, and manually inspected the extracted couplings. With the default parameter setting, XEDITOR identified 30 couplings, among which 22 couplings are true positives. It means that our approach can extract rules with high precision (73%). Furthermore, based on our manual inspection results, we built a ground truth data set of coupling occurrences in the 4,248 DD. We randomly split the XML corpus into 10 portions and conducted 10-fold cross validation to evaluate XEDITOR's effectiveness of bug detection. In each fold, we used nine portions of data for coupling extraction; we constructed test cases by removing some XML entities from the remaining one portion of data, and applied XEDITOR to those test cases. Our evaluation shows that on average, XEDITOR detected bugs with 92% precision, 96% recall, and 94% accuracy.

Additionally, we applied XEDITOR to the program commits in another 478 open-source projects. XEDITOR revealed 25 incorrectly updated DD, 15 of which were later fixed by developers. This implies that XEDITOR can help developers avoid introducing bugs when they modify DD. Finally, we compared XEDITOR with a baseline approach that extracts couplings from co-changed entities, and applies both approaches to the same data sets. Our comparison shows that XEDITOR detected more rule violations than the baseline; XEDITOR obtained lower precision (92% vs. 98%), much higher recall (96% vs. 78%), and higher accuracy (94% vs. 87%).

In summary, this paper makes the following contributions:

- We developed a novel approach—XEDITOR—to automatically extract configuration couplings in DD and detect related bugs. Different from most prior work, XEDITOR does not need users to manually prescribe any rule or matching logic.
- We built XEDITOR to extract couplings from the coexistence of XML entities. Compared with a baseline technique that extracts couplings from co-changed entities, XEDITOR worked better by detecting rule violations with higher accuracy.
- We conducted a comprehensive evaluation on XEDITOR. Our evaluation shows that (1) XEDITOR could identify important couplings because developers did make mistakes by ignoring

such delicate constraints, and (2) XEDITOR suggested useful corrective changes for buggy XML files.

At https://figshare.com/s/d4dc1f8ab527c1ce68ef, we open-sourced our program and data.

2 BACKGROUND

This section first introduces DD (Section 2.1) and XML syntax (Section 2.2). It then explains why it is challenging to configure DD appropriately (Section 2.3).

2.1 Deployment Descriptors (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 deployment descriptor (e.g., web.xml) describes component classes, resources, and configurations of *App*; it also specifies how a server uses these components to serve web requests [7]. Similarly, in a Java EE application, the deployment descriptor (e.g., application.xml) clarifies the configurations, container options, and security settings [8]. **XML is used for the syntax of DD**. Depending on the types of applications and modules, DD may be located in various file folders and named differently.

Listing 1: A simplified version of a web.xml **file [2]**

```
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>
```

2.2 XML Syntax

XML syntax defines how an XML file can be written [10]. According to the syntax rules, each XML file includes one or more **XML elements**, which are organized in a tree structure. Namely, there is only one **root element** in any XML file, and the root element has one or more **child elements**. For the exemplar XML file shown in Listing 1, the `<web-app>` element is root; one of its child element is `<security-constraint>`.

Generally speaking, an XML document consists of **markups** and **data**. Markups are provided in the form of **tags** and **attributes**. Data is the text that goes in between tags or is provided as values for 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`") Attributes can be added to XML elements; they are represented as **name-value pairs** (e.g., "`version="2.5"`"). This paper uses **XML entities** to refer to both XML elements and attributes.

2.3 Problem Statement

In DD, there are various rules that developers have to follow in order to realize their deployment requirements. For instance, as shown in Listing 1, a security constraint (`security-constraint`) is used to define the access privileges to a collection of resources; an authorization constraint (`auth-constraint`) authorizes certain role(s) with the defined access privileges, and has one or more `role-name` elements to list the authorized roles [17]. Meanwhile, a security role (`security-role`) defines an abstract name that can be assigned to users and groups [9]. A relevant rule is that ***each role name listed in*** `auth-constraint` ***(e.g.,*** `comm`***) must correspond to the role name defined in one of the*** `security-role` ***elements (e.g.,*** `comm`***)***. However, in the big and lengthy Java EE Tutorial (with 980 pages), there is only one small paragraph together with a single code example [17] vaguely implying the above-mentioned rule:

"The following snippet of a deployment descriptor declares the roles that will be used in an application using the `security-role` *element and specifies which of these roles is authorized to access protected resources using the* `auth-constraint` *element: ..."*

It is very tedious and error-prone for developers to identify, remember, and follow all domain-specific rules when they maintain DD. According to a recent study on StackOverflow [36], many developers asked various questions on how to correctly configure DD and expressed frustrations with XML debugging. Unfortunately, there is limited tool support for bug detection or fix in DD. Two reasons may explain such technique insufficiency:

(1) Domain-specific rules are usually vaguely described or even poorly documented, so it can be time-consuming for tool builders to extract rules from software library/framework documentation and then code those rules into their tools.
(2) Different software defines divergent DD rules, so it can be challenging for tool builders to frequently integrate the rules related to newly released software into their tools.

To build a tool that can help developers debug DD, we need to solve the two technical challenges mentioned above.

3 APPROACH

In this section, we explain our automatic approach—XEDITOR—that detects bugs in XML files and provides corrective suggestions. To overcome the two technical challenges mentioned in Section 2.3, we designed XEDITOR to infer def-use like configuration couplings from open-source DD, and to adopt the inferred couplings for bug detection. As shown in Figure 3, XEDITOR has two phases. This section first summarizes the steps in each phase and then describes each step in detail (Section 3.1-Section 3.3).

Phase I: Rule Inference

- Given a set of open-source projects, XEDITOR locates a set of DD: $F = \{f_1, f_2, \ldots, f_m\}$, from which files XEDITOR extracts candidate XML entity pairs $C = \{c_1, c_2, \ldots, c_n\}$, where $c_i = (e_{i1}, e_{i2})$ with e_{i1} and e_{i2} being coexisting entities.
- For each candidate c_i, XEDITOR searches among all files F to find the occurrence of either entity (i.e., e_{i1} and e_{i2}); it further applies our customized association rule mining to infer any def-use like couplings between entities. Each mined rule has the format "$A \rightarrow B$" and is saved into a database D.

Phase II: Rule Application

- Given a new XML file f, for each rule $r \in D$, XEDITOR checks whether A and B coexist in the file. XEDITOR reports a bug if (1) A exists but B does not, or (2) A refers to a string literal which is not held by any B.

By inferring def-use like rules from open-source DD in a domain-agnostic way, we avoid the manual effort of (1) extracting such rules from software documentation and (2) hardcoding the rules in XEDITOR. By implementing the first phase to store rules to D and the second phase to load rules from D, we ensure that XEDITOR can be easily extended to cover new rules introduced by newly released software frameworks or libraries.

3.1 Candidate Identification

Given a set of open-source projects, we need to first locate DD. Although the DD of different Java projects are all XML files, not all XML files are DD. To efficiently locate DD among the available XML files, we adopted a heuristic to focus on files whose paths have any of the following keywords: "WEB-INF", "spring", "security", and "web". We defined this heuristic because based on our experience, DD usually exist in specific folders or have specialized names.

For each located XML file, we applied Antlr [15, 16] to generate a parsing 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 for any two coexisting entities (i.e., (e_i, e_j), where $i, j \in [1, n]$ and $i \neq j$).

However, since many irrelevant entities may coexist in the same file for distinct requirements, their coexistence is meaningless. Therefore, many of the candidate pairs constructed by the above-mentioned naïve approach are actually useless for rule inference.

To overcome the challenge of noisy entity pairs and to identify promising candidates, we used a heuristic that "*if two XML entities hold the same string value at least once, they are likely to be correlated*". As shown in Listing 1, the `<role-name>` element under `<security-constraint>` holds the data "`comm`" (line 9), while the `<role-name>` element under `<security-role>` contains the same string literal (line 14). Therefore, XEDITOR generates a candidate pair based on the two elements.

Before generalizing rules from the concrete candidate pairs, we need to solve another challenge: *how can we represent candidates in an unambiguous way?* In Listing 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 parent and even ancestors 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 root to A. 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).
```

When a rule-to-infer r has multiple occurrences, it can correspond to multiple candidates with distinct XML paths, such as c_p=(`beans_bean_id`, `beans_bean_property_ref`) and c_q=(`beans_beans_bean_id`, `beans_beans_bean_property_ref`) shown in Figure 4. If we do not appropriately process the path divergences between candidates, we may fail to infer true rules or always infer duplicated rules. Essentially, we need an **abstract context-aware representation** of

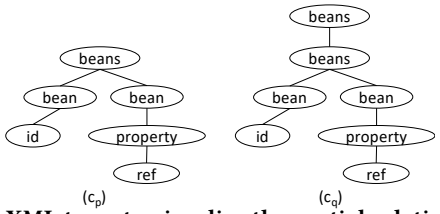

Figure 4: XML trees to visualize the spatial relationship between entities in c_p and c_q

candidates that is (1) sensitive to the path divergences between entities inside each candidate, but (2) insensitive to the path divergences among candidates showing the same rule.

To create the abstract representation, for each candidate $c_i = (e_{i1}, e_{i2})$, XEDITOR identifies the lowest common XML ancestor, and converts the paths of both entities based on that ancestor. For instance, the entities in c_p of Figure 4 have the lowest common ancestor as `beans`, so the abstract representation is (`*_beans_bean_id`, `*_beans_bean_property_ref`). Here "**wildcard (*)**" represents the common XML path prefix of `beans` shared by both entities. Similarly, c_q is converted to (`*_beans_bean_id, *_beans_bean_property_ref`). Therefore, the above-mentioned candidate pair from Listing 1 can be abstracted as:

```
(*_web-app_security_security-constraint_auth-constraint_role-name,
 *_web-app_security_security-role_role-name).
```

By representing candidate pairs in an abstract and context-aware way, we can (1) differentiate between same-tag entities, (2) cluster candidates showing the same rule, and (3) simplify rule extraction and application (Section 3.2-Section 3.3).

3.2 Rule Extraction

After extracting a set of candidate pairs (i.e., $C = \{c_1, c_2, \ldots, c_m\}$) from corpus F, XEDITOR infers rules with our customized association rule mining technique. This technique first adopts traditional association rule mining [51] to infer the occurrence coupling between XML entities; it then applies two filters to remove cooccurring entity pairs that are less likely to have def-use relations.

Association rule mining (ARM) [51] is 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 when 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 cooccur for a sufficient number of times and whenever e_1 occurs, e_2 usually occurs. Formally, suppose that the numbers of occurrences of e_1 and e_2 are separately $freq(e_1)$ and $freq(e_2)$. We represent the number of cooccurrences 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 cooccurrences, 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 $c_i = (e_{i1}, e_{i2})$, 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 reveal existence couplings between entities.

Two Filters. In certain projects, some irrelevant entities accidentally coexist a lot. To avoid inferring noisy def-use like rules from such accidental coexistence, we decided to build two filters that refine the above-mentioned existence couplings.

(F1) $pfreq(e_1, e_2) \geq pth$, where pth is the minimum number of projects that support the cooccurrences between e_1 and e_2.

We designed this filter because certain accidental coexistence was introduced by the coding habits of some programmers. However, it is very unlikely that such accidental coexistence popularly exists in many projects. This filter removes any rule that is only supported by a small number of projects.

(F2) $Pr(same\_string) = \dfrac{freq(string(e_{i1}) = string(e_{i2}))}{freq(e_{i1}, e_{i2})} \geq vth$, where vth is the minimum rate of same-string cooccurrences.

We designed this filter because in some DD, developers unnecessarily set irrelevant XML entities to hold identical values such that they do not have to carefully examine the correspondence between entities. However, more developers still use distinct and meaningful string values to tell apart irrelevant entities. Therefore, this filter removes any rule where the two entities do not frequently hold the same string literal.

By default, XEDITOR extracts a def-use like rule "$e_{i1} \rightarrow e_{i2}$" if $supp = 10$, $conf = 0.9$, $vth = 0.95$, and $pth = 9$. We used these threshold values because our evaluation shows that XEDITOR works most effectively with this setting (Section 4.3).

At the end of this step, XEDITOR obtains a set of def-use configuration couplings and saves them into its database D. For the example shown in Listing 1, from this exemplar file together with several other files containing relevant entity pairs, XEDITOR can infer the following rule:

`*_web-app_security-constraint_auth-constraint_role-name` $\rightarrow$
`*_web-app_security-role_role-name.` (Rule 1)

3.3 Rule-Based Checking

Given a new XML file f, XEDITOR enumerates the inferred rules to detect bugs in the file and suggests changes when possible. Specifically, XEDITOR adopts Antlr to create a parsing tree for f and extracts all entities from the tree. For each rule $r = A \rightarrow B$ in the database, XEDITOR searches entity matches for A and B separately. For simplicity, we represent the two sets of found matches with $E_A = \{e_{A1}, e_{A2}, \ldots, e_{Ah}\}$ and $E_B = \{e_{B1}, e_{B2}, \ldots, e_{Bl}\}$. Next, XEDITOR pairs up entities between E_A and E_B based on (1) abstract context-aware representations of entities and (2) the common string literals they share. If e_{Ai} ($i \in [1, h]$) cannot be paired up with any entity in E_B, XEDITOR reports a bug and suggests possible fixes.

For instance, by applying the above-mentioned inferred rule (Rule 1) to the XML file in Listing 2, XEDITOR can find two matches for the antecedent `<role-name>` element under `<security-constraint>`, but find no match for the consequent `<role-name>` element. In this scenario, XEDITOR reports two bugs and suggests the corresponding fixes as below:

"*Insert a <role-name> entity with value 'prof' under <web-app> <security-role>; or delete the <role-name> entity with value 'prof' under <security-constraint> <auth-constraint>.*"

"*Insert a <role-name> entity with value 'stu' under <web-app> <security-role>; or delete the <role-name> entity with value 'stu' under <security-constraint> <auth-constraint>.*"

Listing 2: A simplified version of another web.xml **file [1]**

```
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>
```

4 EVALUATION

This section first introduces our data sets (Section 4.1) and evaluation metrics (Section 4.2). Next, it presents our evaluation on XEDITOR's effectiveness of rule inference (Section 4.3) and rule application (Section 4.4). Finally, it expounds on the comparison between XEDITOR and a baseline technique (Section 4.5).

4.1 Data Sets

We constructed two major data sets for evaluation. Specifically, we mined Java open-source projects on GitHub [3] using the heuristics described in Section 3.1. We crawled the website for any project that contains at least one XML file, whose file path has any of the following keywords: "WEB-INF", "spring", "security", and "web". After removing redundant projects, we randomly put the crawled projects into two data sets. The first set (**D1**) contains 70% of projects (i.e., 1,137), while the second set (**D2**) contains 30% of projects (i.e., 478). For D1, XEDITOR identified 4,248 DD in the latest version of 1,137 projects. We used these DD to explore three research questions:

RQ1: How effectively does XEDITOR infer rules?
RQ2: How effectively does XEDITOR apply rules to locate bugs?
RQ3: How well does XEDITOR compare with a baseline approach, which detects XML bugs based on co-changed instead of coexisting entities?

After inferring rules from D1, we used XEDITOR to apply rules to individual program versions in D2. We investigated two questions:

RQ4: Did developers introduce any of the XML bugs that XEDITOR can detect during XML file maintenance?
RQ5: How well do XEDITOR's change suggestions match developers' actual fixes for those detected XML bugs?

Our experiments with D1 intend to assess the usability of XEDITOR, while our experiments with D2 intend to mimic the real application scenarios of XEDITOR and assess the tool's usefulness to developers. Notice that XEDITOR is actually applicable to arbitrary DD to examine for any rule violation, no matter whether the files have been newly created, recently updated, or unchanged for a long time.

In our experiments with D2, we intentionally applied XEDITOR to revisions of programs to demonstrate one typical usage of the tool.

4.2 Metrics

We used the following metrics to evaluate the effectiveness of XEDITOR and the baseline technique.

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

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of generated reports}} \times 100\%. \quad (1)$$

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 a technique:

$$R = \frac{\text{\# of correct reports}}{\text{Total \# of true positives}} \times 100\%. \quad (2)$$

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:

$$F = \frac{2 \times P \times R}{P + R} \times 100\%. \quad (3)$$

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

4.3 Effectiveness of Rule Inference

There are four parameters in XEDITOR that can influence its capability of rule inference: *supp*, $conf$, vth and pth. To investigate how sensitive XEDITOR is to these parameters, we applied XEDITOR to the first data set D1 with distinct parameter settings, 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, it is infeasible for us to evaluate the recall of XEDITOR's rule inference. To evaluate the precision, given a rule reported by XEDITOR, our manual inspection involves online search for (1) the related library specifications or framework tutorials, and/or (2) relevant discussions on technical forums (e.g., StackOverflow [6]). We consider an inferred rule to be true if online documentation or discussion recommends it, or no online XML example violates the rule.

4.3.1 Sensitivity to supp. We increased *supp* from 0 to 19, and explored 20 settings for the parameter. Due to the space limit, in Table 1, we present the results for only five settings. Intuitively, *supp* counts the number of supporting instances for any candidate rule. The more supporting instances there are, the more likely that a candidate rule is a real one. As shown in the table, when *supp* increases from 8 to 12, the number of inferred rules by XEDITOR decreases from 56 to 40, while the number of correctly inferred rules decreases from 34 to 24. The observations are understandable because with the other parameters unchanged, the more supporting instances are required, the fewer rules have sufficient occurrence rates to meet the criterion. Among all investigated settings, we identified a precision peak at $supp = 10$, which is 64%. Therefore, we set $supp = 10$ by default.

Table 1: Rule inference with different settings of *supp*

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
8	0.9	0.9	3	56	34	61
9	0.9	0.9	3	53	32	60
10	0.9	0.9	3	50	32	64
11	0.9	0.9	3	44	27	61
12	0.9	0.9	3	40	24	60

Table 2: Rule inference with different settings of $conf$

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
10	0.75	0.9	3	59	32	54
10	0.80	0.9	3	54	32	59
10	0.85	0.9	3	52	32	62
10	0.90	0.9	3	50	32	64
10	0.95	0.9	3	40	25	63

Table 3: Rule inference with different settings of vth

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
10	0.9	0.75	3	67	38	57
10	0.9	0.80	3	58	35	60
10	0.9	0.85	3	53	33	62
10	0.9	0.90	3	50	32	64
10	0.9	0.95	3	36	23	64

Table 4: Rule inference with different settings of pth

supp	conf	vth	pth	# of Inferred Rules	# of Correct Rules	Precision (%)
10	0.9	0.95	8	32	23	72
10	0.9	0.95	9	30	22	73
10	0.9	0.95	10	27	20	74
10	0.9	0.95	11	24	17	71
10	0.9	0.95	12	15	10	75

4.3.2 Sensitivity to $conf$. We increased $conf$ from 0.00 to 0.95 with 0.05 increment, and explored 20 settings for the parameter. Due to the space limit, Table 2 only presents our results for five parameter settings. Theoretically, for each candidate rule $A \rightarrow B$, $conf$ reflects the likelihood of B's occurrence given A's occurrence. Thus, the higher $conf$ is, the more likely that a candidate rule is a true positive. According to the table, as $conf$ increases from 0.75 to 0.95, the number of inferred rules decreases from 59 to 40, while the number of correctly inferred rules remains to be 32 initially and then decreases to 25. The major reason for these observed trends is that as the threshold increases, there are fewer rules satisfying the filtering condition. Because the precision of XEDITOR's inference increases first and then decreases, we set $conf = 0.9$ by default.

4.3.3 Sensitivity to vth. We increased vth from 0.00 to 0.95 with 0.05 increment, and explored 20 settings for the parameter. Table 3 shows our results for five of those investigated settings. Basically, given a candidate rule $A \rightarrow B$, vth reflects the ratio of same-value cooccurrences between A and B among all of their cooccurrences. Therefore, the higher vth is, the more convincing a candidate rule is. In Table 3, as vth increases, XEDITOR inferred fewer rules and

Table 5: The 10-fold cross validation for XEDITOR's effectiveness of bug detection

Id	Rule Inference		Rule Application					
	# of Rules Inferred	# of Correctly Inferred Rules	# of Bugs Reported	# of Known Bugs	# of Correctly Reported Bugs	Precision	Recall	Accuracy
1	25	20	304	283	271	89%	96%	92%
2	25	20	322	291	285	89%	98%	93%
3	21	17	258	251	240	93%	96%	94%
4	23	19	307	300	292	95%	97%	96%
5	26	21	283	276	272	96%	99%	97%
6	23	20	298	277	272	91%	98%	95%
7	21	17	270	269	252	93%	94%	94%
8	20	18	293	271	265	90%	98%	94%
9	20	18	270	268	253	94%	94%	94%
10	22	18	282	266	244	87%	92%	89%

acquired fewer correct rules; the inference precision goes up from 57% to 64%. Thus, by default, we set $vth = 0.95$.

4.3.4 Sensitivity to pth. We increased pth from 0 to 19, and explored 20 settings for the parameter. Table 4 shows our results for five of the explored settings. Essentially, pth counts the number of projects holding at least one supporting instance for any candidate rule. The more projects there are to support a candidate rule, the more possible that the rule is true. In Table 4, as pth increases, the number of rules inferred by XEDITOR decreases from 32 to 15 and the number of correct rules decreases from 23 to 10. The precision rate first increases, then decreases, and next increases again. To achieve a good trade-off between the number of correctly inferred rules and inference precision, we set $pth = 9$ by default.

Finding 1: *For rule inference, XEDITOR's effectiveness is considerably influenced by the settings of all four parameters. As each parameter has its value increased, XEDITOR usually infers fewer rules and obtains fewer correct rules, while the precision rate may increase, decrease, or remain the same.*

4.3.5 XEDITOR' Precision for Rule Inference. After manually checking all inferred rules by XEDITOR with different parameter settings, we confirmed 57 correctly inferred rules. Among these rules, 22 rules can be retrieved by XEDITOR with its default configuration: $supp = 10$, $conf = 0.9$, $vth = 0.95$, and $pth = 9$. Essentially, different parameter settings indicate different trade-offs between two factors: (1) the number of inferred rules and (2) the precision rate. When we decided upon the default setting, we cared more about precision than the other factor. This is because to make XEDITOR usable to people, we want to ensure that most of the rules reported by XEDITOR are correct and valuable.

Our experiment implies significant space for future improvement in rule inference from DD. Specifically, with our carefully chosen parameter configuration, XEDITOR inferred rules with 73% precision; novel approaches are still needed to considerably boost the precision rate. Additionally, among the 57 validated rules so far, at most 38 rules can be retrieved with one of the explored parameter combinations. More advanced techniques are still in need to identify more true rules with reasonably good precision rates.

Finding 2 (Response to RQ1): *With the default setting $supp = 10$, $conf = 0.9$, $vth = 0.95$, and $pth = 9$, XEDITOR effectively inferred rules with high precision (73%).*

4.4 Effectiveness of Bug Detection and Fix

To evaluate XEDITOR's effectiveness of rule application, we conducted two experiments. The first experiment splits D1 into 10 portions evenly, and conducts 10-fold cross validation to evaluate how bugs detected by XEDITOR match the known bugs in our data sets. In the second experiment, with 57 rules extracted from D1, we applied XEDITOR to different versions of deployment descriptors in D2, and validated the reported bugs and fixes based on later versions in software history or developers' feedback.

4.4.1 Experiment Based on D1. We split the 4,248 DD into 10 portions $P = \{p_1, p_2, \ldots, p_{10}\}$, and conducted 10-fold cross validation to evaluate XEDITOR's effectiveness of bug detection and fix. In each fold, we fed XEDITOR with nine portions of data for rule inference (e.g., including $p_1, p_2, \ldots, p_9$); we used the remaining one portion of data (e.g., p_{10}) to create a test set, and then relied on the test data to assess XEDITOR's effectiveness of rule application. Among the 10 folds of validation, we rotated data and ensured that each portion was used exactly once for test data creation.

Specifically, to build a test set with a data portion p_i ($i \in [1, 10]$), we first applied Antlr to convert each deployment descriptor $f \in p_i$ to a parsing tree t, and then extracted entities from t. If there is any entity pair $c = (e_1, e_2)$ demonstrating a rule $r = A \rightarrow B$ which belongs to the 22 confirmed rules in Section 4.3, we removed B's match (e.g., e_2) from the tree to get a different tree t'. In this way, if XEDITOR works successfully, it should be able to (1) infer rule r from the other nine portions of data, and (2) use that rule to report a bug (e.g., missing B's match e_2) when scanning t'. With the created test sets, we compared all bugs reported by XEDITOR against the known bugs in modified parsing trees, and evaluated the precision, recall, and accuracy accordingly.

Table 5 presents our experiment results for the 10-fold cross validation. In the table, **# of Rules Inferred** shows how many rules XEDITOR inferred from the nine data portions for each round. **# of Correctly Inferred Rules** reports how many of the inferred rules are actually correct according to our manual analysis. **# of Bugs Reported** presents the number of bugs XEDITOR detected in the test set. **# of Known Bugs** shows the number of bugs we introduced by modifying parsing trees and removing certain entities. **# of Correctly Reported Bugs** counts the reported bugs that match our ground truth. **Precision**, **Recall**, and **Accuracy** reflect the effectiveness of XEDITOR's bug detection capability. For instance, in round 1, XEDITOR inferred 25 rules from the given data portions, while 20 of these rules are true positives. Based on all inferred 25

Table 6: The 15 real XML bugs fixed by developers

Bug Index	Violated Rule ($A \rightarrow B$)	Root Cause Category	Fixing Strategy	Vdiff(fix, bug)
1	*_beans_bean_property_ref → *_beans_bean_id	Delete B only	Import an XML file	2
2	*_web-app_security-constraint_auth-constraint_role-name → *_web-app_security-role_role-name	Update the data of A while B's data is unchanged	Update the data of B	1
3	*_web-app_security-constraint_auth-constraint_role-name → *_web-app_security-role_role-name	Update the data of A while B's data is unchanged	Update the data of B	1
4	*_web-app_security-constraint_auth-constraint_role-name → *_web-app_security-role_role-name	Insert A only	Insert B	4
5	*_web-app_security-constraint_auth-constraint_role-name → *_web-app_security-role_role-name	Insert A only	Insert B	4
6	*_beans_bean_property_ref → *_beans_bean_id	Insert A only	Delete A	2
7	*_beans:beans_beans:bean_beans:property_ref → *_beans:beans_beans:bean_id	Insert A only	Delete A	1
8	*_beans:beans_beans:bean_beans:property_ref → *_beans:beans_beans:bean_id	Delete B only	Delete A	5
9	*_web-app_servlet-mapping_servlet-name → *_web-app_servlet_servlet-name	Insert A and B with different string literals used	Update the data of A	3
10	*_web-app_servlet-mapping_servlet-name → *_web-app_servlet_servlet-name	Delete B only	Insert B	1
11	*_web-app_servlet-mapping_servlet-name → *_web-app_servlet_servlet-name	Insert A only	Insert B	150
12	*_web-app_servlet_servlet-name → *_web-app_servlet-mapping_servlet-name	Insert A only	Delete A	6
13	*_web-app_servlet-mapping_servlet-name → *_web-app_servlet_servlet-name	Insert A only	Delete A	1
14	*_web-app_servlet-mapping_servlet-name → *_web-app_servlet_servlet-name	Insert A only	Delete A	1
15	*_web-app_servlet_servlet-name → *_web-app_servlet-mapping_servlet-name	Insert A and B with different string literals	Update the data of B	3

rules, XEDITOR detected 304 bugs in the test set, although there are 283 known bugs in the set. The intersection between bug reports and our ground truth is 271 bugs. Thus, XEDITOR achieved 89% precision, 96% recall, and 92% accuracy for round 1.

In each round, given 9 portions of data, XEDITOR inferred 20-26 def-use like rules, and 17-21 of these rules are true rules. Based on all inferred rules, XEDITOR reported 258-322 bugs, and 240-292 of them are true bugs. Among the 10 rounds, on average, XEDITOR achieved 92% precision, 96% recall, and 94% accuracy for bug detection. Two reasons can explain why XEDITOR could not achieve 100% accuracy. First, when XEDITOR obtained false positives for rule inference, the incorrectly inferred rules misled XEDITOR to produce false positives for rule application and thus report false bugs. Second, some true rules have insufficient supporting instances in the selected nine data portions. Consequently, these rules cannot be inferred by XEDITOR, neither can XEDITOR detect the known bugs in the test set for these rules. In other words, the false negatives in rule inference cause false negatives in rule application.

Finding 3 (Response to RQ2): *XEDITOR detected bugs in XML files with 92% precision, 96% recall, and 94% accuracy.*

4.4.2 Experiment Based on D2. With the 57 true rules revealed in Section 4.3, we applied XEDITOR to the second data set D2 to evaluate whether XEDITOR can (1) detect any real bug in updated XML files, and (2) suggest fixes that correspond to developers' actual fixes in reality. 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 bug detected, we manually examine versions after C in the repository to decide whether the reported bug was ultimately fixed by developers. If so, the reported bug is a real bug, and we can further compare XEDITOR's fix suggestion against the actual fix applied by developers.

By manually checking the bug reports generated by XEDITOR, we identified 25 really problematic XML updates. Interestingly, 15 of these 25 bugs were later fixed by developers according to the version history. This observation means that developers did introduce the XML bugs that XEDITOR can detect, and XEDITOR is capable of revealing developers' mistakes when they edited DD.

Finding 4 (Response to RQ4): *XEDITOR revealed 15 XML bugs in open-source software repositories. Our observation indicates the importance of XEDITOR, because developers did introduce the XML bugs that XEDITOR could detect during XML file maintenance.*

Table 6 presents the 15 bugs that developers later fixed. Specifically, column **Bug Index** shows the index we assigned to each bug. **Violated Rule** shows the rule XEDITOR used to identify the bug. **Root Cause Category** explains how developers introduced the bug. **Fixing Strategy** describes how developers resolved the bug in a later version. **Vdiff(fix, bug)** describes the version difference between the bug-fixing commit and bug-introducing one. Let us take the first bug as an example. The violated rule is `*_beans_bean_property_ref → *_beans_bean_id`. In one commit C_i, developers deleted the consequent entity (B) while keeping the antecedent (A). In a later commit C_{i+2}, developers fixed the bug by importing an XML file where B defines the literal used by A. Therefore, Vdiff(fix, bug) = (i+2) - i = 2.

Table 7: The 10-fold cross validation for BASELINE's effectiveness of bug detection

Id	Rule Inference		Rule Application					
	# of Rules Inferred	# of Correctly Inferred Rules	# of Bugs Reported	# of Known Bugs	# of Correctly Reported Bugs	Precision	Recall	Accuracy
1	23	18	252	272	248	98%	91%	95%
2	17	12	260	322	257	99%	80%	88%
3	21	16	225	287	222	99%	77%	87%
4	21	16	270	296	266	99%	90%	94%
5	21	16	254	288	248	98%	86%	92%
6	22	17	246	365	245	100%	67%	80%
7	20	15	230	321	227	99%	71%	82%
8	20	15	241	261	236	98%	90%	94%
9	21	16	238	343	234	98%	68%	81%
10	23	18	221	352	217	98%	62%	76%

According to Table 6, four bugs (i.e., 4^{th}, 5^{th}, 10^{th}, and 11^{th}) were fixed via the insertion of B. Developers fixed another six bugs (i.e., 6^{th}, 7^{th}, 8^{th}, 12^{th}, 13^{th}, and 14^{th}) by deleting A. Four bugs (i.e., 2^{nd}, 3^{rd}, 9^{th}, and 15^{th}) were fixed via data updates to A or B. One bug (i.e., 1^{st}) was fixed when developers imported another XML file for the data value used by A. Among all these applied fixes, 10 fixes are covered by the strategies suggested by XEDITOR (Insert B or Delete A). It means that XEDITOR usually suggests helpful fixes.

Additionally, for 6 out of the 15 bugs, developers applied fixes in the immediate next commit; they fixed the remaining bugs after at least two commits. Most interestingly, developers fixed one bug (i.e., 11^{th}) after 150 commits. These observations imply that if developers 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 5 (Response to RQ5): *XEDITOR's change suggestions match developers' actual fixes for 10 of the 15 detected XML bugs.*

4.5 Comparison with Baseline

Prior change suggestion tools mine software version history for *co-change* patterns, and use those patterns to identify any missing change [26, 27, 32, 40, 49, 53]. 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 its related entity (e.g., *B*) is not, ROSE suggests the missing change [53]. These tools are similar to XEDITOR due to their adoption of ARM, but different by relying on the *co-changes* instead of *coexistence* 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 two methodologies, i.e., coexistence-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 our customized association rule mining to infer rules of the format "$Change(A) \rightarrow Change(B)$". 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 are frequently co-changed within the same files and (ii) they usually refer to the same string literal.

Experiment Setting. As with XEDITOR, BASELINE also has four parameters: *supp*, $conf$, vth, and pth. For fair comparison, we tuned parameters in the manner described in Section 4.3 and found a reasonably good default setting for BASELINE: $supp = 5$, $conf = 0.65$, $vth = 0.8$, and $pth = 8$. Afterwards, we evaluated BASELINE using 10-fold cross validation based on the data set $D1$.

Results. Table 7 presents BASELINE's effectiveness of bug detection in the 10-fold cross validation experiment. According to the table, in each round, BASELINE infers 17-23 rules, while 12-18 of these inferred rules are correct. When detecting bugs based on all inferred rules, BASELINE obtained 98%-100% precision, 62%-91% recall, and 76%-95% accuracy. On average, BASELINE detected bugs with 98% precision, 78% recall, and 87% accuracy. Compared with XEDITOR's effectiveness shown in Table 5, on average, BASELINE achieved higher precision (98% vs. 92%), lower recall (78% vs. 96%), and lower accuracy (87% vs. 92%). These results indicate that XEDITOR and BASELINE achieved different trade-offs between precision and recall. When software practitioners choose an approach to use for XML debugging, they can either choose BASELINE for higher precision and lower false positive rates, or select XEDITOR for higher recall and lower false negative rates.

Two reasons can explain the above-mentioned comparison results. First, BASELINE analyzes the evolution history of 4,288 DD, while XEDITOR examines only one version of these files. Actually, the frequent co-changes indicate stronger relevance between entities than recurring coexistence. Namely, if two entities are often changed together in the same file, they definitely coexist; nevertheless, if two entities often coexist in the same file, they do not have to be changed or get frequently changed together. Consequently, the rules inferred from co-changed entities are generally more precise. Second, When two closely related entities are barely changed in version history, BASELINE cannot infer any correlation between them, neither can it predict any co-changes for those entities. Consequently, BASELINE obtained lower recalls.

Finding 6 (Response to RQ3): *Compared with BASELINE, XEDITOR detected bugs with lower precision (92% vs. 98%), higher recall (96% vs. 78%), and higher accuracy (92% vs. 87%). These two approaches made distinct trade-offs between precision and recall.*

5 THREATS TO VALIDITY

Threats to External Validity. All inferred rules and detected bugs mentioned in this paper 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.

Threats to Construct Validity. In our 10-fold cross validation for XEDITOR's effectiveness of bug detection and suggestion, for each fold, we automatically generated bugs by removing some XML entities from DD. These bugs may not represent the real bugs introduced by developers during software maintenance, so our empirical measurements can be biased. In the future, we will construct data sets with real bugs in DD to better evaluate XEDITOR's capability of bug detection and suggestion.

Threats to Internal Validity. Our manual analysis for the output by XEDITOR is subject to human bias and limited to our domain knowledge. To mitigate the problem, we had two authors to examine the rules reported by XEDITOR for agreement. The two authors agreed with each other in most scenarios. When they disagreed upon certain rules mainly because of their limited domain knowledge, they shared relevant documentation or examples with each other for discussion until coming to an agreement. Among the 25 really buggy XML updates detected by XEDITOR, there are 10 bugs not fixed by developers. We sent emails to the owner developers to check whether any of these 10 bugs is true. So far, we have only received one response email, which pointed out that the violation shown in Listing 3 is a false positive. As we gather more comments from developers, we will further improve the quality of inferred rules and change suggestions.

6 LESSONS LEARNED

We conducted customized association rule mining to extract def-use like XML rules in a domain-agnostic way. Our approach is based on the insight that "*if two entities frequently coexist in the same file and often hold the same data, they are correlated*". Our evaluation shows that XEDITOR reveals some interesting rules that can effectively capture the XML bugs caused by human errors. More importantly, by manually analyzing the false positives produced by XEDITOR, we also learnt four research challenges in this area.

First, *related entities do not always coexist in the same deployment descriptor.* We observed that some correlated entities are defined in two separate XML files; their relationship is established when (1) one XML file imports the other file or (2) a Java annotation in source code specifies both files. For instance, for the rule

`*_beans_bean_property_ref → *_beans_bean_id`,

we found 1,196 occurrences of `ref` in D2, and 49 of them correspond to the `id` defined in another XML file. To resolve such issues, we will conduct cross-file association rule mining by treating multiple XML files and Java files as a whole and analyzing them simultaneously.

Listing 3: A file that violates one of our inferred rules but is considered to be valid by its owner developer [4]

```
1  ...
2  <beans xmlns="http://www.springframework.org/... >
3    <mongo:db-factory dbname="${mongo.database.name}" mongo
         -ref="mongo" />
4    <bean id="mongoTemplate" class= ... >
5      <constructor-arg name="mongoDbFactory" ref="
           mongoDbFactory" />
6    </bean>
7  </beans>
```

Second, *some related entities may never refer to the same value.* Listing 3 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 → *_beans_bean_id`,

Listing 3 has a rule violation and should include another bean defined with the id `mongoDbFactory`. However, from our email conversation with the developer and relevant documentation [41], 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`. Such delicate constraints are currently not inferable by XEDITOR, because the specification does not align well with our insight. To extract such constraints, more heuristics and domain-specific insights are needed for better approach design.

Third, *some related entities may not cooccur frequently enough.* Currently, XEDITOR adopts multiple parameters (i.e., *supp* and *pth*) to refine inferred rules based on the cooccurrence frequency between entities. According to our experience, there are def-use rules that XEDITOR could not identify simply because the cooccurrence rates are low. This limitation is commonly shared among all probability-based rule inference approaches. More novel techniques are still needed to reveal rules based on rare entity occurrences.

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. Researches may need better techniques to propose more fixing strategies automatically and to fully automate XML repair.

7 RELATED WORK

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

7.1 Validation of Metadata

Several approaches were proposed to help check and/or fix the usage of metadata (i.e., XML and annotations) [20, 22, 23, 25, 30, 37, 43, 45]. 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 [20] 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 use 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. [37] 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 [43]. However, this approach focuses on the relations between metadata and code; it does not handle any editing constraint within XML files.

7.2 Program Change Prediction

Researchers built tools to mine version histories for co-change patterns, and used those patterns to predict any missing change [26, 27, 49, 53]. Specifically, Gall et al. mined release data for the co-change relationship between subsystems [26] and classes [27]. Zimmerman et al. and Ying et al. further extracted the co-change relationship between finer-grained program entities (e.g., classes, methods, and fields) [49, 53]. 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) [28, 35, 50]. 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, these approaches also mine history for co-changes and rank entities accordingly. Given a new commit, these approaches combine the two ranked lists in distinct ways to reveal any missing change. However, the effectiveness of these approaches are also limited by the frequency of co-changed entities.

Several researchers used the syntactic relationship between entities or files to predict changes [33, 42, 46]. 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 [42]. Given a changed file, the model predicts what other files to change together. Wang et al. characterized the common field accesses and/or common method invocations between co-changed methods [47]. Based on the characterization, Wang et al. built CMSuggester, a tool to predict methods for change given an added field or method and one or more changed methods [33, 46]. However, these approaches only analyze source code; they are not applicable to non-code artifacts like XML files.

7.3 Traceability Management

Software artifact traceability means "the ability to follow the life of a requirement in a forward and backward direction" [29]. Maintaining traceability across software artifacts helps ensure the coevolution of artifacts [38]. For instance, when a high-level requirement document is changed, traceability helps locate the pieces of design or code which should also be changed. Various information retrieval approaches were proposed to reveal traceability links mainly between requirement documentation and other types of artifacts (e.g., design documents and source code) [21]. Additionally, Kagdi and Maletic built a tool to analyze commits in a software version histories and to mine for highly frequently co-occurring changes to different artifacts [34]. Lozano et al. built MaTraca—a tool that supports users to specify traceability links across domains via logic predicates. With users' specifications, MaTraca checks the links between (1) entity definitions in one domain (e.g., Java) and (2) entity usage in another (e.g., XML). However, none of these tools examine the coevolution patterns within the same XML file.

7.4 Configuration Debugging

Several tools were built to diagnose or fix software configuration errors [18, 19, 39, 48, 52]. 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 [52]. By comparing the behavioral differences between two types of runs in terms of recorded predicates, ConfDiagnoser identifies the candidate 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 [48].

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

8 CONCLUSION

Deployment descriptors are hard to create and maintain, because there are domain-specific constraints on the XML formats defined by different software libraries. In this paper, we built Xeditor—an approach to automatically infer the def-use like configuration couplings in DD. By applying the inferred rules to a given XML file, Xeditor can identify any rule violation to report bugs and provide suggestions. Similar to prior rule mining approaches, Xeditor also leverages association rule mining to infer rules based on statistics. However, different from prior work, Xeditor (1) infers rules from DD instead of source code, (2) relies on the common data shared between XML entities to locate candidate pairs, (3) mines rules based on the coexistence instead of co-changes between entities, and (4) adopts more filters to refine the mined rules.

Our evaluation reveals interesting phenomena. First, the effectiveness of Xeditor is sensitive to its parameter settings. With appropriate configuration, Xeditor was able to infer rules with high precision (73%). Second, Xeditor could detect bugs in DD with high precision (92%), high recall (96%), and high accuracy (94%). Third, Xeditor could identify real bugs in DD, which were actually later fixed by developers. This demonstrates the usefulness of Xeditor and the necessity of similar static analysis tools for DD.

There is still significant space for future improvements in DD-related rule inference and application. Our research currently focuses on def-use like configuration couplings because we observed such rules in various software frameworks. However, it is still unknown what major types of DD bugs exist in real software systems and how def-use bugs compare with other bug categories in terms of the occurrence rates and severity. As the future work, we will conduct an empirical study on DD bugs in open-source projects.

ACKNOWLEDGMENT

We thank reviewers for their insightful comments. This work was supported by NSF grants CCF-1845446 and CNS-1929701, and Beijing Natural Science Foundation No. 4192036.

REFERENCES

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

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

[3] GitHub. https://github.com.

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

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

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

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

[8] Viewing Deployment Descriptors. https://www.ibm.com/support/knowledgecenter/en/SS7K4U_9.0.5/com.ibm.websphere.zseries.doc/ae/trun_app_deploymtdesc.html.

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

[10] XML Syntax. https://www.w3schools.com/xml/xml_syntax.asp.

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

[12] Manual Wiring Functional. https://github.com/yholkamp/addressbook-sample-jpa/commit/726bf62be03eac0e8292362340b117a7e10dd611, 2012.

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

[14] How to Correctly Manage Feature Configuration Deployment in JBoss Fuse 6.2.1? https://stackoverflow.com/questions/39706237, 2016.

[15] ANTLR. https://www.antlr.org, 2020.

[16] Gumtreediff/gumtree. https://github.com/GumTreeDiff/gumtree, 2020.

[17] Securing Web Applications. https://docs.oracle.com/javaee/7/tutorial/security-webtier002.htm, 2020.

[18] 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.

[19] 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.

[20] 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.

[21] M. Borg, P. Runeson, and A. Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, 2014.

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

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

[28] 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.

[29] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, 1994.

[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] Z. Jiang, Y. Wang, H. Zhong, and N. Meng. Automatic method change suggestion to complement multi-entity edits. *Journal of Systems and Software*, 159:110441, 2020.

[34] H. Kagdi and J. Maletic. Software repositories: A source for traceability links. *TEFSE/GCT'07*, 2007.

[35] 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.

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

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

[38] R. Oliveto, G. Antoniol, A. Marcus, and J. Hayes. Software artefact traceability: the never-ending challenge. In *IEEE International Conference on Software Maintenance, ICSM*, pages 485 – 488, 11 2007.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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\discretionary{-}{}{}on-control.

[45] J. W. W. Wan and G. Dobbie. Extracting association rules from XML documents using XQuery. In *Proceedings of the 5th ACM International Workshop on Web Information and Data Management*, WIDM '03, pages 94–97, New York, NY, USA, 2003. Association for Computing Machinery.

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

[47] Y. Wang, N. Meng, and H. Zhong. An empirical study of multi-entity changes in real bug fixes. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 287–298, 2018.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

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