Detecting Metadata-Related Bugs in Enterprise Applications

MD MAHIR ASEF KABIR, Virginia Tech, USA
XIAOYIN WANG, The University of Texas at San Antonio, USA
NA MENG, Virginia Tech, USA

When building enterprise applications (EAs) on Java frameworks (e.g., Spring), developers often configure application components via *metadata* (i.e., Java annotations and XML files). It is challenging for developers to correctly use metadata, because the usage rules can be complex and existing tools provide limited assistance. When developers misuse metadata, EAs become misconfigured, which defects can trigger erroneous runtime behaviors or introduce security vulnerabilities. To help developers correctly use metadata, this paper presents (1) RSL—a domain-specific language that domain experts can adopt to prescribe metadata checking rules, and (2) MeCheck—a tool that takes in RSL rules and EAs to check for rule violations.

With RSL, domain experts (e.g., developers of a Java framework) can specify metadata checking rules by defining content consistency among XML files, annotations, and Java code. Given such RSL rules and a program to scan, MeCheck interprets rules as cross-file static analyzers, which analyzers scan Java and/or XML files to gather information and look for consistency violations. For evaluation, we studied the Spring and JUnit documentation to manually define 15 rules, and created 2 datasets with 115 open-source EAs. The first dataset includes 45 EAs, and the ground truth of 45 manually injected bugs. The second dataset includes multiple versions of 70 EAs. We observed that MeCheck identified bugs in the first dataset with 100% precision, 96% recall, and 98% F-score. It reported 152 bugs in the second dataset, 49 of which bugs were already fixed by developers. Our evaluation shows that MeCheck helps ensure the correct usage of metadata.

CCS Concepts: • **Software and its engineering → Specification languages**; **Interpreters**; *Software maintenance tools*; Parsers.

Additional Key Words and Phrases: Domain-specific language, metadata, XML, annotation, consistency

ACM Reference Format:
Md Mahir Asef Kabir, Xiaoyin Wang, and Na Meng. 2025. Detecting Metadata-Related Bugs in Enterprise Applications. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE053 (July 2025), 23 pages. https://doi.org/10.1145/3715772

1 Introduction

Enterprise application (EA) development is a complex process of creating large-scale, multi-tiered, scalable, reliable, and secure network applications for business purposes [65]. To reduce the complexity of EA development, developers usually build EAs on top of the Java EE platform [13] or third-party frameworks like Spring [14]. Such platforms or frameworks promote the principle of separation of concerns [58]: they address nonfunctional concerns including persistence, transactions, and security, so that developers only need to implement the core functionality of an EA by hand. Most of these platforms or frameworks support a declarative programming model, allowing EA developers to use metadata (i.e., Java annotations and XML files) when configuring (1) how application components are deployed, and (2) how these components interact with each other.

Authors' Contact Information: Md Mahir Asef Kabir, mdmahirasefk@vt.edu, Virginia Tech, Blacksburg, Virginia, USA; Xiaoyin Wang, The University of Texas at San Antonio, San Antonio, Texas, USA, xiaoyin.wang@utsa.edu; Na Meng, Virginia Tech, Blacksburg, Virginia, USA, nm8247@vt.edu.

ACM 2994-970X/2025/7-ARTFSE053
https://doi.org/10.1145/3715772

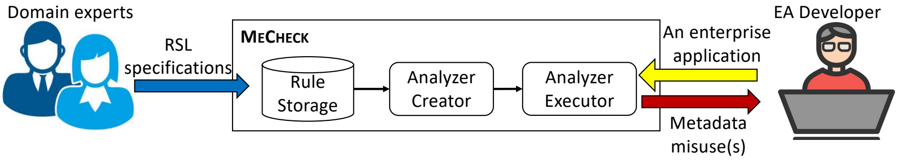


Fig. 1. The workflow of our approach

However, correctly using metadata can be challenging for developers due to three reasons. First, the usage rules are domain-specific and vary with Java frameworks, so developers can easily get confused and misuse metadata. Second, when a deployment or configuration task poses consistency constraints on the content of (i) Java annotations, (ii) code implementation, and/or (iii) XML files, developers may fail to observe all constraints when evolving software and thus inconsistently update metadata or code. Third, existing tools [54, 67, 70] rarely analyze metadata together with Java code, let alone find semantic inconsistencies between software artifacts. Consequently, when metadata misuse leads to any misconfiguration [6], abnormal runtime behavior [8], confusing error [1], or security vulnerability [9], developers are on their own to handle metadata-related issues.

To help developers debug metadata usage, we invented a semi-automatic solution that has two parts: **RSL (Rule Specification Language)** and **MeCheck (Metadata Checker)**. As shown in Fig. 1, to define a metadata-usage rule for a Java framework (e.g., Spring), domain experts (e.g., owner developers of that framework) can exploit RSL to (i) retrieve all relevant metadata and/or code items, (ii) refine those items based on certain conditions, (iii) constrain the content correspondence among refined items, and (iv) prescribe the error-reporting format if any constraint is violated. MeCheck stores such RSL specifications locally as domain-specific metadata checking rules. When an EA developer provides a software application to check, MeCheck loads all rules, creates parsing trees based on the RSL grammar for those rules, and treats the trees as analyzers. When executing each analyzer, MeCheck parses Java and XML files on demand, gathers and filters metadata and/or code items as instructed, and compares the item content for consistency checking. If any item violates a constraint, a customized error message is reported.

To evaluate the effectiveness of RSL and MeCheck, we defined and investigated three research questions (RQs) in our experiments:

- **RQ1**: *How effectively can RSL express metadata-usage rules?* We studied the documentation of Spring and JUnit frameworks [16, 17, 68], and distilled 15 metadata-usage rules. Seven of the rules are about content consistency between **XML items (i.e., elements and attributes)** and Java code; six rules are about consistency checking between code and annotations; one rule checks the consistency between XML and annotations; and one rule examines the consistency among code, XML items, as well as annotations. We managed to express all rules using RSL, demonstrating its great power in expressing diverse rules.
- **RQ2**: *How accurately can MeCheck detect bugs?* We manually injected 45 metadata-related bugs into the latest version of 45 open-source projects, and applied MeCheck to those projects. Our evaluation shows that MeCheck reported bugs with 100% precision, 96% recall, 98% F-score. This implies that MeCheck detected bugs with high accuracy.
- **RQ3**: *How effectively does MeCheck reveal real-world bugs?* We applied MeCheck to the version history of another 70 open-source projects, in order to reveal metadata-related bugs in real-world settings. In total, MeCheck reported 152 bugs in the version history of 21 projects, 117 of which bug reports are true positives. Developers have fixed 49 of those bugs so far. Our experiment indicates that MeCheck effectively identified real bugs in EAs. If developers had adopted MeCheck to scan their projects before committing any program changes, they could have found and addressed metadata-related issues more easily.

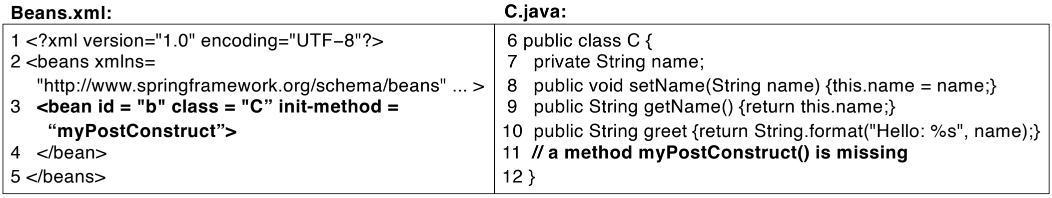
In summary, this paper makes the following contributions:

- We designed a domain-specific language (DSL)—RSL—for domain experts to specify metadata-usage rules. Different from prior DSLs, RSL can express consistency relations among XML items, Java annotations, and Java code.
- We created MeCheck, to interpret RSL specifications and examine user-provided EAs accordingly. Compared with existing tools, MeCheck implements a novel algorithm that (1) extracts data in both Java and XML files, (2) tracks relations among the extracted data, and (3) differentiates between data instances for refinement and comparison.
- We comprehensively evaluated RSL and MeCheck. Our evaluation demonstrates the great expressiveness of RSL and the high detection accuracy of MeCheck for synthetic data; it also reveals real-world scenarios where MeCheck effectively locates metadata-related bugs.
- To optimize MeCheck's runtime performance, we implemented a caching mechanism in the tool, which computes new data only when necessary.

In the following sections, we will explain our research with a motivating example (Section 2), introduce the background knowledge of metadata usage (Section 3), describe our new approach: RSL and MeCheck (Sections 4–5), and present our evaluation (Section 6).

2 A Motivating Example

This section uses an example to intuitively explain our research. The software framework Spring [14] supports developers to *configure beans to have initialization and cleanup methods* [21]. Namely, a "bean" is any plain-old Java object that follows standard configuration patterns; it can be defined in Java or XML files. Suppose that an XML file declares bean `b` as an instance of Java class `C`, and sets the bean's attribute "`init-method`" or "`destroy-method`" to a Java method name. Then the method must exist in the corresponding Java class `C`. This is because when the attribute is set, Spring automatically calls the corresponding method during runtime to either initialize `b` after the bean is created, or perform destruction tasks before the bean is destroyed.



Beans.xml:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns=
    "http://www.springframework.org/schema/beans" ... >
3   <bean id = "b" class = "C" init-method =
      "myPostConstruct">
4   </bean>
5 </beans>
```

C.java:

```
6 public class C {
7   private String name;
8   public void setName(String name) {this.name = name;}
9   public String getName() {return this.name;}
10  public String greet {return String.format("Hello: %s", name);}
11  // a method myPostConstruct() is missing
12 }
```

Fig. 2. A bean object in XML has attribute `init-method` refer to a Java method in the corresponding class

As shown in Fig. 2, the exemplar XML file sets the attribute `init-method` of bean `b` to "`myPostConstruct`" (line 3). Ideally, EA developers should define a corresponding method in the exemplar Java class—`myPostConstruct()`. However, when EA developers fail to do so (see line 11), no existing compiler or static checker can reveal such errors of omission. As a result, developers can only observe the consequence of this metadata-related bug during program execution, and then manually diagnose root cause for the triggered runtime error `org.springframework.beans.factory.BeanCreationException`.

Listing 1. An RSL specification for detecting missing methods

```
Rule method–exists {
  for (file xml in getXMLs()) {
    for (<bean> bean in getElms(xml, "<bean>")) {
      String beanClassFQN = getAttr(bean, "class");
      if (classExists(beanClassFQN)) {
        class c = locateClassFQN(beanClassFQN);
```

```
7        for (String s in getAttrs(bean, "*method")) {
8          assert(exists(method m in getMethods(c))(getName(m) == s)) {
9            msg("The referenced method: %s in bean: %s is not defined in class: %s", s, getName(bean), getFQN(c))
                   ;}}}}}
```

To help developers detect metadata-related bugs earlier and more easily, we developed RSL for framework developers to prescribe metadata-checking rules. Specifically, to statically detect the missing `init-method` mentioned above, framework developers can define an RSL rule. As shown in Listing 1, intuitively, the rule describes four major things:

- **What metadata/code items are involved?** Given a software project (i.e., EA), locate all $\langle bean \rangle$ objects defined in XML files (see lines 2–3).
- **How are these items refined?** Associate the $\langle bean \rangle$ objects with their corresponding Java class declarations, and focus on such $\langle bean \rangle$-class pairs (lines 4–6).
- **What is the consistency constraint?** For attribute `<*method>` of any $\langle bean \rangle$ (i.e., `<init-method>` or `<destroy-method>`), a matching method should be defined in the related class (lines 7–8).
- **What is the error message?** When the constraint is violated and there is a <*method>-attribute value not matching any method definition, a bug should be reported (line 9).

Our new tool MeCheck can take in the above-mentioned RSL rule to statically check EAs. It can reveal the $\langle *method \rangle$ misconfiguration shown in Listing 1, before EA developers run that program.

3 Background

To facilitate comprehension, this section will explain the three alternative ways of metadata-based EA configuration (Sections 3.1–3.3), and our research problem (Section 3.4).

3.1 XML-Based Configuration

There is a special kind of XML files named **deployment descriptors (DDs)** [19], which are frequently used in EAs for configuration purposes. According to the XML syntax, **XML elements** are the basic building blocks of XML files [20]. Each element is used as a container to store text content, other XML elements, or attributes. The syntax of XML elements is:

`<element-name attributes> Contents ...</element-name>`

The multiple attributes of any element are separated by white spaces, and each attribute associates the attribute name with a string value. For simplicity, we use **XML items** to refer to XML elements and their attributes. For instance, Fig. 3 (a) shows an exemplar DD that defines two bean objects: `infoMessage` and `messageRenderer` (see lines 3–6). The second bean has a `<constructor-arg>` element (line 6), which references the first bean as the value of its attribute `ref`.

3.2 Annotation-Based Configuration

Recently, more and more Java frameworks provide annotations as a substitute for deployment descriptors [15, 18]. Annotations allow EA developers to specify component configurations within Java classes. An annotation can be attached to a Java class, method, or field. When annotations have attributes, the related syntax can be "`@annotation-name("value")`" or "`@annotation-name(key = {values})`", where **key** is the attribute name. Fig. 3 (b) presents an annotation-based alternative to the DD shown in Fig. 3 (a). Specifically, **`@Configuration`** (line 9) implies that the current Java class is equivalent to a DD. Inside the class, **`@Bean`** (lines 11 and 15) is a direct analog of the XML `<bean>` element. **`@Bean`** means that the Java method it annotates will return an object that should be registered as a bean in the application context; the registered bean name is the method name (e.g., `infoMessage`). The second bean references the first bean by invoking `infoMessage()` (lines 15–18).

(a) XML-based configuration

```
1 <?xml version="1.0" encoding="UTF−8"?>
2 <beans xmlns=
    "http://www.springframework.org/schema/beans" ... >
3   <bean id = "infoMessage" class="InfoMessage"/>
4   <bean name="messageRenderer”
class="MessageRenderer">
5     <constructor−arg name="message" ref="infoMessage"/>
6   </bean>
7 </beans>
```

(b) Annotation-based configuration

```
 8 import org.springframework.context.annotation.∗;
 9 @Configuration
10 public class AppConfig {
11   @Bean
12   public InfoMessage infoMessage() {
13     return new InfoMessage();
14   }
15   @Bean
16   public MessageRenderer messageRenderer() {
17     return new MessageRenderer(infoMessage());
18   } }
```

(c) Configuration based on both XML and annotations

```
my−bean.xml:
19 <?xml version="1.0" encoding="UTF−8"?>
20 <beans xmlns=
     "http://www.springframework.org/schema/beans" ... >
21   <bean id = "infoMessage" class="InfoMessage"/>
22 </beans>

AppConfig.java:
23 import org.springframework.context.annotation.∗;
24 @Configuration
25 @ImportResource(location = {"classpath:my−bean.xml"})
26 public class AppConfig {
27   @Bean
28   public MessageRenderer messageRenderer() {
29     ApplicationContext context = new
         AnnotationConfigApplicationContext(
         ApplicationConfig.class);
30    return new MessageRenderer(context.getBean(
        "infoMessage"));
31   } }
```

Fig. 3. Three alternative ways to implement the same bean configuration

3.3 Combining XML Files with Annotations

With both XML- and annotation- based configuration methods available, developers can take a hybrid approach between the two. Namely, given an EA, developers can configure part of the application with XML files and configure the remaining with annotations; the two types of configuration interact with each other in specialized ways. For instance, Fig. 3 (c) implements the XML-based configuration presented by Fig. 3 (a) with two files: my-bean.xml defines the bean object `infoMessage` (line 21), and AppConfig.java defines the other bean `messageRenderer` (lines 27–31). The Java class uses `@ImportResource` to import the XML file and to access all beans defined there.

3.4 Problem Statement

As reflected by the examples so far, there are diverse metadata-usage rules that EA developers should follow. These rules can vary with the adopted software frameworks, developers' configuration needs, and configuration methods; many of the rules require for content consistency among code elements, XML items, and annotations. It is challenging for EA developers to correctly memorize and follow all domain-specific rules; it is even harder for developers to consistently maintain the metadata usage scattered in multiple files. A recent study on StackOverflow [60] shows that numerous developers posted questions on how to properly configure frameworks. There is a desperate need for approaches and tools that can help with metadata debugging. To create an automatic static checker that can satisfy developers' need, we encountered two technical challenges: (1) how to cope with the diversity of domain-specific rules, and (2) how to extract and relate the content from different software artifacts to validate metadata usage?

4 Rule Specification Language (RSL)

To overcome the challenges mentioned above, we designed a DSL—RSL—for framework developers to specify metadata-usage checking rules. The RSL rules will serve two purposes. First, they can help EA developers (i.e., framework users) understand the content consistency among metadata and code items. Second, they will be sent to MeCheck for automatic detection of metadata-related bugs. As illustrated in Fig. 4, the RSL syntax defines statements, expressions, and built-in functions.

4.1 Statements

RSL supports five types of statements; each statement contains simple or complex expressions, or other statements. With these statement types, an RSL specification or rule describes four important aspects of the usage-checking logic: extracting relevant metadata/code items, refining or filtering items, checking for consistency, and reporting errors.

(1) **ForStmt** means `for`-loop, used to describe *what metadata/code items to extract and enumerate in a software application.* Users can adopt a single or nested loop to structure the extraction and handling of all items potentially relevant to a constraint. As shown in Listing 1, `for(file xml in getXMLs()){...}` means to get all XML files in the project, iterate over those files using the variable `xml`, and process each iteration as instructed by the `for`-loop body.

(2) **IfStmt** is conditional `if`-statement. It describes *how to refine items or locate items of interest.* Namely, when an iterated item satisfies the specified `if`-condition, it is processed by the body. For instance, in Listing 1, `if(classExists(beanClassFQN)){...}` checks whether a Java class with the same fully-qualified name as `beanClassFQN` exists in the source code. If so, the body first locates that class via `class c = locateClassFQN(beanClassFQN)`, and then checks content correspondence between the Java class and XML file. Otherwise, the `bean`-object is skipped for further processing. This is because when the class is not defined in source code (e.g., defined by a third-party library or by another languaged program), the content checking can become overly complicated or even infeasible; thus, we decided to quit checking on the `bean`-object to avoid false positives.

(3) **AssertStmt** means assertion, to describe *what constraint to check.* Each statement has two parts: a condition and the body. The condition is a simple or complex expression, to define constraints or predicates that items must satisfy. The `assert`-body is defined with a MsgStmt (see below), to express action-to-take when the condition is not satisfied. In Listing 1, the assertion means that given an attribute value of `<*method>`, there should be a Java method with the specified name.

(4) **MsgStmt** is message statement to describe *what error message to report when a bug is detected.* People can use MsgStmt to compose an error-message template, and offer expressions or values to instantiate the template. The MsgStmt in Listing 1 incorporates a Java method name, bean name, and class name to pinpoint the issue of a misconfigured `<init-method>` or `<destroy-method>`.

(5) **DeclStmt** is declaration statement—an auxiliary statement to facilitate rule definition. Such a statement declares a variable with its data type, and initializes the variable using an expression. With such statements, recurring expressions only need to be evaluated once, as the evaluated value can be passed to a user-defined variable and that variable can get used to replace multiple occurrences of the same expression. For instance, Listing 1 has a DeclStmt to declare variable `c` that holds the located Java class item, given a `bean`-class name specified in XML.

```
Specification := Rule Id Body
         Body := ′{′Stmt Stmt∗′}′
         Stmt := ForStmt | IfStmt | AssertStmt | DeclStmt ′;′
      ForStmt := for ′(′ Type Id in Exp ′)′ Body
       IfStmt := if ′(′ Exp ′)′ Body
   AssertStmt := assert ′(′ Exp ′)′ ′{′ MsgStmt ′;′ ′}′
      MsgStmt := msg ′(′ ′,′ SimExp (′,′ SimExp) ∗ ′)′
     DeclStmt := Type Id ′ =′ Exp
          Exp := SimExp | SimExp AND Exp | SimExp OR Exp |NOT Exp
       SimExp := Id | Lit | FunctionCall | ′(′ Exp ′)′ | FunctionCall ′ ==′ SimExp | exists ′(′ Type Id in Exp ′)′ ′(′ Exp ′)′
         Type := ′⟨′ Id ′⟩′ | file | class | method | field | String
          Lit := StringLit | CharLit | IntLit | FloatLit
 FunctionCall := Id ′(′ Params ′)′
       Params := SimExp (′,′ SimExp)∗
```

Fig. 4. Core syntax of RSL

Table 1. Built-in functions in RSL

Category	Functions
Code-related functions	`callExists`, `classExists`, `getArg`, `getClasses`, `getConstructors`, `getFamily`, `getFields`, `getFQN`, `getMethods`, `getName`, `getReturnType`, `getSN`, `getType`, `hasField`, `hasParam`, `hasParamType`, `indexInBound`, `isIterable`, `isLibraryClass`, `isUniqueSN`, `locateClassSN`, `locateClassFQN`
Annotation-related functions	`getAnnoAttr`, `getAnnoAttrNames`, `getAnnotated`, `hasAnnotation`, `hasAnnoAttr`
XML-related function	`elementExists`, `getAttr`, `getAttrs`, `getElms`, `getXMLs`, `hasAttr`
Miscellaneous functions	`endsWith`, `isEmpty`, `indexOf`, `join`, `pathExists`, `substring`, `startsWith`, `upperCase`

4.2 Expressions

RSL defines various expressions. There are six kinds of simple expressions: identifier, literal, invocation of built-in function(s), parenthesized expression, equivalence-checking for simple expressions, and `exists`-clause. In particular, the `exists`-clause has two parts: a header and the body. The header `exists (Type Id in Exp)` describes what items to enumerate, while the body `(Exp)` describes a constraint to satisfy by any item. This clause is similar to ForStmt, as it also describes *what items to extract and enumerate*. However, different from ForStmt, this clause does not have to process all items in a given set; it can exit early and return true whenever finding an item to satisfy the specified constraint. In addition to simple expressions, RSL also supports three kinds of complex expressions, which connect simple expressions via logical operators AND, OR, and NOT.

4.3 Built-in Functions

As shown in Table 1, we defined four types of built-in functions to facilitate (1) data extraction from software artifacts and (2) content comparison among items.

(i) **Code-related functions** support data extraction from either raw Java code or processed code items (i.e., classes, fields, and methods). For instance, `classExists(String fqn)` checks whether the project defines a Java class, whose fully qualified name is `fqn`; `getClasses()` retrieves all classes defined by a Java project; `getFamily(class c)` retrieves all ancestors in the class hierarchy of a given class `c`, together with `c` itself; `hasParam(method m, String aName)` checks whether method `m` has a parameter named `aName`; `indexInBound(method m, int idx)` checks whether a Java method has at least `idx+1` parameters. Given a code element, `getName(...)` returns the element's simple name, `getFQN(...)` returns the fully qualified name, and `getType(...)` returns the type binding.

(ii) **Annotation-related functions** extract information from annotations or annotated Java code. Specifically, `getAnnoAttr(class c, String anno, String attr)` returns the value of a specified attribute `attr`, of annotation `anno` used in Java class `c`; `getAnnoAttrNames(class c, String anno)` retrieves all attribute names of annotation `anno` in the given class `c`; `getAnnotated(String anno, String type)` retrieves all code items, which are decorated by the specified annotation and are of certain kind (e.g., Java class); `hasAnnoAttr(class c, String anno, String attr)` checks whether the specified class has annotation `anno`, and whether that annotation has the attribute `attr`.

(iii) The **XML-related functions** support data extraction from XML files, and the processing of XML elements or attributes. As shown in Listing 1, `getXMLs()` returns all XML files in the project; `elementExists(xml, "<bean>")` examines whether the given file has an XML element named as `<bean>`; `getElms(xml, "<beans>")` returns all <bean>-elements in the given file; `getAttr(bean, "class")` returns the value of `class`-attribute for `bean`; `getAttrs(bean, "*method")` retrieves values of `bean`'s attributes, whose names match the specified regular expression pattern.

(iv) **Miscellaneous functions** define operations applicable to Strings or Lists. For instance, `upperCase(String s)` capitalizes all letters in String `s`; `join(List...values)` concatenates lists of items to create a larger list; `pathExists(String path)` checks whether a given path exists in the file system.

5 MeCheck

MeCheck conducts static program analysis, to check whether metadata is used correctly. As shown in Fig. 1, when being applied to a Java enterprise application, MeCheck (i) loads RSL rules, (ii) creates analyzers from those rules, and (iii) executes analyzers to detect bugs. This section focuses on steps (ii) and (iii) of the procedure.

5.1 Analyzer Creator

Given RSL specifications, this component produces parsing trees that we refer to as **analyzers**, because they reflect the logic of statically analyzing EAs to detect metadata-related bugs. We leveraged JavaCC [31] to implement the creator. Specifically, after we provided (1) token patterns defined in regular expressions and (2) syntax grammar defined in extended Backus-Naur form (EBNF), JavaCC derived a lexical analyzer (scanner) from the token patterns and created a parser from the grammar. The generated scanner and parser could create a parsing tree given an RSL rule.

5.2 Analyzer Executor

When developing Analyzer Executor, we encountered three challenges (C1–C3):

C1. **Semantics**: The five statement types supported by RSL execute in different ways, so our executor must observe the differences when interpreting their semantics.
C2. **Scoping**: If an RSL rule define multiple variables using the same name, the executor should differentiate between the scopes of variables for correct static analysis.
C3. **Performance**: When multiple rules require MeCheck to repetitively collect and process data in the same way (e.g., scanning all XML files to gather `<bean>`-elements), we need to reduce or even eliminate redundant computation to optimize performance.

To overcome all challenges, we created an executor of analyzers (i.e., parsing trees) as a visitor to traverse tree nodes. When accessing each node, the executor collects and processes program data on demand; its traversal manner varies with the node type. For implementation, we used JavaParser [59] to parse Java code and extract data from the resulting parsing trees. We also used JDOM [4] to parse XML files and extract data. In this section, we will introduce our novel *context-aware interpretation* algorithm to address C1 (Section 5.2.1), the specially designed data structures to overcome C2 (Section 5.2.2), and a novel caching mechanism to address C3 (Section 5.2.3).

5.2.1 Context-Aware Interpretation Algorithm.

As shown in Algorithm 1, given the parsing tree of analyzer t and the software-under-analysis P, MeCheck implements a *read-eval* loop to traverse statement-level nodes in a top-down manner, execute statements in sequence, and report metadata-related bugs based on the logic reflected by t (see lines 1.3–1.4). In particular, MeCheck adopts a stack S to keep track of the *execution context*, i.e., all variables defined and their scopes. During execution, MeCheck creates and pushes a frame f before executing all statement-level child nodes under t (line 1.2), but pops and destroys f after executing those statements (line 1.5). Algorithms 2–4 reflect how MeCheck works differently when executing different kinds of statements.

ForStmt Execution. As shown in Algorithm 2, MeCheck creates and pushes a frame f (line 2.1), before executing a ForStmt's header and body in an iterative way. A typical way of defining the ForStmt header is: "`for (T v in func(...))`". Given such a header, MeCheck calls `func(...)` to extract or gather data, and then uses a variable `v` of type `T` to enumerate data items. As shown in lines 2.5–2.9, in each loop iteration, MeCheck first adds one entry `{T, v, e}` to f, to associate `v` with its data type `T` and enumerated value `e` for that iteration; it then executes statement-level child nodes in sequence; afterwards, it clears frame f to remove all variables locally defined by or for that iteration (line 2.9), since the values of those variables are limited to that iteration. After executing the entire `for`-loop, MeCheck removes f to discard all variables locally declared (line 2.10).

IfStmt Execution. As shown in lines 3.3–3.9 of Algorithm 3, MeCheck evaluates the `if-condition` expression, and proceeds to the statement's body when that expression is true. Furthermore, before executing the body, MeCheck creates and pushes a frame f; it then executes statements inside the body sequentially; it also pops and discards f after body execution.

The Execution of AssertStmt and MsgStmt. As shown in Algorithm 4, MeCheck first evaluates the `assert`-condition. If that condition is false, MeCheck formulates a bug report based on the MsgStmt embedded in the `assert`-body.

Algorithm 1: The **main()** function in our algorithm

```
Input: $t$—root node of the rule/analyzer, and $P$—the given software application
Output: Reported metadata-related bugs
1.1 Initialize a stack $S$ for variable frames
1.2 Create a frame $f$ and push it onto $S$
1.3 foreach statement node $c$ in $t$'s body do
1.4     $process(c, S, P)$
1.5 pop $f$ from $S$
```

Algorithm 2: The **processFor()** function

```
Input: $n$—statement node, $S$—stack for variable frames, and $P$—the given software application
Output: Reported metadata-related bugs
2.1  Create a frame $f$ and push it onto $S$
2.2  $T \leftarrow$ variable type part from $n$'s header
2.3  $v \leftarrow$ variable name part from $n$'s header
2.4  $container \leftarrow evaluate($container part from $n$'s header, $S, P)$
2.5  foreach element $e$ in $container$ do
2.6      add $\{T, v, e\}$ to $f$
2.7      foreach statement node $c$ in $n$'s body do
2.8          $process(c, S, P)$
2.9      clear the frame $f$
2.10 pop $f$ from $S$
```

Algorithm 3: The **process()** function

```
Input: $n$—statement node, $S$—stack for variable frames, and $P$—the given software application
Output: Reported metadata-related bugs
3.1  if $n$ is ForStmt then
3.2      $processFor(n, S, P)$
3.3  else if $n$ is IfStmt then
3.4      $res \leftarrow evaluate(n$'s expression, $S, P)$
3.5      if $res$ is True then
3.6          Create a frame $f$ and push it onto $S$
3.7          foreach statement node $c$ in $n$'s body do
3.8              $process(c, S, P)$
3.9          pop $f$ from $S$
3.10 else if $n$ is AssertStmt then
3.11     $processAssert(n, S, P)$
3.12 else if $n$ is DeclStmt then
3.13     $f \leftarrow$ top of $S$
3.14     $T \leftarrow$ variable type part of $n$
3.15     $v \leftarrow$ variable name part of $n$
3.16     $value \leftarrow evaluate($variable value part of $n, S, P)$
3.17     add $\{T, v, value\}$ to $f$
```

Algorithm 4: The **processAssert()** function

```
Input: $n$—statement node, $S$—stack for variable frames, and $P$—the given software application
Output: Reported metadata-related bugs
4.1 $res \leftarrow evaluate(n$'s expression, $S, P)$
4.2 if $res$ = False then
4.3     $msg \leftarrow evaluate($message part of the assert statement, $S, P)$
4.4     $print(msg)$
```

DeclStmt Execution. As shown in lines 3.12–3.17 of Algorithm 3, MeCheck retrieves the current top frame of stack S. It identifies both the type name `T` and variable name `v` used in the declaration; it also evaluates the right-hand side of the statement for `value`. Finally, MeCheck adds an entry `{T, v, value}` to f to record the declared variable. In this way, when the variable is used later, MeCheck revisits f to retrieve the variable's type as needed, and to read or write values on-demand.

Expression Evaluation. Algorithm 5 illustrates how expressions are processed and executed. Intuitively, when an expression is a variable or literal, MeCheck simply returns the value as the evaluation result. When an expression is more complex and contains multiple sub-expressions or nested expressions, MeCheck traverses the expression tree in a top-down manner, and evaluates values in a bottom-up manner. For instance, to evaluate the expression `locateClassFQN(getAttr(bean,` `"class")`, MeCheck first retrieves the value of `bean` by accessing the stack; it then evaluates the value of function call `getAttr(bean, "class")`; based on that evaluation, MeCheck further calculates the value of function call `locateClassFQN(...)`.

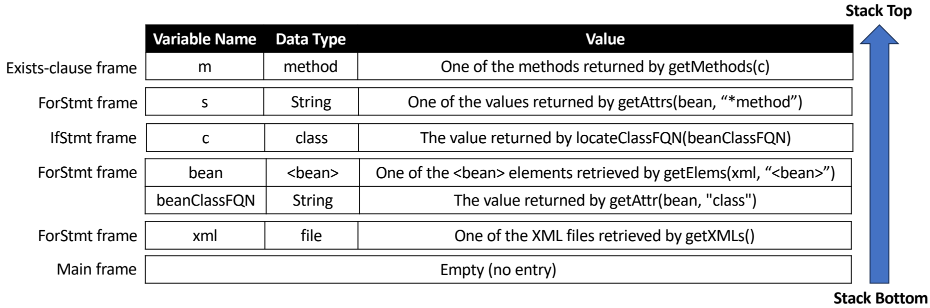
5.2.2 *Stack and Frames.*

MeCheck adopts a stack S to keep track of the execution context; it (1) pushes frames onto S, (2) pops frames from S, and (3) walks through frames to manage variables, their types, as well as their values. A **frame** is a dictionary, where the key is a variable name, and the value is a pair of variable type and variable name. MeCheck creates a frame before executing the entire analyzer, any ForStmt, IfStmt, or `exists`-clause; each created frame is then pushed onto stack to record any variable declared for or in the corresponding program structure. During execution, MeCheck refers to the stack to search for declared variables, query their data types, or obtain the values. After the execution of a ForStmt, IfStmt, exists-clause, or the entire analyzer, MeCheck pops a frame from the stack to discard all variables created by that program structure.

Algorithm 5: The **evaluate()** function

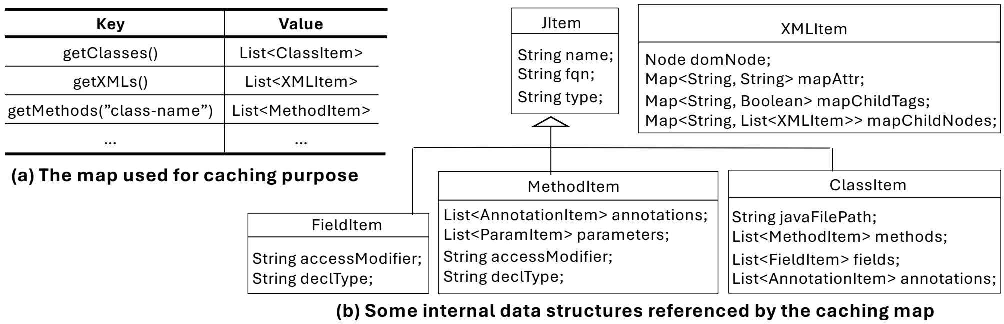
```
Input: $n$—expression node, $S$—stack for variable frames, and $P$—the given software application
Output: Return a boolean value based on the expression evaluation
if $n$ is an identifier then
    $res \leftarrow$ retrieve the value of $identifier$ from $S$ starting from top frame
    $return\ res$
else if $n$ is a literal then
    $return$ the literal's value
else if $n$ is a built-in function call then
    $return$ the function call's return value
else if $n$ is a parenthesized expression then
    $return\ evaluate$(the expression inside the parenthesis, $S, P$)
else if $n$ is an equivalence-checking expression then
    $return\ evaluate$(left-side of the expression, $S, P$) == $evaluate$(right-side of the expression, $S, P$)
else if $n$ startsWith "exists" then
    Create a frame $f$ and push it onto $S$
    $T \leftarrow$ variable type part from $n$
    $v \leftarrow$ variable name part from $n$
    $container \leftarrow evaluate$(container part from $n, S, P$)
    foreach element $e$ in $container$ do
        add $\{T, v, e\}$ to $f$
        $logic \leftarrow$ expression inside the exists-expression's body
        $res \leftarrow evaluate(logic, S, P)$
        if $res$ = True then
            $return$ True
        clear the frame $f$
    pop $f$ from $S$
    $return$ False
else if $n$ is an AndExpression then
    $return\ evaluate$(left-side of the expression, $S, P$) && $evaluate$(right-side of the expression, $S, P$)
else if $n$ is an OrExpression then
    $return\ evaluate$(left-side of the expression, $S, P$) || $evaluate$(right-side of the expression, $S, P$)
else if $n$ is a NotExpression then
    $return\ evaluate$(the expression after NOT operator, $S, P$)
```

Fig. 5 presents a snapshot of stack S during runtime when MeCheck executes the `exists`-clause of analyzer in Listing 1. Variable entries are added to frames when the variables are declared, or



	Variable Name	Data Type	Value
Exists-clause frame	m	method	One of the methods returned by getMethods(c)
ForStmt frame	s	String	One of the values returned by getAttrs(bean, "*method")
IfStmt frame	c	class	The value returned by locateClassFQN(beanClassFQN)
ForStmt frame	bean	<bean>	One of the <bean> elements retrieved by getElems(xml, "<bean>")
	beanClassFQN	String	The value returned by getAttr(bean, "class")
ForStmt frame	xml	file	One of the XML files retrieved by getXMLs()
Main frame	Empty (no entry)		

Fig. 5. The stack status when MeCheck executes the exists-clause in Listing 1



Key	Value
getClasses()	List<ClassItem>
getXMLs()	List<XMLItem>
getMethods("class-name")	List<MethodItem>
...	...

(a) The map used for caching purpose

(b) Some internal data structures referenced by the caching map

Fig. 6. The map used for caching and some related internal data structures in our implementation

when iterating variables for ForStmt and `exists`-clause are initialized at the beginning of each loop iteration. Some entries get removed from frames at the end of loop iterations, because the values of iterating variables get reset for each iteration and the local variables declared for one iteration are invalid for other iterations. Given a variable name for resolution, e.g., `c` used in the `exists`-body, MeCheck starts with the top frame to search for a corresponding variable declaration. As shown in Fig. 5, the top frame has no variable declared as `c`, so MeCheck moves on to the next frame—the ForStmt frame—to search. Such a stack walking continues until MeCheck finds the variable declared in a IfStmt frame. Our stack-based data structure allows the analysis-logic implemented in inner scopes to freely access variables defined in outer scopes; it also enables MeCheck to differentiate between the same-named variables declared in distinct scopes.

5.2.3 Performance Improvement. When MeCheck executes multiple analyzers to scan the same project in one run, it is possible that the analyzers call same functions and repetitively retrieve the same results. Such repetitive function calls can waste lots of computing resources, hindering MeCheck from analyzing software efficiently. What makes things even worse, based on our experience, many domain-specific constraints or RSL analyzers require MeCheck to parse all Java classes by calling `getClasses()`, or parse all XML files by calling `getXMLs()`. When a software project is large and contains thousands of files, the repetitive file-parsing can be very time-consuming.

To optimize performance, we implemented a caching mechanism in MeCheck to eliminate redundant computation as much as possible. Specifically, we defined a global cache to map function calls to their return-values, for each software-under-analysis. For instance, as shown in Fig. 6 (a), `getClasses()` is mapped to the list of class items MeCheck creates based on Java parsing; `getXMLs()` is mapped to a list of XML items MeCheck creates based on XML parsing; `getMethods("class-name")` is mapped to a list of method items created for all methods declared by the Java class named "`class-name`". We decided to cache a function call if that function extracts data from Java/XML files, or derives information by enumerating extracted data. With such a caching mechanism, for each project-under-analysis, MeCheck parses all Java (or XML) files only once if any of the analyzers needs to retrieve all code (or metadata) items; when a function is called with different parameter values, their results can be cached differently as those values are used as part of the key in cache. Furthermore, to avoid eagerly extracting all methods and fields from Java files when MeCheck is only interested in the class-level information, MeCheck conducts lazy initialization for both `fields` and `methods` members of `ClassItem` objects (see Fig. 6 (b)): either member is initialized only when any analyzer explicitly requests for the information of a particular class.

6 Evaluation

To assess the usefulness of our approach, we conducted three experiments and explored the following research questions (RQs):

- **RQ1**: How effectively can RSL express metadata-usage rules?
- **RQ2**: How accurately can MeCheck detect bugs?
- **RQ3**: How effectively does MeCheck reveal real-world bugs?

6.1 Effectiveness of RSL

To assess the expression power of RSL, we studied (1) the documentation of Spring [14]—the most popular application development framework for EAs in Java, and (2) documentation as well as tutorials on JUnit [30, 33, 38, 43]—the most popular framework for unit testing in Java. We manually distilled 15 metadata-usage constraints from those documents. As shown in Table 2, seven constraints are about the correspondence between XML and code (r_1–r_7); six constraints involve annotations and code ($r8$–$r13$); one constraint is on the correspondence between XML and annotation ($r14$); one constraint is relevant to XML, annotation, and code. These constraints are diverse in terms of the metadata/code items involved, the number of files scanned, and the consistency-checking logic. We successfully expressed constraint-checking rules in RSL, which fact evidences the great expression power of our domain-specific language. In the following experiments, we will leverage MeCheck to generate analyzers from these rules, and evaluate MeCheck's effectiveness in identifying buggy EAs that violate any of the rules.

Finding 1 (Responding to RQ1): *RSL has great expressiveness. It is capable of describing a variety of metadata-usage checking rules that involve XML, annotation, and/or code.*

6.2 Accuracy of MeCheck

To assess how accurately MeCheck can detect metadata-related bugs, we created a ground truth dataset of buggy EAs (Section 6.2.1), defined evaluation metrics (Section 6.2.2), and evaluated MeCheck using the dataset and metrics (Section 6.2.3).

6.2.1 A Dataset of Buggy EAs. We mined GitHub [28] for enterprise applications, using the heuristic-based approach proposed by prior work [70]. Specifically, we crawled GitHub for any project that contains at least one XML file, whose file path has any of the following keywords: "Spring", "security", "web", and "WEB-INF". If a project satisfies this requirement, it is likely to be an EA. Next, we refined the mined projects by removing irrelevant and redundant projects, getting 831 projects.

To create the dataset of buggy EAs, we injected 45 bugs in 45 different projects. To inject bugs for each constraint mentioned in Table 2, we leveraged the involved metadata items (e.g., "`@RunWith(Parameterized.class)`" or "`<property>`") as keywords to search for three different projects. We manually inspected the relevant metadata usage to ensure that each retrieved project does not violate the corresponding constraint. Then we modified the metadata to intentionally inject a bug for each of the projects. For the 15 rules, we retrieved and modified in total 45 (=15*3) projects.

Table 2. The 15 metadata-usage constraints we summarized

Id	**Rule Name**	**Constraint Summary**	**Involved Items**
r_1	xml-path-check	When constructor "`ClassPathXMLApplicationContext(String configLocation)`" is called, the provided argument should correspond to an existent XML file in the file system.	XML, code
r_2	bean-class-exists	Java classes mentioned by the `class`-attribute of <bean> should exist in the project, unless they are library classes.	XML, code
r_3	constructor-arg-type-field-map	When `<constructor-arg>` is specified as a sub-element of <bean>, its `type`-attribute value should match a constructor parameter's type name of the corresponding Java class.	XML, code
r_4	constructor-arg-name-field-map	The `name`-attribute value of `<constructor-arg>` should match a constructor parameter name in the corresponding Java class.	XML, code
r_5	constructor-index-out-of-bound	The `index`-attribute value of <constructor-arg> should fit into the index boundary of at least one constructor in the corresponding Java file.	XML, code
r_6	method-exists	When a <bean> has any of the following attributes configured: `init-method` and `destroy-method`, the attribute values should match the names of methods defined in corresponding Java classes.	XML, code
r_7	property-setter-map	The `name`-attribute values of `<property>` items should map to setter methods' names in the corresponding Java class. For instance, if the `name`-attribute of a `<property>` item is "`pn`", then there should be a setter named "`setPn`" in the corresponding Java class.	XML, code
r_8	runwith-no-parameters	If a Java class has "`@RunWith(Parameterized.class)`" annotation, then it should also have a method with "`@Parameters`" annotation.	annotation, code
r_9	runwith-no-test	If a Java class has "`@RunWith(Parameterized.class)`" annotation, then it should also have a method with "`@Test`" annotation.	annotation, code
r_{10}	runwith-no-suiteclasses	If a Java class has "`@RunWith(Suite.class)`" annotation, then the class should also have " `@SuiteClasses`" annotation.	annotation, code
r_{11}	suiteclasses-no-runwith	If a Java class has "`@SuiteClasses`" annotation, then the class should also have "`@RunWith(Suite.class)`" annotation.	annotation, code
r_{12}	suiteclasses-no-test	If a Java class is annotated with "`@SuiteClasses`" or "`@Suite.SuiteClasses`", then the Java class mentioned in the annotation attribute should (1) be decorated with either annotation, or (2) have a method in itself or any ancestor class satisfying any of the following requirement: (i) the method is annotated with `@Test`, (ii) the method name starts with "`test`", and (iii) the method name is "`suite`".	annotation, code
r_{13}	testParams-not-iterable	Each Java method with "`@Parameters`" annotation should have an iterable return type (e.g., List).	annotation, code
r_{14}	import-resource-path	When "`@ImportResource`" is used and its attribute "`location`" is configured, the attribute value should be a valid file path corresponding to an existent XML file in the file system.	XML, annotation
r_{15}	bean-exists	The Spring API "`ApplicationContext.getBean(String str)`" searches for a bean by name or by type. Search-by-type works when the argument `str` is a Java class name (i.e., ending with "`.class`"), and the class should be either (1) annotated with "`@Component`", "`@Service`", "`@Repository`", "`@Controller`", or "`@RestController`", or (2) mentioned by the `class`-attribute of any <bean> in XML. Search-by-name requires either a Java method annotated with "`@Bean`" is named with `str`, or a <bean> in XML has its `id`-attribute value as `str`.	XML, annotation, code

In this way, we injected 45 bugs to 45 distinct open-source projects, making each analyzer inside MeCheck have 3 bugs to find. We considered these 45 injected bugs as ground truth to evaluate the detection capability of MeCheck.

6.2.2 Metrics. We used three metrics to evaluate MeCheck's effectiveness for bug detection.

Precision (P) measures among all bugs MeCheck reported, how many of them are true positives:

$$P = \frac{\text{\# of correctly reported bugs}}{\text{Total \# of bug reports}}$$

Recall (R) measures among all known bugs, how many of them are reported by MeCheck:

$$R = \frac{\text{\# of correctly reported bugs}}{\text{Total \# of known bugs}}$$

F-score (F) is the harmonic mean of P and R; it reflects the bug-detection accuracy of MeCheck:

$$F = \frac{2 \times P \times R}{P + R}$$

All metrics mentioned above have values in [0%, 100%]; the higher the better. To compute P and R, we will intersect the set of bugs reported with the ground-truth bug set; any overlap between the two sets captures bugs correctly reported by MeCheck. Thus, the larger overlap there is, the better.

6.2.3 Experiment Results. MeCheck detected bugs with high precision (100%), high recall (96%), and high F-score (98%). In total, MeCheck reported 43 bugs, all of which match the ground truth of injected bugs. However, it missed two bugs related to the rule r_2 bean-class-exists, due to the design choice we made when implementing MeCheck.

Listing 2. The RSL rule of bean-class-exists

```
1 Rule bean−class−exists {
2   for (file xml in getXMLs()) {
3     if (elementExists(xml, "<bean>")) {
4       for (<bean> bean in getElms(xml, "<bean>")) {
5         String beanClassFQN = getAttr(bean, "class");
6         assert(classExists(beanClassFQN) OR isLibraryClass(beanClassFQN)) {
7           msg("Bean class: %s mentioned in bean: %s, does not exist", beanClassFQN, getName(bean)); }}}}}
```

With more details, r_2 ensures that if the fully qualified name of a Java class is mentioned as the `class`-attribute of any `<bean>`, the class should be either defined by the EA-under-analysis or a library on which EA depends. Listing 2 shows the RSL specification of bean-class-exists, which calls `callExists(...)` and `isLibraryClass(...)` on line 6. In MeCheck, `classExists(...)` is implemented to check whether the pass-in parameter matches any class item extracted from Java code. The implementation of `isLibraryClass(...)` defines a list of regular expressions (RegEx) to describe the naming patterns of frequently used library classes, such as "`^org\.hibernate\..+$`" for Hibernate classes. We crafted the regular expressions based on (1) our domain knowledge of widely used libraries, and (2) libraries we frequently observed in EAs. Next, `isLibraryClass(...)` examines whether a pass-in parameter matches any RegEx to determine if the class is defined by a library. We manually injected each of the two missed bugs, by specifying an invalid class name as the `class`-attribute of `<bean>`. Ideally, MeCheck should report both bugs because the `<bean>`-classes do not exist. However, since the invalid names we specified (e.g., `org.hibernate.search.hibernate.example.dao.impl.BookDaoImplChanged`) accidentally match predefined RegEx patterns, MeCheck incorrectly considered them to be valid and failed to locate both bugs.

We could have reduced such false negatives by discarding some RegEx patterns used in function `isLibraryClass(...)`. However, the downside of doing so is that many library classes will fail to match the remaining patterns. Consequently, MeCheck will wrongly interpret those classes as non-library classes, and report false positives when it also fails to find those classes defined by EA. In another word, we may get false positives when trying to overcome such false-negative issues. We prefer generating precise bug reports as developers may not want to get frequently bothered with false positives. Thus, we designed MeCheck to include more RegEx patterns and report bugs more precisely, instead of improving recall rates at the cost of sacrificing precision.

Finding 2 (Responding to RQ2): *MeCheck demonstrated great detection accuracy when being applied to our 45-project dataset with injected bugs. It found bugs with 100% precision, 96% recall, and 98% F-score.*

6.3 Detection of Metadata-Related Bugs in Real-World Settings

To assess how well MeCheck reveals real bugs, we also created a dataset of 70 real-world open-source software repositories (Section 6.3.1), and applied MeCheck to that dataset (Section 6.3.2).

6.3.1 A Dataset of Real-World Software Repositories. To create this dataset, we started with the 831 projects initially mined from GitHub (see Section 6.2.1). From those projects, we removed 85 projects that are irrelevant to any of the 15 rules implemented in MeCheck. Namely, if a project does not contain any code/XML/annotation item involved in those rules, we consider it irrelevant. Afterwards, we removed the 45 projects used for creating buggy EAs (see Section 6.2.1), to avoid doing multiple experiments on the same projects. Naïvely, we could experiment with all program versions, for each of the remaining 701 projects, to check whether any version has metadata-related bugs and violates our predefined rules. However, it would take too much time to analyze all versions of each project, so we decided to only experiment with a subset of the pool. To ensure the representatives of our experiment and results, we decided to include projects of distinct scales (i.e., small, medium, and large projects). Thus, we ranked the 701 projects in ascending order of Java file counts, as file counts roughly reflect project sizes. We randomly sampled 10 projects for every 100-project interval, and got 70 projects.

In our resulting dataset, each project has 1–987 Java files and 5–1068 total files; the mean value of Java files in each project is 68; the median value of Java files is 24.

6.3.2 Experiment and Results. We applied MeCheck to different versions of each selected project, to thoroughly explore whether developers committed any mistakes when maintaining metadata and related Java code. Because many projects have long version histories, it can be very time-consuming to apply MeCheck to all versions of each project. Therefore, to accelerate the bug detection procedure, we developed scripts to filter versions, and focus our experiments on versions that edit any Java or XML file containing code/annotation/XML items relevant to the 15 rules.

Our results show that among the selected 70 software repositories, MeCheck reported in total 152 bugs in 21 projects. After manually inspecting the bug reports, we found 117 reports to be true positives, as they violate either r_1 (i.e., xml-path-check) or r_2 (i.e., bean-class-exists). In particular, 18 bugs were detected in the first commits of software repositories; 99 bugs were found in later commits, meaning that developers accidentally introduced the bugs when they revised software for maintenance. 115 of the 117 bugs were reported together with bugs of the same kind. The existence of multiple bugs in the same program version does not affect our evaluation results, as bugs are analyzed and reported independently. Furthermore, we noticed that 49 of the 117 true positives were already fixed by developers. We found that out by checking later versions of the same projects and by observing revision of those buggy programs. The remaining 68 bugs had not been fixed yet, so we filed bug reports to contact developers and seek for their feedback. So far, we have not heard from developers for those bug reports. There are 35 false positives in the 152 bug reports, because the RegEx patterns we used to identify library classes do not cover all the libraries used by EAs.

Table 3 presents the 49 bugs that developers later fixed. All these bugs violate r_2—the bean-class-exists rule. Namely, each of the bugs references a nonexistent class when declaring a bean. In Table 3, column **Idx** shows the index we assigned to each buggy project. **Project Name** lists

Table 3. The 49 real bugs later fixed by developers

Idx	Project Name	# of Bugs	Ddiff(bug, fix) (days)	Vdiff(bug, fix)	Time cost per version (seconds)
1	aioweb [29]	9	1–2	1–2	4–6
2	angular-js-spring-mybatis [24]	1	0	1	0
3	biyam_repository [27]	2	194	4	2
4	cv-web [42]	2	44–47	1–2	0
5	enterprise-routing-system [45]	2	0	1	5–6
6	FileExplorer [25]	1	0	1	3
7	generica [46]	2	0	1	2
8	I377-esk [23]	1	0	1	7
9	jarvis [40]	4	9–10	4–5	2–6
10	johnsully83_groovy [32]	10	23	2	17
11	Kognitywistyka [35]	8	0	1–3	3–4
12	LIBRARY [26]	2	0	1	2
13	rop [34]	3	5–556	2–27	7–9
14	ShcUtils [41]	1	1	1	0
15	spring-vaadin [36]	1	10	1	0

the projects where the bugs occurred. **# of Bugs** counts the total number of bugs we found in different versions of each project. **Ddiff(bug, fix)** describes the day difference between committing dates of a bug-fixing version and the related bug-introducing version. Similarly, **Vdiff(bug, fix)** describes the version difference between the bug-fixing version and buggy version. Let us take the second project angular-js-spring-mybatis [24] as an example. In one commit C_i checked in on Jan 29, 2014, developers declared a bean by wrongly referencing a nonexistent class. In a later commit C_{i+1} checked in on the same day, developers fixed the bug by correcting the class reference. Therefore, Vdiff(bug, fix) = (i+1) - i = 1, and Ddiff(bug, fix) = 0.

For only 17 of the 49 bugs, developers applied fixes on the same day. However, for the remaining 32 bugs, developers applied fixes either on the following day or after quite a long time. 15 of the bugs we detected were fixed 1-10 days after the bug-introducing commits; 17 of the bugs were fixed more than 10 days after the bug-introducing commits. The largest time difference we observed between a bug-fixing commit and a buggy version is 556 days. The phenomena imply the difficulty of revealing those bugs, and the necessity of our approach. Additionally, for 24 of the 49 bugs, developers applied fixes in the immediately next commit. However, they fixed the remaining bugs after at least two commits. Most interestingly, developers fixed a bug in rop [34] after 27 commits. These observations indicate the great benefits developers can potentially get out of the MeCheck usage. Namely, if developers had used MeCheck to examine their software before committing program changes, they should have avoided checking in the erroneous program changes, or even have fixed the introduced bugs earlier.

Finally, MeCheck has very low runtime cost. We experimented with a laptop that has (1) an Intel i7-8565U CPU with four cores and eight logical processors, and (2) 15.9 GB memory. As shown in Table 3, when MeCheck was applied to detect bugs based on the 15 rules, it spent no more than 6 seconds on each program version.

Finding 3 (Response to RQ3): *MeCheck demonstrated great effectiveness and high performance when being applied to different versions of 70 real-world open-source EAs. It reported in total 152 bugs; 117 of the bugs are true positives, 49 of which have been already fixed by developers*

7 Threats to Validity

Threat to External Validity. We leveraged RSL to express 15 rules summarized for Spring and JUnit, and applied MeCheck to in total 115 (45 + 70) open-source projects for bug detection. The rule set may be limited to our experience with Java frameworks, while the experiment results can be limited to our project datasets. In the future, we would like to include more rules and 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 bug reports, it is possible that our manual analysis are 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 bug detection.

Threat to Internal Validity. Currently, MeCheck determines whether a given class C is defined by a library dependency of EA using pattern matching. Namely, if the fully qualified name of C matches a RegEx pattern predefined in MeCheck, it is a library class. However, our RegEx pattern set may not be comprehensive; some library classes may fail the matching process and get wrongly treated as non-library classes. One way to overcome this limitation is to eagerly scan the bytecode of all library dependencies, trying to find an exact match for C's name. However, we decided not to perform such a heavyweight scanning for library classes because based on our experience, many open-source projects do not contain all JAR files for their library dependencies. Such a lack of dependency information will considerably limit the effectiveness of a heavyweight scanning and the applicability of MeCheck. In the future, we will investigate more advanced lightweight approaches to mitigate this issue.

8 Discussion

This section discusses the importance of our 15 rules, bug criticality, potential application scope of MeCheck, the necessity of defining new rules, and a potential alternative design as well as implementation of MeCheck.

8.1 Rule Importance and Bug Criticality

The 15 rules we investigated are important, because we extracted them from the documentation of Spring and JUnit. All the real bugs we revealed using those rules are critical and severe, as they all trigger runtime instead of compilation errors. It means that without any tool support, developers have to wait till the testing phase, to reveal those rule violations. Additionally, developers spent lots of time revealing some of the bugs we found. As mentioned in Section 6.3, 15 of the bugs we detected were fixed 1-10 days after the bug-introducing commits; 17 of the bugs were fixed more than 10 days after the bug-introducing commits. The phenomena imply the difficulty of revealing those bugs, and the necessity of our approach.

According to our experiment for Section 6.3, nevertheless, not all rules were violated by real-world projects. One possible reason can be that some developers fixed those violations before checking in commits. Even if we only observed violations of r_1 and r_2 in our own real-world experiment, the online resources listed in Table 5 show that people violated or are likely to violate the remaining rules. Thus, the rules are important as developers tend to violate them.

8.2 The Potential Application Scope of MeCheck

So far, we have defined 15 RSL rules based on the documentation of Spring and JUnit. However, the application scope of MeCheck is not limited to these 15 rules. To use MeCheck, developers can also define their own rules based on other metadata-related constraints [37] or their manual rule extraction from other libraries/frameworks. As XML and annotations have been prevalent configuration methods in various areas and well-known libraries/frameworks (see Table 4), we believe that MeCheck can be applied to a wider scope than what is demonstrated in this paper. Essentially,

Table 4. The potential areas and libraries/frameworks where MeCheck is applicable

Area	Exemplar Libraries or Frameworks
Enterprise Applications	JavaEE
Testing Frameworks	JUnit, TestNG
Dependency Injection and Application Frameworks	Spring Framework, Camel, Guice
Object-Relational Mapping	Hibernate, MyBatis
Web Development	Spring MVC, Struts, JSF
(De)Serialization	Jackson, Gson
Build Tools	Maven, Ant
UI Layouts	Android
Security	Spring Security
Microservice Frameworks	Quarkus, Micronaut

Table 5. The additional rule violations that get reported or discussed by developers

Rule	Violations Reported or Discussed
r_3	[2, 5, 22]
r_5	[3]
r_7	[10]
r_8	[12, 39]
r_{12}	[11, 49]
r_{13}	[39]
r_{14}	[47]
r_{15}	[7]

MeCheck is applicable to arbitrary Java projects that use either XML-based configuration only, annotation-based configuration only, or both XML-based and annotation-based configurations.

8.3 The Necessity of Defining New Rules

We foresee that developers are motivated to define new rules, when they use MeCheck to examine metadata usage. Three reasons help explain the motivation. First, many existing rules [37] are expressible with RSL and checkable by MeCheck. It indicates a strong need for developers to extend MeCheck's current 15-rule set to cover those known rules. Second, prior work [60] shows that when developers asked questions on StackOverflow concerning Spring security usage, the majority of questions are on metadata-based configurations. Such phenomena imply that (1) existing work provides insufficient tool support and (2) some delicate constraints are undocumented. Therefore, it is almost impossible to define a comprehensive ruleset to capture all rules in the wild today. Developers need to extend the ruleset after revealing previously unknown or hidden rules. Third, as XML and Java annotations are widely used for library/framework configurations, it is likely that future software deriving from these libraries/frameworks will inherit the configuration methods but define new domain-specific rules. Even if we can define a comprehensive ruleset for MeCheck today, as the time goes, new domain-specific rules appear; developers still need to expand MeCheck's ruleset to cover those rules. Therefore, it is necessary and important to define RSL for rule definition.

8.4 A Potential Alternative Design and Implementation of MeCheck

Existing static analysis tools like PMD [50] are created to detect bugs in Java code. Some readers may wonder why we did not extend those tools to define metadata-related rules. We thought about that option but decided to define our own DSL and create a new tool, mainly because the metadata-related rules we focus on are so unique that existing tools barely handle them. Take PMD as an example. PMD does not examine any of the rules listed in this paper. To extend the ruleset of PMD for XML analysis, people have to learn and use XPath—another DSL—to define queries on XML documents. However, XPath does not support queries on source code or annotations. It has a narrower scope than RSL. Furthermore, PMD is complex, with lots of implementation irrelevant to our focus, which can make our tool development on top of PMD very time-consuming and error-prone. To (1) avoid dealing with the complexity of PMD and (2) quickly prototype our research, we created MeCheck without reusing PMD or XPath.

9 Related Work

The related work of our research includes DSLs defined for metadata usage checking, detection of metadata-related bugs, and configuration debugging.

9.1 Domain-Specific Languages (DSLs) Defined for Metadata-Usage Checking

People invented DSLs to check and/or fix the usage of metadata (i.e., XML and annotations) [53–57, 61, 67]. 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 [54]. Similarly, CDuce [53] and XDuce [57] are independently developed DSLs for XML processing. To validate Java annotation usage, Eichberg et al. created a DSL for users to define constraints [56]. To check user-specified constraints, the researchers automatically converted Java bytecode to XML documents, and converted constraints to XQuery path expressions. Darwin [55] and Noguera et al. [61] also defined distinct DSLs for users to specify and validate the constraints on annotation usage. All DSLs mentioned above only focus on one type of metadata (i.e., either XML or annotations), but not on both types or on the relations between them.

Song and Tilevich created (1) MIL to express metadata invariants, and (2) the language implementation/engine to examine code-annotation relations (i.e., relations between Java code and annotations) and code-XML relations (i.e., relations between code and XML) [67]. We tried to run MIL engine in our experiments, albeit without success. RSL is different from MIL in three ways. First, it is an imperative instead of declarative language, so users have more flexibility in controlling how items are extracted, enumerated, filtered, and checked. Second, RSL has stronger expressiveness. It can express rules to examine XML-annotation-code constraints and XML-annotation constraints, but we did not find MIL capable of doing that. Third, RSL can have better performance, because we applied specialized optimizations in MeCheck to eliminate repetitive computation but MIL engine was unoptimized.

9.2 Detection of Metadata-Related Bugs

Researchers created tools to automatically detect metadata-related bugs [62, 63, 70, 71]. For instance, Wen et al. created XEDITOR to automatically infer and apply def-use like configuration couplings in XML files [70]. Specifically, XEDITOR extracts XML entity pairs that (i) frequently coexist in the same files and (ii) hold the same data at least once; it then applies customized association rule mining to infer def-use like couplings “A→B” between entities. For bug detection, given a new XML file, XEDITOR checks whether the file violates any couplings; if so, XEDITOR reports the violation(s). Noguera et al. created an extension of the Eclipse IDE’s refactoring engine, to check whether any Java code refactoring violates the correspondence constraints between existing code and annotations [62]. Nuryyev et al. devised a frequent-itemset based pattern mining approach to mine annotation-usage rules for MicroProfile, an open-source Java microservice framework [63]. By scanning 533 MicroProfile projects for violations of 12 of the mined rules, the researchers found 100 violations of 5 mined rules in 16 projects. Zhang et al. noticed that existing static program analyzers are unaware of the semantic changes introduced by annotations, and consequently can produce imprecise analysis results [71]. Thus, they conducted a study of annotation-induced faults (AIF) by analyzing 246 issues in 6 open-source and popular static analysis (i.e., PMD, SpotBugs, CheckStyle, Infer, SonarQube, and Soot). Based on their findings in the study, the researchers created a tool to generate new tests for static analyzers, and revealed 43 new faults, 20 of which have been fixed.

Our research complements prior work in two ways. First, it examines correspondence constraints in a wider scope, by querying and checking on the content correspondence in XML files, Java code, and annotations; nevertheless, existing tools only examine correspondence in one type of metadata. No existing tool examines the content constraints between XML and annotations, but MeCheck does so. Second, RSL enables people (e.g., library developers) to freely express domain-specific constraints for XML-based and annotation-based configurations. This feature is especially important when some domain-specific constraints are not detectable for any mining tool. Furthermore, when a constraint is mined by existing tools, people can easily extend MeCheck by defining an RSL specification for it, to embed the mined knowledge into automatic metadata analysis.

9.3 Configuration Debugging

Some tools were built to diagnose or fix software configuration errors [51, 52, 64, 66, 69, 72]. For instance, Attariyan et al. [51] and Zhang et al. [72] separately created tools to record predicates that may be affected by configuration options, collect the execution profiles of a program's correct and undesired runs, and compare the behavioral differences between the two types of runes to diagnose configuration errors. Rabkin and Katz created a static analysis-based tool to help users debug configuration errors in software [66]. The tool tracks the flow of configuration labels through a program: labels are introduced via configuration reads, and propagate via assignment and library calls. MeCheck produces a table mapping each line in the program's source code to the set of relevant configuration dependencies. This procedure is done by developers at release time. When a user encounters an error, they can use the error message to query this table.

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 [69]. Oh et al. introduce a model checking framework for building Kconfig static analysis tools [64]. The configuration specification language, Kconfig, is defined to prevent invalid configurations of the Linux kernel from being built. To detect bugs in Kconfig specifications, Oh et al. created a symbolic evaluator `kclause` to model Kconfig specifications, and a tool to find bugs in `kclause` models.

The configuration files examined by these tools are irrelevant to XML documents or Java annotations, so they cannot detect the bugs we focus on.

10 Conclusion

As with proper code implementation, correct metadata usage is always essential to software quality. Albeit the importance of high-quality metadata usage, widely used program analysis techniques (e.g., WALA [48] and Soot [44]) barely help with metadata debugging. Some existing tools can help developers identify certain metadata bugs, but the tool support is quite limited. In this paper, we developed RSL—a domain-specific language for describing metadata-usage checking rules, and built MeCheck—a tool to analyze programs based on RSL rules. Compared with existing tools, our approach is unique in three aspects. First, RSL enables library/framework developers to freely specify domain-specific constraints on the relations among code, annotations, and XML items; thus, it empowers MeCheck to enforce diverse rules in a wider scope and equips MeCheck with a strong power extensibility. Second, MeCheck extracts program data from diverse software artifacts (e.g., Java source files and XML files), and reasons about the relationship between the extracted data to detect bugs. It demonstrates a great way of integrating source code analysis with metadata analysis. Third, MeCheck applies optimizations to reduce redundant data processing, when trying to establish pattern matching between a given program and multiple bug patterns described by RSL rules. No prior work pays such a close attention to the time cost of metadata-related bug detection.

There is still significant space for future improvements in static analysis for metadata usage. In the future, we will investigate more domain-specific rules posed by different library frameworks, and extend MeCheck to conduct more advanced synergetic analysis among code and metadata.

11 Data Availability

Our programs and data are available at https://github.com/mahirkabir/detecting-metadata-bugs

Acknowledgments

We thank all reviewers for their valuable feedback. This work was partially funded by NSF-2006278, NSF-1929701, NSF-186467, NSF-2007718, and CCI.

References

[1] 2012. Securing REST urls with Spring. https://stackoverflow.com/questions/13836451/securing-rest-urls-with-spring
[2] 2014. BeanCreationException in spring controller. https://stackoverflow.com/questions/21520827/beancreationexception-in-spring-controller?rq=3.
[3] 2014. "Could not resolve matching constructor" error when passing in constructor-arg for child class. https://stackoverflow.com/questions/21583399/could-not-resolve-matching-constructor-error-when-passing-in-constructor-arg-f.
[4] 2014. Package javax.xml.parsers for processing of XML documents. Oracle Java API Documentation. https://docs.oracle.com/javase/8/docs/api/index.html?javax/xml/parsers/package-summary.html
[5] 2016. Could not resolve matching constructor.Ambiguity issue with Spring dependency injection. https://stackoverflow.com/questions/37648984/could-not-resolve-matching-constructor-ambiguity-issue-with-spring-dependency-in?rq=3.
[6] 2016. Custom Authentication Filters in multiple HttpSecurity objects using Java Config. https://stackoverflow.com/questions/37304211/custom-authentication-filters-in-multiple-httpsecurity-objects-using-java-config.
[7] 2016. org.springframework.beans.factory.BeanCreationException in Spring boot application. https://stackoverflow.com/questions/37938942/org-springframework-beans-factory-beancreationexception-in-spring-boot-applicati.
[8] 2016. 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-on-control
[9] 2016. 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-on-control.
[10] 2016. XML Based Configuration using Spring and Hibernate. https://stackoverflow.com/questions/39891823/xml-based-configuration-using-spring-and-hibernate.
[11] 2019. JUnit No Runnable Methods. https://examples.javacodegeeks.com/java-development/core-java/junit/junit-no-runnable-methods/.
[12] 2021. Getting error when running the code. https://www.qtpselenium.com/selenium-training/forum/7159/getting-error-when-running-the-code.
[13] 2021. Java EE at a Glance. https://www.oracle.com/java/technologies/java-ee-glance.html.
[14] 2021. Spring. https://spring.io.
[15] 2021. Spring - Annotation Based Configuration. https://www.tutorialspoint.com/spring/spring_annotation_based_configuration.htm.
[16] 2021. Spring Framework Documentation. https://docs.spring.io/spring-framework/docs/current/reference/html/.
[17] 2021. Spring Tutorial. https://www.tutorialspoint.com/spring/index.htm.
[18] 2021. Using Java EE Annotations and Dependency Injection. https://docs.oracle.com/cd/E11035_01/wls100/programming/annotate_dependency.html.
[19] 2021. XML Deployment Descriptors. https://docs.oracle.com/cd/A91202_01/901_doc/java.901/a90188/xml.htm.
[20] 2021. XML Elements. https://www.geeksforgeeks.org/xml-elements/.
[21] 2022. Spring – init() and destroy() Methods with Example. https://www.geeksforgeeks.org/spring-init-and-destroy-methods-with-example/.
[22] 2023. Ambiguous argument values for parameter of type [int]. https://blog.csdn.net/qq_49676677/article/details/119714497.
[23] 2024. . https://github.com/hannesnolvak/I377-esk.
[24] 2024. angular-js-spring-mybatis. https://github.com/rurutia/angular-js-spring-mybatis.
[25] 2024. brianleesg1/FileExplorer. https://github.com/brianleesg1/FileExplorer.
[26] 2024. dawe73/LIBRARY. https://github.com/dawe73/LIBRARY.
[27] 2024. gedkang / biyam_repository. https://github.com/gedkang/biyam_repository.
[28] 2024. Github. https://github.com/
[29] 2024. hopestar720/aioweb. https://github.com/hopestar720/aioweb.
[30] 2024. How to create JUnit Test Suite? (with Examples). https://www.browserstack.com/guide/junit-test-suite.
[31] 2024. JavaCC, The most popular parser generator for use with Java applications. https://javacc.github.io/javacc/
[32] 2024. johnsully83/johnsully83_groovy. https://github.com/johnsully83/johnsully83_groovy.
[33] 2024. JUnit - Suite Test. https://www.tutorialspoint.com/junit/junit_suite_test.htm.
[34] 2024. liuzhaomincoding / rop. https://github.com/liuzhaomincoding/rop.
[35] 2024. m2gikbb/Kognitywistyka. https://github.com/m2gikbb/Kognitywistyka.
[36] 2024. mcgray/spring-vaadin. https://github.com/mcgray/spring-vaadin.
[37] 2024. New Inspections in This Release | Inspectopedia Documentation. https://www.jetbrains.com/help/inspectopedia/.

[38] 2024. Parameterized (JUnit API). https://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html.

[39] 2024. Parameterized test class without data provider method. https://www.jetbrains.com.cn/en-us/help/inspectopedia/ParameterizedParametersStaticCollection.html.

[40] 2024. ReeverPD/jarvis. https://github.com/ReeverPD/jarvis.

[41] 2024. ShcUtils. https://github.com/535521469/ShcUtils.

[42] 2024. smyrouf/cv-web. https://github.com/smyrouf/cv-web.

[43] 2024. SOFTWARE TESTING: GETTING STARTED WITH ECLIPSE AND JUNIT (JUNIT 3). https://www.inf.ed.ac.uk/teaching/courses/st/2010-2011/tutorials/tutorial1j3.html.

[44] 2024. Soot. https://github.com/soot-oss/soot.

[45] 2024. sovcn/enterprise-routing-system. https://github.com/sovcn/enterprise-routing-system.

[46] 2024. ui-kreinhard/generica. https://github.com/ui-kreinhard/generica.

[47] 2024. Unresolved file references in @ImportResource locations. https://www.jetbrains.com/help/inspectopedia/SpringImportResource.html.

[48] 2024. WALA. https://github.com/wala/WALA.

[49] 2025. java lang exception no runnable methods. https://www.javatpoint.com/java-lang-exception-no-runnable-methods.

[50] 2025. PMD. https://pmd.github.io.

[51] Mona Attariyan and Jason Flinn. 2008. Using Causality to Diagnose Configuration Bugs. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, Berkeley, CA, USA, 281–286. http://dl.acm.org/citation.cfm?id=1404014.1404037

[52] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 237–250. http://dl.acm.org/citation.cfm?id=1924943.1924960

[53] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices* 38, 9 (2003), 51–63.

[54] Don Chamberlin. 2002. XQuery: An XML query language. *IBM systems journal* 41, 4 (2002), 597–615.

[55] Ian Darwin. 2009. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering* 2010 (2009).

[56] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. 2005. Using annotations to check structural properties of classes. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 237–252.

[57] Haruo Hosoya and Benjamin C Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)* 3, 2 (2003), 117–148.

[58] Walter L. Hürsch and Cristina Videira Lopes. 1995. *Separation of Concerns*. Technical Report.

[59] JavaParser. [n. d.]. JavaParser - Tools for your Java code. https://javaparser.org/. Accessed: 2024-07-21.

[60] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. 372–383.

[61] Carlos Noguera and Laurence Duchien. 2008. Annotation framework validation using domain models. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 48–62.

[62] Carlos Noguera, Andy Kellens, Coen De Roover, and Viviane Jonckers. 2012. Refactoring in the presence of annotations. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 337–346. https://doi.org/10.1109/ICSM.2012.6405291

[63] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. 2022. Mining Annotation Usage Rules: A Case Study with MicroProfile. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 553–562. https://doi.org/10.1109/ICSME55016.2022.00075

[64] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding broken Linux configuration specifications by statically analyzing the Kconfig language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 893–905. https://doi.org/10.1145/3468264.3468578

[65] Oracle. 2021. Overview of Enterprise Applications. https://docs.oracle.com/javaee/6/firstcup/doc/gcrky.html.

[66] Ariel Rabkin and Randy Katz. 2011. Precomputing Possible Configuration Error Diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 193–202. https://doi.org/10.1109/ASE.2011.6100053

[67] Myoungkyu Song and Eli Tilevich. 2012. Metadata Invariants: Checking and Inferring Metadata Coding Conventions. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 694–704.

[68] JUnit Team. 2024. JUnit 5. https://junit.org/junit5/ Accessed: 2024-07-25.

[69] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive System Configuration Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 625–636. http://dl.acm.org/citation.cfm?id=3155562.3155641

[70] Chengyuan Wen, Yaxuan Zhang, Xiao He, and Na Meng. 2020. Inferring and Applying Def-Use Like Configuration Couplings in Deployment Descriptors. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 672–683.

[71] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proc. ACM Softw. Eng.* 1, FSE, Article 33 (jul 2024), 23 pages. https://doi.org/10.1145/3643759

[72] Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 312–321. http://dl.acm.org/citation.cfm?id=2486788.2486830

Received 2024-09-05; accepted 2025-01-14